UNIVERSITY OF OSLO
Department of Informatics

Extending Kleene Algebra with Synchrony – technicalities [1]

Research Report No. 376

Cristian Prisacariu

Isbn 82-7368-336-2
Issn 0806-3036

October 2008

Extending Kleene Algebra with Synchrony – technicalities [†]

Cristian Prisacariu[‡]

October 2008

Abstract

The work reported here investigates the introduction of synchrony into Kleene algebra. The resulting algebraic structure is called *synchronous Kleene algebra*. Models are given in terms of regular sets of concurrent strings and finite automata accepting concurrent strings. The extension of synchronous Kleene algebra with Boolean tests is presented together with models on regular sets of guarded concurrent strings and the associated automata on guarded concurrent strings. Completeness w.r.t. the standard interpretations is given in each case. Decidability follows from completeness. A comparison with Mazurkiewicz traces is made which yields their incomparability with the synchronous Kleene algebra (one cannot simulate the other). We sketch applications to respectively deontic logic of actions and to Hoare-style reasoning about programs with synchrony.

[†]Partially supported by the Nordunet3 project "COSoDIS – Contract-Oriented Software Development for Internet Services" (`http://www.ifi.uio.no/cosodis/`).

[‡]Dept. of Informatics – Univ. of Oslo, P.O. Box 1080 Blindern, N-0316 Oslo, Norway. E-mail: cristi@ifi.uio.no

Contents

1 Introduction

Kleene algebra is a formalism used to represent and reason about programs. *Kleene algebra with tests* combines Kleene algebra with a Boolean algebra; it can express while programs [Koz00] and can encode the propositional Hoare logic using a Horn-style inference system. In one form or another, Kleene algebras appear in various formalisms in computer science: relation algebras, logics of programs and in particular Propositional Dynamic Logic [Pra79, Pra90], regular expressions and formal language theory [KS86].

In this paper we investigate the extension of Kleene algebra with a particular notion of concurrent actions/programs which adopts the synchrony model. *Synchrony* is a model of concurrency which was introduced in the process algebra community in R. Milner's SCCS [Mil83] but which detaches form the general interleaving approach. On the other hand it is a concept which does not belong to the partial order model of true concurrency either [Maz88, NPW79, Pra86]; we see in Section 4.1 how synchrony as we define here compares to Mazurkiewicz traces and pomsets. The synchrony concept proves highly expressive and robust; SCCS can represent CCS (i.e. asynchrony) as a subcalculus, and a great number of synchronizing operators can be defined in terms of the basic SCCS-Meije operators [dS85]. Meije is the calculus at the basis of the Esterel synchronous programming language [BC85]; Meije and SCCS operators are interdefinable.

The motivation for adding synchrony to Kleene algebra spawns from the need to reason about actions (where Kleene algebra is the equational tool of choice in conjunction with PDL) that can be executed in a truly concurrent fashion. We do not need such a powerful concurrency model like the ones based on partial orders; on the other hand the low level interleaving model is not well suited for (abstract) reasoning. The synchrony model has appealing equational representation and thus it is easy to integrate into the Kleene algebra. Moreover, the reasoning power and expressivity that synchrony offers is enough for the applications listed below.

1.1 Applications

Two applications are presented in Sections 2.4 and 3.4. The first application successfully uses synchronous Kleene algebra in the context of deontic logic of actions (in giving the semantics of the logical expressions). The second application uses the extension of synchronous Kleene algebra with tests in the context of Hoare logic for programs with synchrony. Other applications that we envisage are for giving semantics for Java threads and for giving an extension of propositional dynamic logic (PDL) with synchrony which would be an alternative to the PDL$^\cap$ [HKT00] or concurrent PDL [Pel85].

More general, wherever one uses Hoare logic to reason about programs one can use the more powerful Kleene algebra with tests which has also

tool support the KAT-ML prover [AHK06a, Koz00]. Moreover, one may safely choose the synchronous Kleene algebra with tests where in addition reasoning about concurrent executions is needed (similar to some extent to the current work of C.A.R. Hoare [Hoa07]). In these contexts (synchronous) Kleene algebra proves more powerful and more general than classical logical formalisms.

Our particular application of the work presented here (more precisely the formalism of Section 2.4) is to help in giving a direct semantics to the action-based contract-specification language $\mathcal{CL}$ [PS07].

We continue this introductory section with background material.

1.2 Kleene algebra

Kleene algebra ($\mathcal{KA}$) was named after S.C. Kleene who in the fifties [Kle56] studied regular expressions and finite automata. Kleene algebra formalizes axiomatically these structures. Further developments on the algebraic theory of $\mathcal{KA}$ were done by J.H. Conway [Con71]. For references and an introduction to Kleene algebra see the extensive work of D. Kozen [Koz79, Koz90, Koz97b]. Completeness of the axiomatization of $\mathcal{KA}$ was studied in [Sal66, Koz94], complexity in [CKS96], and applications to concurrency control, static analysis and compiler optimization, or pointer arithmetics in [Coh94, KP00, Koz03b, Möl97]. Some variants of $\mathcal{KA}$ include the notion of *tests* [Koz97b], and others add some form of types or discard the identity element $\mathbf{1}$ [Koz98].

Definition 1.1 Kleene algebra *is a structure* $(\mathcal{A}, +, \cdot, ^*, \mathbf{0}, \mathbf{1})$ *where* $+$ *and* $\cdot$ *are binary functional symbols (written infix),* * *is a unary (postfix) functional symbol, and* $\mathbf{0}$, $\mathbf{1}$ *are constants. The structure* $(\mathcal{A}, +, \cdot, \mathbf{0}, \mathbf{1})$ *forms an idempotent semiring (that means that* $(\mathcal{A}, +, \mathbf{0})$ *is an idempotent and commutative monoid,* $(\mathcal{A}, \cdot, \mathbf{1})$ *is a monoid with annihilator* $\mathbf{0}$*, and* $\cdot$ *distributes over* $+$*). We denote elements of* $\mathcal{A}$ *by* $\alpha, \beta, \gamma \ldots$*. For an idempotent semiring the partial order* $\leq_+$ *is defined as:*

$$\alpha \leq_+ \beta \stackrel{\Delta}{=} \alpha + \beta = \beta;$$

we call $\leq_+$ *the* preference relation*, and say that* β *is more preferable than* α*. It is easy to check that the three operators are monotone w.r.t.* $\leq_+$ *and that* $\alpha + \beta$ *is the least upper bound of* α *and* β *w.r.t.* $\leq_+$*. The additional requirements of a Kleene algebra concern the* * *operator (which captures the notion of repetition): the least solution w.r.t.* $\leq_+$ *to the equation* $\beta + \alpha \cdot X \leq_+ X$ *must be* $\alpha^* \cdot \beta$ *and dually* $\beta \cdot \alpha^*$ *must be the least solution for the equation* $\beta + X \cdot \alpha \leq_+ X$*.*

Notation: We call the syntactic terms of $\mathcal{A}$ *actions* and are constructed by the grammar:

$$\alpha \quad ::= \quad a \mid \mathbf{0} \mid \mathbf{1} \mid \alpha + \alpha \mid \alpha \cdot \alpha \mid \alpha^*$$

where $a \in \mathcal{A}_B$ are called *basic (or atomic) actions*. For each finite set of constants $\mathcal{A}_B$ consider the term algebra $T_{\mathcal{KA}}(\mathcal{A}_B)$ associated with the signature $\sigma = \{+, \cdot, ^*, \mathbf{0}, \mathbf{1}, \mathcal{A}_B\}$. Thus we consider a family of algebras indexed by the finite set of basic actions $\mathcal{A}_B$. We will be concerned only with a representant $(\mathcal{A}, +, \cdot, ^*, \mathbf{0}, \mathbf{1}, \mathcal{A}_B)$ of this family of algebras for a fixed finite set of basic actions $\mathcal{A}_B \in \mathcal{A}$. The term algebra $T_{\mathcal{KA}}(\mathcal{A}_B)$ is free in the corresponding class of algebras over the generators of $\mathcal{A}_B \cup \{\mathbf{0}, \mathbf{1}\}$. We will just use $T_{\mathcal{KA}}$ whenever $\mathcal{A}_B$ is understood from context. Note that $T_{\mathcal{KA}}(\mathcal{A}_B)$ corresponds to the set of regular expressions over the alphabet $\mathcal{A}_B$.

Examples of Kleene algebras: Consider, in language theory, Σ^* the set of all finite words over the alphabet Σ [HMU00]. Over the powerset $\mathcal{P}(\Sigma^*)$ we can define:

$$\begin{array}{rcl} \mathbf{0} & \triangleq & \emptyset \\ \mathbf{1} & \triangleq & \{\varepsilon\} \\ \alpha + \beta & \triangleq & \alpha \cup \beta \\ \alpha \cdot \beta & \triangleq & \{xy \mid x \in \alpha, y \in \beta\} \\ \alpha^* & \triangleq & \bigcup_{n \geq 0} \alpha^n \end{array}$$

where $\alpha, \beta \in \mathcal{P}(\Sigma^*)$ are sets of words, and $x, y \in \Sigma^*$ denote words. Moreover we have:

$$\begin{array}{rcl} \alpha^0 & \triangleq & \{\varepsilon\} \\ \alpha^n & \triangleq & \alpha \cdot \alpha^{n-1}. \end{array}$$

By convention when $\alpha = \emptyset$ then $\alpha^* = \{\varepsilon\}$. Thus α^* always contains the word ε and is the concatenation of all the powers of the set α. This operation is called the *Kleene star*. Any subset of $\mathcal{P}(\Sigma^*)$ which contains $\emptyset$ and $\{\varepsilon\}$ and is closed under the three operations defined above forms a Kleene algebra. Moreover, take an element of the family of algebras to contain $\mathcal{A}_B = \{\{a\} \mid a \in \Sigma\}$; in the terminology of S.C. Kleene this is called the algebra of *regular sets*.

For a second example consider the set of all *binary relations* over a set X. In this case $\alpha, \beta \subseteq X \times X$ are relations over X. Consider the following definitions:

$$\begin{array}{rcl} \mathbf{0} & \triangleq & \emptyset \\ \mathbf{1} & \triangleq & \{(x, x) \mid x \in X\} \\ \alpha + \beta & \triangleq & \alpha \cup \beta \\ \alpha \cdot \beta & \triangleq & \{(x, y) \mid \exists z \in X \text{ s.t. } (x, z) \in \alpha \text{ and } (z, y) \in \beta\} \\ \alpha^* & \triangleq & \bigcup_{n \geq 0} \alpha^n \end{array}$$

where we have:

$$\begin{array}{lcl} \alpha^0 & \triangleq & \{(x,x) \mid x \in X\} \\ \alpha^n & \triangleq & \alpha \cdot \alpha^{n-1}. \end{array}$$

Note the associations of $\mathbf{1}$ with the *identity relation*, $+$ with the union of relations, $\cdot$ with the *relational composition*, and * with the *transitive and reflexive closure* of a relation. Any set of relations, which contains $\emptyset$ and the identity relation, and is closed under the above operations over relations forms a (relational) Kleene algebra. This algebra is used in the semantics of logics of programs; like Propositional Dynamic Logic [FL77, HKT00].

As a last example consider the *min,+ algebra* (also called the tropical algebra) which is useful in shortest path algorithms on graphs [Koz97a]. The operations are defined over the domain $\mathbb{R}_+ \cup \{\infty\}$. The $+$ operation from Kleene algebra is defined as the *min* operation on reals giving the minimum of two elements under the natural order on $\mathbb{R}_+ \cup \{\infty\}$ where ∞ is alway the greatest. The operation $\cdot$ is interpreted as the $+$ on $\mathbb{R}_+ \cup \{\infty\}$. The two constants $\mathbf{0}$ and $\mathbf{1}$ are interpreted respectively as ∞ and 0. The * operation is surprisingly defined as $x^* = 0$. The related *max,+ algebra* (called also the *arctic semiring*) is also a Kleene algebra used in proving termination of rewriting systems [KW08].

Note that for the first two example above the preference relation $\leq_+$ is defined to be set inclusion $\subseteq$ where for the last example it is the reverse of the natural order on reals.

1.3 Synchrony

The notion of *synchrony* has different meanings in different areas of computer science. Here we take the distinction between *synchrony* and *asynchrony* as presented in the SCCS calculus of [Mil83] and later implemented in e.g. the Esterel synchronous programming language [BC85, BG92]. We understand *asynchrony* as when two concurrent systems execute at indeterminate relative speeds (i.e. their actions may have different noncorelated durations); whereas in the *synchrony* model each of the two concurrent systems execute instantaneously a single action at each time instant.

The *synchrony model* takes the assumption that time is discrete and that basic actions are instantaneous (i.e. take zero time and represent the time step). Moreover, at each time step, all possible actions are performed, i.e. the system is considered *eager* (idling is not possible). Synchrony assumes a global clock which provides the time unit for all the actors in the system. Note that for practical purposes this is a rather strong assumption which is in contrast with the popular view from process algebras [Mil95,Hoa85]. On the other hand, the experience of the Esterel implementation and use in industry contradict the general belief. Moreover, the equational framework of the synchrony model is much cleaner and more general than the asynchronous interleaving model (SCCS contains CCS as a subcalculus [Mil83]).

SCCS introduces a synchronous composition operator $\times$ over processes which is different from the classical $\|$ of CCS. The operational semantics of $\times$ (SCCS keeps the process algebra style of giving meaning to processes using Structural Operational Semantics) is:

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{b} Q'}{P \times Q \xrightarrow{ab} P' \times Q'}$$

Moreover, the set of actions Act forms a commutative monoid $(Act, \times, 1)$ under $\times$ with a special *idle* action 1 which is the identity element. Note that $\times$ is omitted in favor of juxtaposition when we use it for actions. Equality between processes is defined with the use of bisimulation equivalence. The quotient of the set of processes under the congruence derived from the bisimulation enjoys nice algebraic properties like the distributivity of $\times$ over $+$ (the nondeterministic choice), or the process with *no action* is an annihilator for the $\times$ operation.

2 Kleene Algebra with Synchrony

We add to the standard Kleene algebra of Section 1.2 an operator to model *concurrency* similar to the synchronous composition of SCCS presented in Section 1.3. We call the resulting algebra *synchronous Kleene algebra* ($\mathcal{SKA}$). The $\mathcal{SKA}$ algebra has the following particularities:

1. Formalizes a notion of *concurrent actions* based on the synchrony model.

2. Has a standard interpretation of the actions over *(guarded) concurrent strings*. The actions can be represented as special finite automata which accept the same sets of concurrent strings that form the models of the actions.

3. Incorporates the notion of *conflicting actions*.

In this section we are more precise about the notions presented and we give related intuitions about their application. Note that influenced by our long term goal which intends to apply $\mathcal{SKA}$ to the contract language $\mathcal{CL}$, we talk about (human-like) *actions* and give intuitions for the properties of the $\mathcal{SKA}$ algebra in terms of actions (as found in legal contracts).

2.1 Syntax and axiomatization

Definition 2.1 *Consider a family of algebras indexed by the finite set of basic (or atomic) actions $\mathcal{A}_B$. We investigate a general member $\mathcal{SKA}(\mathcal{A}_B)$ of this family for which the set $\mathcal{A}_B$ is fixed and denotes a finite set of constant symbols of the signature. Henceforth we write only $\mathcal{SKA}$ for this algebraic*

structure. $\mathcal{SKA}$ is a σ-algebra with signature $\sigma = \{+, \cdot, \times, ^*, \mathbf{0}, \mathbf{1}, \mathcal{A}_B\}$ which gives the action operators and the basic actions. The non-constant functions of σ are: "$+$" for choice of two actions, "$\cdot$" for sequence of two actions (or concatenation), "$\times$" for concurrent composition of two actions, and "*" to model repeated execution of one action. The special elements $\mathbf{0}$ and $\mathbf{1}$ are constant function symbols. The operators of $\mathcal{SKA}$ are defined over a carrier set of elements (called compound actions, or just actions) denoted $\mathcal{A}$.

We are more particular in our study, as defined below.

Definition 2.2 (Syntax) *The term algebra $T_{\mathcal{SKA}}(\mathcal{A}_B)$ is free in the corresponding class of algebras $\mathcal{SKA}(\mathcal{A}_B)$ over the generators of $\mathcal{A}_B \cup \{\mathbf{0}, \mathbf{1}\}$ (it is more desirable to work with the initial algebra of a class of algebras; which $T_{\mathcal{SKA}}(\mathcal{A}_B)$ is). In this context the elements of $\mathcal{A}$ (i.e. the actions) are the terms constructed with the grammar below:*

$$\alpha \quad ::= \quad a \mid \mathbf{0} \mid \mathbf{1} \mid \alpha + \alpha \mid \alpha \cdot \alpha \mid \alpha \times \alpha \mid \alpha^*$$

where $a \in \mathcal{A}_B$. The basic actions of $\mathcal{A}_B$ and $\mathbf{0}$ and $\mathbf{1}$ have the property that they cannot be generated from other actions of $\mathcal{A}$. The operators $+$, $\cdot$, $\times$, and * are sometimes called constructors because they are used to construct all the actions of $\mathcal{A}$. We just use $T_{\mathcal{SKA}}$ whenever $\mathcal{A}_B$ is understood from context.

Notation: Throughout this paper we denote by $a, b, c, \ldots$ elements of $\mathcal{A}_B$ (basic actions) and by $\alpha, \beta, \gamma, \ldots$ elements of $\mathcal{A}$ (compound actions). When the difference between basic and compound actions is not important we just call them generically *actions*. We call *concurrent actions* and denote by $\mathcal{A}_B^{\times} \subset \mathcal{A}$ the subset of elements of $\mathcal{A}$ generated from $\mathcal{A}_B$ using only $\times$ constructor (e.g. a, $a \times b \in \mathcal{A}_B^{\times}$ but $a + b, a \times b + c, a \cdot b \notin \mathcal{A}_B^{\times}$ and $\mathbf{0}, \mathbf{1} \notin \mathcal{A}_B^{\times}$). We denote by $\alpha_{\times}$ a general element of $\mathcal{A}_B^{\times}$. Note that $\mathcal{A}_B^{\times}$ is finite because there is a finite number of basic actions in $\mathcal{A}_B$ which may be combined with the concurrency operator $\times$ in a finite number of ways (due to the idempotence of $\times$ over basic actions; see axiom (14) of Table 1). Note the inclusion of sorts $\mathcal{A}_B \subseteq \mathcal{A}_B^{\times} \subset \mathcal{A}$. We call action $\mathbf{1}$ the *skip* action and $\mathbf{0}$ the *violating* action. (In other works $\mathbf{1}$ is called *idle* or *void* action, and $\mathbf{0}$ the *fail* action.) For brevity we often drop the sequence operator and instead of $\alpha \cdot \beta$ we write $\alpha\beta$ (do not confuse with the convention from SCCS). To avoid unnecessary parentheses we use the following precedence over the constructors: $+ < \cdot < \times < {}^*$.

Definition 2.3 (Axioms) *Table 1 collects the axioms that define the structure $\mathcal{SKA}$.*

(1)	$\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$	(10)	$\alpha \times (\beta \times \gamma) = (\alpha \times \beta) \times \gamma$
(2)	$\alpha + \beta = \beta + \alpha$	(11)	$\alpha \times \beta = \beta \times \alpha$
(3)	$\alpha + \mathbf{0} = \mathbf{0} + \alpha = \alpha$	(12)	$\alpha \times \mathbf{1} = \mathbf{1} \times \alpha = \alpha$
(4)	$\alpha + \alpha = \alpha$	(13)	$\alpha \times \mathbf{0} = \mathbf{0} \times \alpha = \mathbf{0}$
(5)	$\alpha \cdot (\beta \cdot \gamma) = (\alpha \cdot \beta) \cdot \gamma$	(14)	$a \times a = a \quad \forall a \in \mathcal{A}_B$
(6)	$\alpha \cdot \mathbf{1} = \mathbf{1} \cdot \alpha = \alpha$	(15)	$\alpha \times (\beta + \gamma) = \alpha \times \beta + \alpha \times \gamma$
(7)	$\alpha \cdot \mathbf{0} = \mathbf{0} \cdot \alpha = \mathbf{0}$	(16)	$(\alpha + \beta) \times \gamma = \alpha \times \gamma + \beta \times \gamma$
(8)	$\alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma$	(17)	$(\alpha_\times \cdot \alpha) \times (\beta_\times \cdot \beta) = (\alpha_\times \times \beta_\times) \cdot (\alpha \times \beta) \quad \forall \alpha_\times, \beta_\times \in \mathcal{A}_B^\times$
(9)	$(\alpha + \beta) \cdot \gamma = \alpha \cdot \gamma + \beta \cdot \gamma$		
(18)	$\mathbf{1} + \alpha \cdot \alpha^* \leq_+ \alpha^*$	(20)	$\beta + \alpha \cdot \gamma \leq_+ \gamma \;\rightarrow\; \alpha^* \cdot \beta \leq_+ \gamma$
(19)	$\mathbf{1} + \alpha^* \cdot \alpha \leq_+ \alpha^*$	(21)	$\beta + \gamma \cdot \alpha \leq_+ \gamma \;\rightarrow\; \beta \cdot \alpha^* \leq_+ \gamma$

Table 1: Axioms of $\mathcal{SKA}$

Remarks: The axioms (1)-(4) define the choice operator $+$ to be associative, commutative, with neutral element $\mathbf{0}$, and idempotent (i.e. a commutative and idempotent monoid). Axioms (5)-(7) define the sequence operator $\cdot$ to be associative, with neutral element $\mathbf{1}$, and with annihilator $\mathbf{0}$ both on the left and right side (i.e. a monoid with an annihilator element). We call the two equations *fail late* (for $\alpha \cdot \mathbf{0} = \mathbf{0}$) and *fail soon* (for $\mathbf{0} \cdot \alpha = \mathbf{0}$). Axioms (8) and (9) give the distributivity of $\cdot$ over $+$. These give the algebraic structure of an idempotent semiring $(\mathcal{A}, +, \cdot, \mathbf{0}, \mathbf{1})$. Axioms (10)-(13) give the properties of $\times$ to be associative, commutative, with identity element $\mathbf{1}$, and annihilator element $\mathbf{0}$ (i.e. $(\mathcal{A}, \times, \mathbf{1}, \mathbf{0})$ is a commutative monoid with an annihilator element). Axioms (10) and (11) basically say that the syntactic ordering of actions in a concurrent action does not matter (the same as for choice $+$). Axiom (14) defines $\times$ to be idempotent over the basic actions $a \in \mathcal{A}_B$. Note that this does *not* imply that we have an idempotent monoid. Axioms (15) and (16) define the distributivity of $\times$ over $+$. From axioms (10)-(16) together with (1)-(4) we conclude that $(\mathcal{A}, +, \times, \mathbf{0}, \mathbf{1})$ is a commutative and idempotent semiring (NB: idempotence comes from axiom (4), and the axiom (14) is just an extra property of the semiring). The equations (18) and (19) and equational implications (20) and (21) are the classical axiomatization of * [Sal66, Koz94] which say that $\alpha^*\beta$ is the least solution w.r.t. the preference relation $\leq_+$ from Definition 1.1 to the equation $\beta + \alpha X \leq_+ X$ (and dually $\beta\alpha^*$ is the least solution for the equation $\beta + X\alpha \leq_+ X$).

At this point we give an informal intuition for the elements (actions) of $\mathcal{A}$: we consider that the actions are "performed" by somebody (being that a person, a program, or an agent). One should not be think exclusively of processes "executing" instructions as this is only one way of viewing the actions. Moreover, we do not discuss operational semantics nor bisimulation equivalences (like in SCCS) in this paper.

With this non-algebraic intuition of actions we can elaborate on the purpose of $\times$, which models the fact that two actions are performed in a truly concurrent manner. Particular to $\times$ is the axiom (14) which defines a weak form of idempotence for the concurrency operator. The idempotence is natural for basic actions but it is not desirable for complex actions. Take as example a choice action performed synchronously with itself, $(a+b)\times(a+b)$. The first entity may choose a and the second entity may choose b therefore performing the synchronous action $a\times b$. Therefore, the complex action is the same as $a+a\times b+b$ (by the distributivity axiom (15)). Particular to our concurrency model is axiom (17) which synchronizes sequences of actions by working in steps given by the $\cdot$ constructor. This encodes the synchrony model.

Note that there is no axiom relating the $\times$ with the Kleene star. There is no need as the relation is done by the synchrony axiom (17) and the fact that $\alpha^* = \mathbf{1} + \alpha \cdot \alpha^*$ from the axioms of *. Moreover, when combining two repetitive actions concurrently $\alpha^*\times\beta^*$ the synchrony operator will go inside the * operator. This results in an action inside * which has a maximum of $|\alpha| * |\beta|$ steps (i.e. number of $\cdot$ applications). This is strong related to the dimension of the automaton constructed in Theorem 2.5 to handle the $\times$ (the size of the new automaton is the size of the cartesian product of the two smaller automata; i.e. is the product of the number of states).

Definition 2.4 (demanding relation) *We call $<_\times$ the demanding relation and define it as:*

$$\alpha <_\times \beta \triangleq \alpha\times\beta = \beta \tag{1}$$

We say that β is more demanding than α iff $\alpha <_\times \beta$. Note that the least demanding action is $\mathbf{1}$ (skiping means not doing any action). On the other hand, if we do not consider $\mathbf{1}$ then we have the basic actions of $\mathcal{A}_B$ as the least demanding actions; the basic actions are not related to each other by $<_\times$. We denote by $\leq_\times$ the relation $<_\times \cup =$; i.e. $\alpha \leq_\times \beta$ iff either $\alpha <_\times \beta$ or $\alpha = \beta$.

Proposition 2.1 *The relation $<_\times|_{\mathcal{A}_B^\times}$ is a partial order.*

Proof : For the relation $<_\times$ restricted to concurrent actions of $\mathcal{A}_B^\times$ the reflexivity is assured by the weak idempotence axiom (14). The transitivity and antisymmetry are immediate and moreover, they hold for the whole set $\mathcal{A}$ of actions; e.g. for *transitivity* take any $\alpha, \beta, \gamma \in \mathcal{A}$ s.t. $\alpha <_\times \beta$ and $\beta <_\times \gamma$. Then it is the case that from $\alpha\times\beta = \beta$ and $\beta\times\gamma = \gamma$ we get $\alpha\times\gamma = \alpha\times\beta\times\gamma = \beta\times\gamma = \gamma$ which is the desired conclusion $\alpha <_\times \gamma$ (note that we used the commutativity of the $\times$ operation and the transitivity of the equality of actions).

Note that reflexivity is not a property of the general actions, but only of the concurrent actions. Therefore, ireflexivity is not a property of the

general actions either. For example $(a+b)\times(a+b) = a+b+a\times b$ but on the other hand $(a+b+a\times b)\times(a+b+a\times b) = a+b+a\times b$ due to the idempotence of the $\times$ over $\mathcal{A}_B$. Moreover, we can prove that for any action α there exists a fix point for the application of the $\times$ to the action itself. In other words, define $\beta_0 = \alpha$ and $\beta_i = \beta_{i-1}\times\alpha$, then $\exists n \in \mathbb{N}$ and $\exists j < n$ s.t. $\beta_j = \beta_n$. The proof uses the canonical representation of the action α. □

For a better intuitive understanding consider the following examples: $\mathbf{1} <_\times a$, $a <_\times a\times b$, $a \not<_\times a$, $a \leq_\times a$ $a+b \not<_\times a+b$, $a \not<_\times b$, and $a \not<_\times b\times c$.

Proposition 2.2 *1. If* $\alpha_\times^1 <_\times \beta_\times^1 \wedge \cdots \wedge \alpha_\times^n <_\times \beta_\times^n$ *then* $\alpha_\times^1 \cdot \cdots \cdot \alpha_\times^n <_\times \beta_\times^1 \cdot \cdots \cdot \beta_\times^n \cdot \gamma$

where $\alpha_\times^i, \beta_\times^j \in \mathcal{A}_B^\times$ *and* $\gamma \in \mathcal{A}$.

2. If $\alpha_\times^i <_\times \beta_\times^j, \forall i \leq n, j \leq m$ *then* $(\alpha_\times^1 + \cdots + \alpha_\times^n) <_\times (\beta_\times^1 + \cdots + \beta_\times^m)$.

Proof: Routine.

For the first part of the proposition, the hypothesis is translated to $\alpha_\times^1 \times \beta_\times^1 = \beta_\times^1, \ldots, \alpha_\times^n \times \beta_\times^n = \beta_\times^n$. For the conclusion we have $(\alpha_\times^1 \cdot \cdots \cdot \alpha_\times^n)\times(\beta_\times^1 \cdot \cdots \cdot \beta_\times^n \cdot \gamma)$ which by the synchrony axiom (17) combines the $\alpha_\times^i$ and $\beta_\times^i$ two by to and by the hypothesis we get that it is equal to $\beta_\times^1 \cdot \cdots \cdot \beta_\times^n \cdot \gamma$.

The proof of the second part of the proposition is similar. It makes use of the distributivity axiom (15) first, then uses the hypothesis in the same manner as before, and finally it contracts the same actions $\beta_\times^i$ with the axiom (4) of idempotence of $+$. □

Remark: For practical purposes the demanding relation can be used to give a partial decision methodology for the nondeterministic choice in the following way. One just adds to the axioms of $\mathcal{SKA}$ the following equational implication as an axiom:

$$\alpha <_\times \beta \rightarrow \beta \leq_+ \alpha$$

The new axiom is read as: "the less demanding action is the more preferable one". Depending on the application domain the order on the right part of the axiom can be reversed so that to read the opposite: "the less demanding action is the less preferable one". We do not pursue further this line of research.[1]

Definition 2.5 (conflict and compatibility) *We consider a symmetric and irreflexive relation over the set of basic actions* $\mathcal{A}_B$ *which we call conflict relation and denote by* $\#_\mathcal{C} \subseteq \mathcal{A}_B \times \mathcal{A}_B$. *The converse relation of* $\#_\mathcal{C}$ *is the*

[1] One immediate investigation is to make sure that this new axiom complies with the other axioms of the algebra (i.e. does not contradict the axioms of Table 1).

symmetric and reflexive compatibility relation which we denote by $\sim_{\mathcal{C}}$ *and is defined as*

$$\sim_{\mathcal{C}} \triangleq \mathcal{U}_B \setminus \#_{\mathcal{C}}$$

where $\mathcal{U}_B = \mathcal{A}_B \times \mathcal{A}_B$ *is the universal relation over basic actions.*

The intuition of the conflict relation is that if two actions are in conflict then the actions cannot be executed concurrently. This intuition explains the need for the following equational implication:

$$(22) \quad a \#_{\mathcal{C}} b \rightarrow a \times b = \mathbf{0} \qquad \forall a, b \in \mathcal{A}_B.$$

The intuition of the compatibility relation is that if two actions are compatible then the actions can always be executed concurrently. There is *no transitivity* of $\#_{\mathcal{C}}$ or $\sim_{\mathcal{C}}$: In general, if $a \#_{\mathcal{C}} b$ and $b \#_{\mathcal{C}} c$, not necessarily $a \#_{\mathcal{C}} c$. This is natural as action b may be in conflict with both a and c but still $a \sim_{\mathcal{C}} c$.

Remark: R. Milner [Mil83] suggests a natural notion of *inverse on an action* w.r.t. the synchrony operation; $a \times \tilde{a} = \mathbf{1}$. In programming languages this means that the action (which is a program instruction in this case) and the inverse work on the same ressource. It is known that this is impossible and many solutions have been proposed; see separation logic [IO01,Rey02] or Owicki and Gries [OG76]. In process algebras this leads to the interleaving diamond because $a \times \tilde{a} = \mathbf{1}$ it means that also $a \cdot \tilde{a} = \mathbf{1}$ and $\tilde{a} \cdot a = \mathbf{1}$ which means $(a \cdot \tilde{a}) + (\tilde{a} \cdot a) = \mathbf{1} = a \times \tilde{a}$.

Outside programming languages, when considering the actions as human actions in legal contracts or deontic logic (see Section 2.4), the idea of an inverse of an action becomes more plausible. It may be that one basic action compensates another basic action which means that when performed at the same time the state of the world will not change. For example "Redraw X money from account A" and "Deposit X money into account A" are both one the inverse of the other. We must consider these actions at a higher level of abstraction where the details of the implementation are ignored.

2.2 Standard interpretation over regular concurrent sets

We give the *standard interpretation* of the actions of $\mathcal{A}$ by defining a *homomorphism* $\hat{I}_{\mathcal{SKA}}$ which takes any action of the $\mathcal{SKA}$ algebra into a corresponding *regular concurrent set* and preserves the structure of the actions given by the constructors.

Definition 2.6 (regular concurrent sets) *Consider a finite set of elements, for our purpose we denote it* $\mathcal{A}_B$ *and can be thought of as the basic actions. Consider a finite alphabet* $\Sigma = \mathcal{P}(\mathcal{A}_B)$ *consisting of all the sets over*

$\mathcal{A}_B$ (denote them $x, y \in \mathcal{P}(\mathcal{A}_B)$). Concurrent strings over $\mathcal{A}_B$ are elements of $\mathcal{P}(\mathcal{A}_B)^*$ including the empty string $\{\}$ (denote them $u, v, w \in \mathcal{P}(\mathcal{A}_B)^*$). A concurrent set is a subset of strings from $\mathcal{P}(\mathcal{A}_B)^*$. We denote concurrent sets by A, B, C. Consider the following definitions and operations on concurrent sets:

$$\begin{aligned} \mathbf{0} &\triangleq \emptyset \\ \mathbf{1} &\triangleq \{\{\}\} \\ A + B &\triangleq A \cup B \\ A \cdot B &\triangleq \{uv \mid u \in A, v \in B\} \\ A \times B &\triangleq \{u \times v \mid u \in A, v \in B\} \\ A^* &\triangleq \bigcup_{n \geq 0} A^n \end{aligned}$$

where $u, v \in \mathcal{P}(\mathcal{A}_B)^*$ are concurrent strings and $u \times v$ is defined as:

$$\begin{aligned} u \times v \triangleq\ & (x_1 \cup y_1)(x_2 \cup y_2) \ldots (x_n \cup y_n) y_{n+1} \ldots y_m \\ & \text{where } u = x_1 \ldots x_n \text{ and } v = y_1 \ldots y_m \text{ and } n < m, \end{aligned}$$

and where $x_i, y_j \in \mathcal{P}(\mathcal{A}_B)$ are sets of elements of $\mathcal{A}_B$. Recall the convention $u\varepsilon = u$ ($\varepsilon u = u$) from formal language theory which in our concurrent sets translates to $u\{\} = u$ ($\{\}u = u$). A concurrent set is called regular iff it is build from $\mathbf{0}$, $\mathbf{1}$, and the singleton sets $\{x\}$ with $x \in \mathcal{A}_B$ a concurrent string of length 1, by repeated application of the operations on sets $+, \cdot, \times, ^*$ (NB: $\mathcal{A}_B \subset \mathcal{P}(\mathcal{A}_B) \subset \mathcal{P}(\mathcal{A}_B)^*$, i.e. all basic actions are elements of the alphabet which are also strings of length 1). The powers of a set A^n are defined as in Section 1.2 with similar conventions.

Theorem 2.3 *Any set of concurrent sets containing $\mathbf{0}$, $\mathbf{1}$, and the singleton sets $\{x\}$ for all $x \in \mathcal{A}_B$ (for some fixed set $\mathcal{A}_B$) and closed under the operations $+, \cdot, \times, ^*$ of Definition 2.6 is a synchronous Kleene algebra. Call these the class of algebras of concurrent sets.*

Proof: Routine check that the operations of Definition 2.6 obey the axioms of $\mathcal{SKA}$ from Table 1. Particular care needs to be taken for axioms (14) and (17) as they are defined on particular elements. In the case of regular concurrent sets axiom (14) is defined only on the singleton sets $\{\{x\} \mid x \in \mathcal{A}_B\}$, and axiom (17) is defined only on singleton concurrent sets $\{\{x\} \mid x \in \mathcal{P}(\mathcal{A}_B)\}$.

The $+$ operation over concurrent sets respects axioms (1),(2), (3), and (4) because it is defined using the union operation $\cup$ over sets and $\mathbf{0}$ is defined as the empty set. The $\cdot$ operation over concurrent sets respects (5); take a concurrent string from the left part of the axiom $u_A u_B u_C$. This comes from combining first $u_B u_C$ with $u_B \in B$ and $u_C \in C$ two arbitrary concurrent strings and then concatenating an arbitrary concurrent string $u_A \in A$ with $u_B u_C$. This same concurrent string $u_A u_B u_C$ is included in the right part

of the axiom too: first concatenate the two $u_A \in A$ and $u_B \in B$ to obtain $u_A u_B$ and then by making all the combinations with the concurrent strings of C we get to concatenate with $u_C \in C$ also, to obtain $u_A u_B u_C$. The $\cdot$ respects (6) because of the convention $u\{\} = u$ and it respects (7) because there exists no combination uv with $v \in \emptyset$.

The distributivity axioms (8) and (9) are respected. On the right side of the axiom each concurrent string of A is concatenated with all the concurrent strings of the union of B with C. This is the same as concatenating first with the concurrent strings of B and then concatenating with the concurrent strings of C and in the end making the union of the resulting sets of concurrent strings.

To prove the associativity axiom (10) for the $\times$ operation over concurrent sets it is used a similar reasoning as in the proof for the $\cdot$ operation. For the commutativity axiom (11) the proof is simple by observing that the $\times$ operation of concurrent strings is commutative because it uses the union operation $\cup$ at each element of the string. The axioms (12) and (13) are simple from the definitions of the $\mathbf{1}$ and $\mathbf{0}$ respectively. The special axiom (14) holds because of the union operation that is used by the $\times$ on the concurrent strings. The argument for the distributivity axiom (15) and (16) is the same as for the $\cdot$ and $+$ operations.

For the synchrony axiom (17) the proof makes use of the fact that we need to consider only the special singleton concurrent sets $\{\{x\} \mid x \in \mathcal{P}(\mathcal{A}_B)\}$. Take two arbitrary singleton concurrent sets $\{x\}$ and $\{y\}$ and first concatenate them with all the concurrent sets of respectively A and B. This gives concurrent strings of the form xu_A respectively yu_B. Now combine all these concurrent strings among eachother to obtain $(x \cup y)(u_A \times u_B)$. The same concurrent strings are obtained on the right part of the axiom by making first the combination $\{x\} \times \{y\}$ with is the same as $x \cup y$. Then making the combination $A \times B$ which gives all the concurrent strings $u_A \times u_B$. The concatenation then gives the same set of concurrent strings as on the left part of the axiom.

The axioms (18)-(21) hold because the definition of the * is classic as in the formal language definition from Section 1.2. □

Remarks: The smallest algebra of concurrent sets which contains all the singleton sets $\{x\}$ is called the *algebra of regular concurrent sets* (denote it by $\mathcal{ARCS}(\mathcal{A}_B)$). Henceforth we ignore $\mathcal{A}_B$ from the notations and it should be understood as implicit. $\mathcal{ARCS}$ is generated by $\{\mathbf{0}, \mathbf{1}\} \cup \{\{x\} \mid x \in \mathcal{A}_B\}$, where $\mathcal{A}_B$ is some finite and fixed beforehand set.

Definition 2.7 (standard interpretation) *An interpretation is a homomorphism with domain the term algebra $T_{\mathcal{SKA}}$. We call standard interpretation the homomorphism $\hat{I}_{\mathcal{SKA}} : T_{\mathcal{SKA}} \to \mathcal{ARCS}$. $\hat{I}_{\mathcal{SKA}}$ is defined as the*

homomorphic extension of the map $I_{\mathcal{SKA}} : \mathcal{A}_B \cup \{\mathbf{0}, \mathbf{1}\} \rightarrow \mathcal{ARCS}$ *to the whole actions of* $T_{\mathcal{SKA}}$. $I_{\mathcal{SKA}}$ *maps the generators of* $T_{\mathcal{SKA}}$ *into regular concurrent sets as follows:*

$$
\begin{aligned}
I_{\mathcal{SKA}}(a) &= \{\{a\}\},\ \forall a \in \mathcal{A}_B \\
I_{\mathcal{SKA}}(\mathbf{0}) &= \emptyset \\
I_{\mathcal{SKA}}(\mathbf{1}) &= \{\{\}\}
\end{aligned}
$$

The homomorphic extension is classical:

$$
\begin{aligned}
\hat{I}_{\mathcal{SKA}}(\alpha) &= I_{\mathcal{SKA}}(\alpha),\ \forall \alpha \in \mathcal{A}_B \cup \{\mathbf{0}, \mathbf{1}\} \\
\hat{I}_{\mathcal{SKA}}(\alpha + \beta) &= \hat{I}_{\mathcal{SKA}}(\alpha) + \hat{I}_{\mathcal{SKA}}(\beta) \\
\hat{I}_{\mathcal{SKA}}(\alpha \cdot \beta) &= \hat{I}_{\mathcal{SKA}}(\alpha) \cdot \hat{I}_{\mathcal{SKA}}(\beta) \\
\hat{I}_{\mathcal{SKA}}(\alpha \times \beta) &= \hat{I}_{\mathcal{SKA}}(\alpha) \times \hat{I}_{\mathcal{SKA}}(\beta) \\
\hat{I}_{\mathcal{SKA}}(\alpha^*) &= \hat{I}_{\mathcal{SKA}}(\alpha)^*
\end{aligned}
$$

Informally the skip action **1** means not performing any action and its interpretation as the set with only the empty string goes well with the intuition. The violating action **0** is interpreted as the empty concurrent set following the intuition that there is no way of respecting a violating action.

Remark: The standard interpretation interprets a concurrent action $\alpha_\times \in \mathcal{A}_B^\times$ with $\alpha_\times = a^1 \times \ldots \times a^n$ as the singleton set $\{\{a^1, \ldots, a^n\}\}$ where the only string $w = \{a^1, \ldots, a^n\}$ has just one element of the alphabet $\mathcal{P}(\mathcal{A}_B)$ (i.e. the set of basic actions which form the concurrent action $\alpha_\times$). Denote the set w by $\{\alpha_\times\}$. In this particular case the demanding relation restricted to concurrent actions, i.e. $<_\times|_{\mathcal{A}_B^\times}$ is interpreted as set inclusion. That is: $\alpha_\times <_\times \beta_\times$ iff $\{\alpha_\times\} \subseteq \{\beta_\times\}$.

2.3 Completeness and Decidability

Regular expressions and finite state automata (FA) are equivalent syntactic representations of regular languages (or regular sets as we call them) [HMU00]. In this section we define finite state automata that accept regular concurrent sets and prove an equivalent of Kleene's representation theorem. That is, for each action of $\mathcal{SKA}$ we can build an equivalent automaton which accepts the same regular concurrent set as the interpretation of the action. Using the translation as automata we give a combinatorial proof of completeness of the $\mathcal{SKA}$ in terms of the standard interpretation. Decidability follows then naturally from completeness and decidability of the inclusion problem for regular concurrent sets.

Definition 2.8 (finite state automata on concurrent sets) *A* (nondeterministic) finite state automaton on regular concurrent sets *(NFA) is a tuple* $A = (S, \mathcal{P}(\mathcal{A}_B), S_0, \rho, F)$ *consisting of a finite set of* states S, *the*

finite alphabet of concurrent actions $\mathcal{P}(\mathcal{A}_B)$ *(i.e. the powerset of the set of basic actions* $\mathcal{A}_B$*), a set of* initial *designated states* $S_0 \subseteq S$*, a* transition relation $\rho : \mathcal{P}(\mathcal{A}_B) \to S \times S$*, and a set of final states* F*. A NFA is called* deterministic finite state automaton *(DFA) iff* $|S_0| = 1$ *and the transition relation returns not a relation over* S *but a partial function, i.e.* $\rho : \mathcal{P}(\mathcal{A}_B) \to (S \to S)$*.*

Notation: We denote the states of the automata by either $s_i \in S$ or by $i \in S$ with $i \in \mathbb{N}$. Instead of the transition relation $(s_1, s_2) \in \rho(\{\alpha_\times\})$ we often use the graphical notation $s_1 \xrightarrow{\{\alpha_\times\}} s_2$ or the tuple notation $(s_1, \{\alpha_\times\}, s_2)$. A transition $s \xrightarrow{\{\}} s'$ is called a ε-transition. We often use s_0 to stand for the initial state and s_f to stand for a final state. Note that by definition NFA and DFA may have ε-transitions. Sometimes we need to talk about automata that do not have ε-transitions which are called ε-free; we denote such an automaton $\not\varepsilon$-NFA ($\not\varepsilon$-DFA).

Definition 2.9 (acceptance) *A* run *of the automaton is a sequence of transitions starting in the initial state, i.e.* $s_0 \xrightarrow{\{\alpha_\times^1\}} s_1 \xrightarrow{\{\alpha_\times^2\}} s_2 \ldots \xrightarrow{\{\alpha_\times^n\}} s_n$*. A run is called* accepting *if it ends in a final state, i.e.* $s_n \in F$*. An automaton* accepts a string w *iff there exists an accepting run s.t.* $\{\alpha_\times^1\}\{\alpha_\times^2\} \ldots \{\alpha_\times^n\} = w$*. All the strings accepted by an automaton form the* language *accepted by the automaton; denoted* $\mathcal{L}(A)$ *or just* A*. Therefore* $w \in \mathcal{L}(A)$ *or* $w \in A$ *means that the string* w *is accepted by the automaton* A*.*

Proposition 2.4 ([HMU00])

1. *DFA and NFA recognize the same class of languages; i.e. for any NFA there is a determinisation procedure to generate a DFA which accepts the same language (and any DFA is also a NFA).*
2. *For any DFA one may use a Myhill-Nerode minimization procedure to obtain a minimal and unique DFA which accepts the same language as the initial automaton.*
3. *For any NFA we can construct an equivalent NFA which has only one final state and no transitions starting from the final state.*
4. *For any NFA we can construct (using the classical* ε*-closure construction) a* ε*-free NFA which will accept the same language.*
5. *For any NFA with unique final state and no transitions starting from the final state we can remove all the* ε*-transitions which do not end in the final state (using a variation of the* ε*-closure which does not consider the final state as part of the closures), and the resulting automaton will accept the same regular set.*

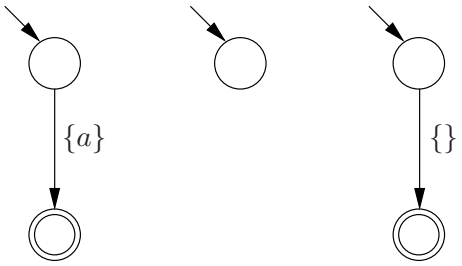

Figure 1: Automata corresponding to $a \in \mathcal{A}_B$, $\mathbf{0}$, and $\mathbf{1}$. Circles represent states, arrows represent transitions, pointed circles are the initial states, double circles are final states.

Because of the results of Proposition 2.4 we are free to work with NFA with one initial state, possibly one final state with no outgoing transitions, and which may have ε-transitions only ending in the final state; we denote these automata by CNFA.

Theorem 2.5 (actions to automata) *For any action $\alpha \in \mathcal{SKA}$ we can construct an automaton $CNFA(\alpha)$ which accepts the regular concurrent set $\hat{I}_{\mathcal{SKA}}(\alpha)$.*

Proof: The proof is adapted from [HMU00] for the regular expressions operators ($+$, $\cdot$, and *) and adds a new construction for the synchrony operator $\times$. We use induction on the structure of the actions.

Basis: For each action $a \in \mathcal{A}_B$, $\mathbf{0}$, and $\mathbf{1}$ we build the automata from Fig. 1. It is easy to see that: the automaton from Fig. 1(i) accepts the concurrent set $\{\{a\}\} = \hat{I}_{\mathcal{SKA}}(a)$; the automaton from Fig. 1(ii) accepts the empty set $\emptyset = \hat{I}_{\mathcal{SKA}}(\mathbf{0})$ (because there is no final state $F = \emptyset$); and the automaton from Fig. 1(iii) accepts the special concurrent set $\{\{\}\} = \hat{I}_{\mathcal{SKA}}(\mathbf{1})$. Moreover, all automata from Fig. 1 are of CNFA type. In notation the automaton of Fig. 1(i) is: CNFA(a)= $(\{s_1, s_2\}, \{a\}, s_1, \rho, \{s_2\})$ with $(s_1, s_2) \in \rho(a)$ as the only transition.

Inductive step: For actions of the form $\alpha + \beta$ the induction hypothesis states that we have the automata CNFA(α)=$(S_\alpha, \mathcal{P}(\mathcal{A}_B^\alpha), s_1, \rho_\alpha, s_2)$ which accepts $\hat{I}_{\mathcal{SKA}}(\alpha)$ and CNFA(β)=$(S_\beta, \mathcal{P}(\mathcal{A}_B^\beta), s_3, \rho_\beta, s_4)$ which accepts $\hat{I}_{\mathcal{SKA}}(\beta)$. We construct the automaton CNFA($\alpha + \beta$) from Fig. 2(i) as:

$$\text{CNFA}(\alpha + \beta)=(S_{\alpha\beta}, \mathcal{P}(\mathcal{A}_B^\alpha \cup \mathcal{A}_B^\beta), s_0, \rho_{\alpha\beta}, s_5)$$

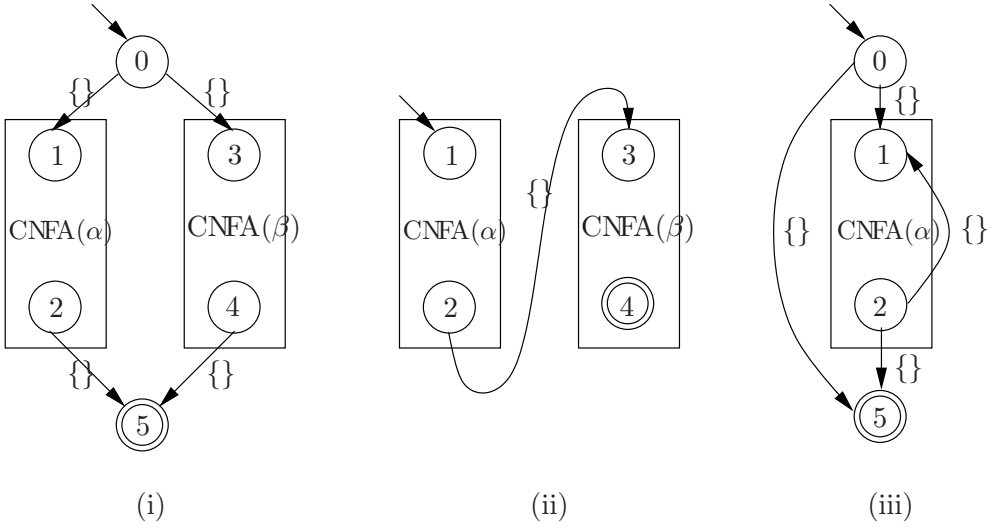

Figure 2: Automata corresponding to $\alpha + \beta$, $\alpha \cdot \beta$, and α^*.

with $S_{\alpha\beta} = S_\alpha \cup S_\beta \cup \{s_0, s_5\}$ where s_0, s_5 are new states not belonging neither to S_α nor to S_β. The new transition relation is:

$$\rho_{\alpha\beta} = \rho_\alpha \cup \rho_\beta \cup \{(s_0, \{\}, s_1), (s_0, \{\}, s_3), (s_2, \{\}, s_5), (s_4, \{\}, s_5)\}.$$

It is easy to see that the regular concurrent set accepted by CNFA($\alpha +$ β) is the union of the regular concurrent sets accepted by CNFA(α) and CNFA(β). But by induction this gives the set $\hat{I}_{\mathcal{SKA}}(\alpha) \cup \hat{I}_{\mathcal{SKA}}(\beta)$ which by the homomorphic definition of $\hat{I}_{\mathcal{SKA}}$ we have that CNFA($\alpha + \beta$) accepts $\hat{I}_{\mathcal{SKA}}(\alpha + \beta)$.

Similarly we construct for the actions of the form $\alpha \cdot \beta$ the automaton CNFA($\alpha \cdot \beta$) of Fig. 2(ii) from the two smaller automata CNFA(α) and CNFA(β). For the actions of the form α^* the automaton CNFA(α^*) is pictured in Fig. 2(iii). Note that the automata of Fig. 2 are not of CNFA type; therefore after each operation we need to apply Proposition 2.4(5) to remove the unwanted ε-transitions. For the three constructions we have in Fig. 2 it is easy to see how Proposition 2.4(5) works: e.g. in Fig. 2(i) states $0, 1, 3$ collapse into one, call it 013, and all transitions of the form $1 \xrightarrow{a} i$ or $3 \xrightarrow{a} i$ are replaced by a transition $013 \xrightarrow{a} i$, and the two ε-transitions are removed.

The new construction on automata is the one for actions of the form $\alpha \times \beta$ which is briefly pictured in Fig. 3. The automaton CNFA($\alpha \times \beta$) is constructed from the two smaller automata given before, CNFA(α) and CNFA(β), as follows:

$$\text{CNFA}(\alpha \times \beta) = (S_\alpha \times S_\beta, \mathcal{P}(\mathcal{A}_B^\alpha \cup \mathcal{A}_B^\beta), (s_1, s_3), \rho_{\alpha\beta}, (s_2, s_4)).$$

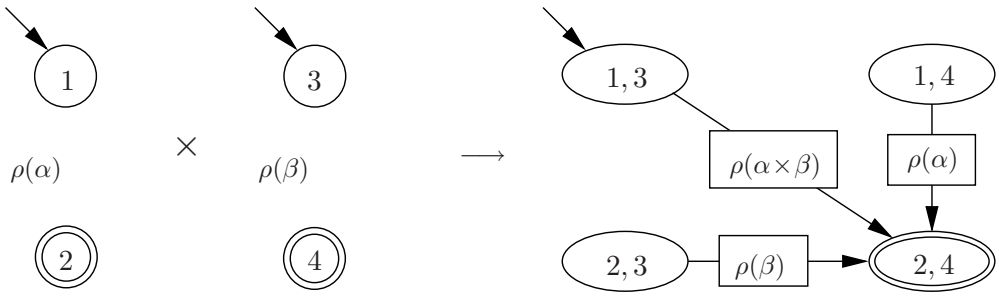

Figure 3: Automaton construction corresponding to $\alpha \times \beta$.

Note that states of CNFA($\alpha \times \beta$) are now equivalent of pairs of states of the old automata. Therefore, the initial state is the pair of the two initial states (s_1, s_3) and the final state is the one pair of the old final states (s_2, s_4). For brevity we write $s_1 s_3$ instead of (s_1, s_3). The new transition relation is:

$$(s_i^\alpha s_j^\beta, \gamma, s_k^\alpha s_l^\beta) \in \rho_{\alpha\beta} \text{ iff either:}$$

- $(s_i^\alpha, \gamma_1, s_k^\alpha) \in \rho_\alpha$ and $(s_j^\beta, \gamma_2, s_l^\beta) \in \rho_\beta$ and $\gamma_1 \cup \gamma_2 = \gamma$, or
- $(s_i^\alpha, \gamma, s_k^\alpha) \in \rho_\alpha$ and $s_j^\beta = s_l^\beta = s_4$, or
- $(s_j^\beta, \gamma, s_l^\beta) \in \rho_\beta$ and $s_i^\alpha = s_k^\alpha = s_2$.

The new automaton CNFA($\alpha \times \beta$) has size $|S_\alpha| * |S_\beta|$. We need to prove now that the automaton CNFA($\alpha \times \beta$) accepts exactly the concurrent strings of the regular concurrent set $\hat{I}_{\mathcal{SKA}}(\alpha \times \beta)$. We know that $\hat{I}_{\mathcal{SKA}}$ is a homomorphism, thus $\hat{I}_{\mathcal{SKA}}(\alpha \times \beta) = \hat{I}_{\mathcal{SKA}}(\alpha) \times \hat{I}_{\mathcal{SKA}}(\beta)$, and from the inductive hypothesis we know that CNFA(α) accepts exactly $\hat{I}_{\mathcal{SKA}}(\alpha)$ and that CNFA(β) accepts exactly $\hat{I}_{\mathcal{SKA}}(\beta)$. Therefore we need to prove the following double implication:

$$\exists w \in \text{CNFA}(\alpha \times \beta) \Leftrightarrow \exists u \in \text{CNFA}(\alpha) \text{ and } \exists v \in \text{CNFA}(\beta) \text{ s.t. } u \times v = w.$$

For the $\Leftarrow$ implication we have that: there exists an accepting run of CNFA(α) for u, i.e. $s_0^\alpha \xrightarrow{\{\alpha_\times^1\}} s_1^\alpha \xrightarrow{\{\alpha_\times^2\}} s_2^\alpha \ldots \xrightarrow{\{\alpha_\times^n\}} s_n^\alpha$ with $u = \{\alpha_\times^1\}\{\alpha_\times^2\}\ldots\{\alpha_\times^n\}$; there exists an accepting run of CNFA(β) for v, i.e. $s_0^\beta \xrightarrow{\{\beta_\times^1\}} s_1^\beta \xrightarrow{\{\beta_\times^2\}} s_2^\beta \ldots \xrightarrow{\{\beta_\times^m\}} s_m^\beta$ with $v = \{\beta_\times^1\}\{\beta_\times^2\}\ldots\{\beta_\times^m\}$; and $m > n$, and $w = (\{\alpha_\times^1\} \cup \{\beta_\times^1\})(\{\alpha_\times^2\} \cup \{\beta_\times^2\})\ldots(\{\alpha_\times^n\} \cup \{\beta_\times^n\})\{\beta_\times^{n+1}\}\ldots\{\beta_\times^m\}$. Considering the construction of CNFA($\alpha \times \beta$) we need to find an accepting run (i.e. starting in (s_0^α, s_0^β) and ending in (s_n^α, s_m^β)) for the string w. It is easy to see that there are the following transitions in $\rho_{\alpha\beta}$: $(s_{i-1}^\alpha, s_{i-1}^\beta) \xrightarrow{\{\alpha_\times^i\} \cup \{\beta_\times^i\}} (s_i^\alpha, s_i^\beta)$ for

$0 < i \leq n$ forming a run which starts in the initial state of CNFA($\alpha\times\beta$) and ends in (s_n^α, s_n^β) which accepts the first part of w. Because s_n^α is the final state of CNFA($\alpha\times\beta$) there are in the CNFA($\alpha\times\beta$) the following transitions $(s_n^\alpha, s_j^\beta) \xrightarrow{\{\beta_\times^{j+1}\}} (s_n^\alpha, s_{j+1}^\beta)$ with $n \leq j < m$ which form a run continuing the previous run and ending in the final state of CNFA($\alpha\times\beta$), and accepting the second part of w. Therefore, we have found an accepting run of CNFA($\alpha\times\beta$) over the whole string w.

For the $\Rightarrow$ implication we have that: there exists an accepting run of CNFA($\alpha \times \beta$) for w, i.e. $(s_0^\alpha, s_0^\beta) \xrightarrow{\{w_\times^1\}} \ldots \xrightarrow{\{w_\times^m\}} (s_n^\alpha, s_m^\beta)$ with $w = \{w_\times^1\} \ldots \{w_\times^m\}$. The choice of indexes does not matter. Consider the operation of constructing the CNFA($\alpha\times\beta$) from the smaller automata CNFA(α) and CNFA(β). We need to show that we can find two accepting runs for $u \in CNFA(\alpha)$ and $v \in CNFA(\beta)$. Consider the first transition to be $(s_0^\alpha, s_0^\beta) \xrightarrow{\{w_\times^1\}} (s_1^\alpha, s_1^\beta)$ with $s_0^\alpha \neq s_1^\alpha$ and $s_0^\beta \neq s_1^\beta$. This means that $\exists\{u_\times^1\}$ and $\exists\{v_\times^1\}$ s.t. $\{w_\times^1\} = \{u_\times^1\} \cup \{v_\times^1\}$ and the transitions $s_0^\alpha \xrightarrow{\{u_\times^1\}} s_1^\alpha \in CNFA(\alpha)$ and $s_0^\beta \xrightarrow{\{v_\times^1\}} s_1^\beta \in CNFA(\beta)$. We continue with the subsequent transitions for w and find subsequent transitions for u and v until we reach a case as: $(s_i^\alpha, s_i^\beta) \xrightarrow{\{w_\times^j\}} (s_j^\alpha, s_j^\beta)$ with $s_i^\alpha = s_j^\alpha$. This case appears only if s_i^α is the final state of CNFA(α); say s_n^α. In this case we have found the accepting run for u. We find the following transition for v in CNFA(β): $s_i^\beta \xrightarrow{\{w_\times^j\}} s_j^\beta$. Note that the same reasoning is carried analogous when $s_i^\beta = s_j^\beta$. Because s_n^α is final state then there is no outgoing transition from it, therefore the only way to continue with the initial run for w is to have transitions of the form $(s_n^\alpha, s_i^\beta) \xrightarrow{\{w_\times^j\}} (s_n^\alpha, s_j^\beta)$. Each of these transitions yields a transition for v in CNFA(β). This process stops in the state (s_n^α, s_m^β) and therefore the run for v stops in the final state s_m^β of CNFA(β). We have found the accepting run for v. From the reasoning we have that $w = u\times v$. □

Proposition 2.6 *1. The determinisation procedure of classical NFA is applicable to the automata on concurrent sets.*

2. The Myhill-Nerode minimization procedure is also applicable to the automata on concurrent sets.

Corollary 2.7 (unique minimal automaton) *For any automaton on concurrent sets there exists a unique minimal deterministic automaton accepting the same regular concurrent set.*

In [Koz94] the completeness of Kleene algebra is proven by appealing to the representation of finite state automata with matrices over arbitrary

Kleene algebras. The most important construction is the * operation over matrices which basically gives the regular expressions encoding the regular languages accepted when going from each state of the automaton to every other state; construction which comes from J.H. Conway [Con71]. In essence this construction is the algebraic equivalent of the combinatorial procedure of transforming a NFA into a regular expression [HMU00]. In the algebraic approach to automata the regular language accepted by the automaton is obtained as (the interpretation of) a single regular expression.

The proof of completeness that we give here follows similar ideas only that it uses a combinatorial argument. The motivation is that in the application of Section 2.4 (and possibly for extensions of PDL with synchrony) we use the automata associated to actions; and thus a combinatorial argument is more clarifying in this direction. The algebraic approach with matrices over synchronous Kleene algebras can be obtained from this. On another hand we make use of the procedure of eliminating states to generate a regular expression from a NFA [HMU00]. Adapting the method of eliminating states to our automata on concurrent strings is trivial. For this method we consider the NFA to have regular expressions labeling the transitions.

Definition 2.10 *Consider $\mathcal{SKA} \vdash \alpha = \beta$ to mean that the $\mathcal{SKA}$ equation on the right can be deduced from the axioms of $\mathcal{SKA}$ using the classical rules of equational reasoning (reflexivity, symmetry, transitivity, and substitution), instantiation, and introduction and elimination of implication. Consider also a relation $\equiv \subseteq T_{\mathcal{SKA}} \times T_{\mathcal{SKA}}$ defined as: $\alpha \equiv \beta \Leftrightarrow \mathcal{SKA} \vdash \alpha = \beta$.*

Remark: The proof that $\equiv$ is a congruence is classical based on the deduction rules and we leave it to the reader.

Lemma 2.8 *Consider the method of eliminating states as a function $\mathcal{E}$ which takes an automaton on concurrent strings and returns an action of $\mathcal{SKA}$; then $\forall \alpha \in \mathcal{SKA}$ we have $\alpha \equiv \mathcal{E}(CDFA(\alpha))$.*

Proof: Note first that $CDFA(\alpha)$ is the minimal deterministic automaton obtained from the automaton $CNFA(\alpha)$ corresponding to the action α and constructed as in Theorem 2.5. The proof of the lemma uses induction on the structure of the action.

Basis: take the actions $\mathbf{0}$, $\mathbf{1}$, and $a \in \mathcal{A}_B$ and thus consider the automata of Fig. 1. These are CDFA and the $\mathcal{E}$ procedure does nothing but to return the regular expressions $\mathbf{0}$, $\mathbf{1}$, and $a \in \mathcal{A}_B$ respectively.

Inductive case: for brevity, we consider only a simple case for $+$ and the case particular to $\mathcal{SKA}$ for $\times$. For $\alpha = \alpha_1 + \alpha_2$ the inductive hypothesis says that for α_1 the procedure $\mathcal{E}$ returns a unique automaton (pictured in Fig.4(i)), and similar for α_2. The construction for $+$ from Theorem 2.5 (i.e.

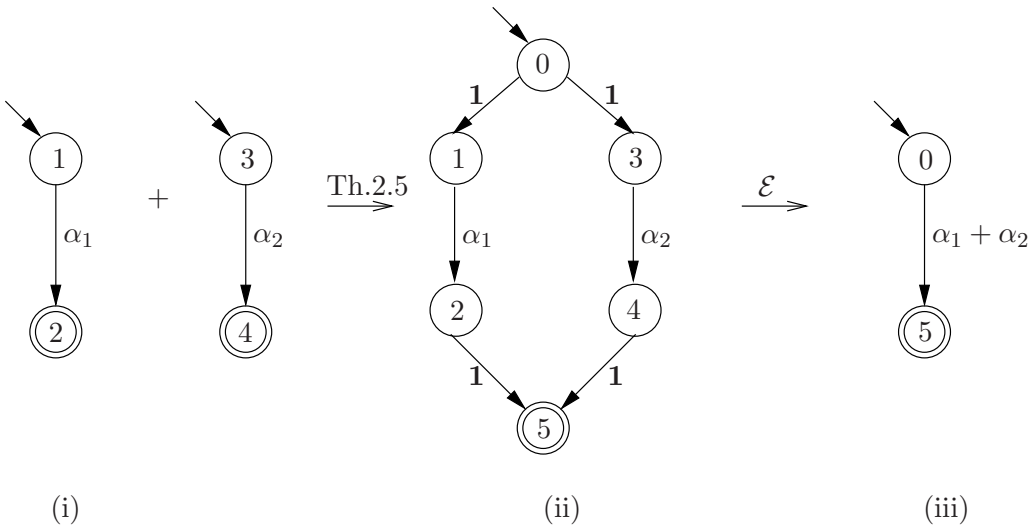

Figure 4: Automata on synchronous Kleene algebra.

Fig.2(i)) can now be applied to the two automata (even in this form with regular expressions on the transitions) to yield the automaton in Fig.4(ii). It is easy to see that the $\mathcal{E}$ procedure eliminates first states 1 and 2 to obtain the action $\mathbf{1}\cdot\alpha_1\cdot\mathbf{1} = \alpha_1$ and then states 3 and 4 to obtain $\mathbf{1}\cdot\alpha_2\cdot\mathbf{1} = \alpha_2$. Then a simple method contracts the two resulting transitions into one labeled with $\alpha_1 + \alpha_2$ (as in Fig.4(iii)).

For $\alpha = \alpha_1 \times \alpha_2$ the inductive hypothesis gives the same unique automata of Fig.4(i). To addapt the method of Theorem 2.5 we need to combine the transitions by using the action combinator $\times$ instead of the $\cup$ operation on sets.[2] By combining the two smaller automata of Fig.4(i) we obtain an automaton similar to the one in Fig.3 with four states ($(1,3)$, $(1,4)$, $(2,3)$, $(2,4)$) and three transitions. The two states $(1,4)$ and $(2,3)$ are just removed by $\mathcal{E}$ without any consequence and we remain with only one transition from the start state $(1,3)$ to the final state $(2,4)$ labeled by $\alpha_1 \times \alpha_2$ (i.e. our goal). □

Theorem 2.9 (completeness of axiomatization) *For any two actions α and β of $\mathcal{A}$ then $\alpha = \beta$ is a theorem of $\mathcal{SKA}$ iff the corresponding regular concurrent sets $\hat{I}_{\mathcal{SKA}}(\alpha)$ and $\hat{I}_{\mathcal{SKA}}(\beta)$ are the same.*

Proof : The proof of the forward implication follows from the fact that $\mathcal{ARCS}$ is a synchronous Kleene algebra (see Theorem 2.3). We use induction

[2]The reader should give more thought to this important change as to be confident that the automata obtained with the method of Theorem 2.5 and the one obtained with the change here accept the same regular concurrent set.

on the derivation and prove as base case that the implication holds for the axioms of $\mathcal{SKA}$. For the inductive case we consider the rules of equational reasoning.

The proof of the converse implication is based on Lemma 2.8. Take two arbitrary actions $\alpha, \beta \in \mathcal{SKA}$ s.t. $\hat{I}_{\mathcal{SKA}}(\alpha) = \hat{I}_{\mathcal{SKA}}(\beta)$ they denote the same regular concurrent set. Now take $CDFA(\alpha)$ and $CDFA(\beta)$ to be the minimal deterministic automata corresponding to the actions and accepting respectively $\hat{I}_{\mathcal{SKA}}(\alpha)$ and $\hat{I}_{\mathcal{SKA}}(\beta)$. (Build first CNFA automata as in Theorem 2.5 and then transform them into unique deterministic automata cf. Corollary 2.7.) Because $\hat{I}_{\mathcal{SKA}}(\alpha) = \hat{I}_{\mathcal{SKA}}(\beta)$ we have that $CDFA(\alpha)$ and $CDFA(\beta)$ denote the same automaton (up to isomorphism of states). Now we apply $\mathcal{E}$ to obtain an action γ which is both $\gamma \equiv \alpha$ and $\gamma \equiv \beta$ (cf. Lemma 2.8). Therefore we have the conclusion $\alpha \equiv \beta$ (i.e. $\mathcal{SKA} \vdash \alpha = \beta$). □

Theorem 2.10 (decidability) *The problem of deciding whether $\alpha = \beta$ in $\mathcal{SKA}$ is solved in quadratic time and is PSPACE-complete.*

Proof: The proof is a simple consequence of the completeness theorem. In order to test the equality of two actions we test the equality of the corresponding regular concurrent sets $\hat{I}_{\mathcal{SKA}}(\alpha)$ and $\hat{I}_{\mathcal{SKA}}(\beta)$. This is done with the help of the translation of the actions into automata on concurrent sets. Then we use the method of [SM73] to get the PSPACE-completeness and a table-filling method to get a quadratic running time [HMU00]. □

2.4 Application to deontic logic of actions

It was advocated by G.H. von Wright [VW68] that deontic logic would benefit from a "foundation of actions", i.e. a logic of actions, and many of the philosophical paradoxes would go away. Important contributions to this approach were done by K. Segerberg for introducing structured actions inside the deontic modalities [Seg82]. More recent the deontic logic community is adopting propositional dynamic logic (PDL) in various forms as a basis for a deontic logic of actions; as found in the seminal work of J.-J.Ch. Meyer [Mey88] or in the recent approach of [BWM01] based on modal μ-calculus. For a history of the developments of the (deontic) logic of actions see the survey [Seg92].

We do not make an investigation of the deontic logic formalism here but only of the algebra that stands at the basis of the semantics of the deontic modalities defined over the synchronous actions (see [Pri08] for the deontic logic presentation). The *synchronous actions* that appear inside the deontic modalities are constructed only with the $+$, $\cdot$, and $\times$ constructors of $\mathcal{KAT}$. We do not consider the Kleene * operator over the actions inside the deontic

modalities. Therefore these actions form a particular algebraic structure which we call $\mathcal{CA}$.

We assume the reader familiar with the logical terminology used in this motivating paragraph; otherwise it is of no importance for the technical algebraic development below. None of the few papers that consider iteration (i.e. Kleene *) as an action operator under deontic modalities [vdM96, BWM01] give a precise motivation for having such recurring actions inside obligations (denoted with O), permissions (P), or prohibitions (F). Experience provides us examples which are counter intuitive: for the simple expression $O(a^*)$ – "One is obliged to not pay, or pay once, or pay twice in a row, or..." – which puts no actual obligations; for $P(a^*)$ – "One has the right to do any sequence of action a." – is a very shallow permission which is captured by the widespread *Closure Principle* in jurisprudence where *what is not forbidden is permitted* [Seg82]. Moreover, as pointed out in [BWM01] expressions like $F(a^*)$ or $P(a^*)$ should be simulated with the PDL modalities by respectively $\langle a^*\rangle F(a)$ and $[a^*]P(a)$. In our opinion repetition in legal contracts can be captured by using temporal or dynamic logic modalities combined with deontic modalities over *-free actions.

Definition 2.11 (the $\mathcal{CA}$ algebra) *$\mathcal{CA}$ is a σ-algebra with signature $\sigma = \{+, \cdot, \times, \mathbf{0}, \mathbf{1}, \mathcal{A}_B\}$ which respects axioms (1)-(17) of Table 1. The term algebra $T_{\mathcal{CA}}(\mathcal{A}_B)$ is free in the corresponding class of algebras $\mathcal{CA}(\mathcal{A}_B)$ over the generators of $\mathcal{A}_B \cup \{\mathbf{0}, \mathbf{1}\}$. In this context the elements of $\mathcal{A}^{\mathcal{D}}$ (i.e. the deontic actions) are the terms constructed with the grammar below:*

$$\alpha \quad ::= \quad a \mid \mathbf{0} \mid \mathbf{1} \mid \alpha + \alpha \mid \alpha \cdot \alpha \mid \alpha \times \alpha$$

where $a \in \mathcal{A}_B$.

Lemma 2.11 *The interpretation $\hat{I}_{\mathcal{SKA}}$ of Definition 2.7 is a homomorphism from $T_{\mathcal{CA}}$ to *-free $\mathcal{ARCS}$ (i.e. the sets of concurrent words without the operation * of Definition 2.6).*

Definition 2.12 (rooted tree) *A* rooted tree *is an acyclic connected graph $(\mathcal{N}, \mathcal{E})$ with a designated node r called* root *node. $\mathcal{N}$ is the set of* nodes *and $\mathcal{E}$ is the set of* edges *(where an edge is an ordered pair of nodes (n, m)). We consider* rooted trees with labeled edges *and denote the labeled directed edges with (n, α, m) and the tree with $(\mathcal{N}, \mathcal{E}, \mathcal{A}_B)$. The labels $\alpha \in 2^{\mathcal{A}_B}$ are* sets of basic labels*; e.g. $\alpha_1 = \{a, b\}$ or $\alpha_2 = \{a\}$ with $a, b \in \mathcal{A}_B$. Labels are compared for set equality (or set inclusion). Note the special* empty set $\{\}$ *label. We restrict our presentation to* finite *rooted trees (i.e., there is no infinite path in the graph starting from the root node). Moreover, we consider only* deterministic *trees (i.e. there are no two edges from the same node labeled with the same label). The set of all such defined trees is denoted $\mathcal{T}$.*

Notation: When the label of an edge is not important (i.e. it can be any label) we may use the notation (n, m) instead of (n, α, m) $\forall \alpha \in 2^{\mathcal{A}_B}$. All nodes $\{m \,|\, (n, m)\}$ are called the *child* nodes of n. A path which cannot be extended with a new transition is called *final*. The final nodes on each final path are called *leaf nodes*; denote by $leafs(T) = \{n \mid n \text{ is a leaf node}\}$.

Theorem 2.12 *Automata generated conform Theorem 2.5 from the three basic automata of Fig. 1 using the constructions corresponding to the operations of $\mathcal{CA}$ (i.e. operations pictured in Fig.2(i), 2(ii), and Fig.3) have the shape of trees.*

Proof: It is easy to see that the only construction that may add loops is the one for * which we do not consider here. Moreover, the removal of ε-transitions conform Proposition 2.4(5) does not introduce loops. Therefore the automata that we need to consider for $\mathcal{CA}$ are without loops. But also they do not have the shape of a tree because of the restriction of uniqueness of the final state. This restriction was necessary to make the construction from Fig. 3 for $\times$ work in conjunction with the * construction. This is not the case any more and we can also drop this requirement, thus getting *nondeterministic* tree shapes for the constructed automata. The nondeterminism comes from the fact that we may have from one state two outgoing edges labelled with the same action. These are easily collapsed into one by collapsing the end nodes and renaming the following transitions accordingly (without changing the language accepted). This simple procedure works because there are no loops (i.e. there are no transitions which come back into the nodes that we collapse). □

We can work with the actions of $\mathcal{CA}$ or with the trees of Theorem 2.12 interchangeably. This is the basic idea used in giving semantics to the deontic modalities applied over actions of $\mathcal{CA}$. The interpretations as trees are used to search the state space of the model. The logic formalism is out of the scope of this paper. In the sequel we concentrate on more properties of the actions which are useful in the deontic setting of legal contracts.

Definition 2.13 (canonical form for $\mathcal{CA}$) *We say that an action α of $\mathcal{CA}$ (i.e. defined with the operators $+$, $\cdot$, $\times$,) is in* canonical form *denoted by $\underline{\alpha}$ iff it has the following form:*

$$\underline{\alpha} = \underset{i \in I}{+} \alpha_{\times}^{i} \cdot \underline{\alpha^{i}}$$

where $\alpha_{\times}^{i} \in \mathcal{A}_B^{\times}$ and $\underline{\alpha^{i}} \in \mathcal{A}^{\mathcal{D}}$ is an action of $\mathcal{CA}$ in canonical form. The indexing set I is finite as the compound actions α are finite; i.e. there is a finite number of application of the $+$ operator. Actions $\mathbf{0}$ and $\mathbf{1}$ are considered in canonical form.

Theorem 2.13 *For every action α of the algebra $\mathcal{CA}$ we have a corresponding $\underline{\alpha}$ in canonical form and equivalent to α.*

Proof: We use structural induction on the structure of the actions of $\mathcal{A}^{\mathcal{D}}$ given by the constructors of the algebra. In the inductive proof we take one case for each action construct. The proof also makes use of the axioms (1)-(17). For convenience in the presentation of the proof we define for an action in canonical form $\underline{\alpha}$ the set $R = \{\alpha^i_\times \mid i \in I\}$ to contain all the concurrent actions on the first "level" of $\underline{\alpha}$. Based on, we often use in the proof the alternative notation for the canonic form $\underline{\alpha} = +_{\alpha_\times \in R}\, \alpha_\times \cdot \underline{\alpha'}$ which emphasizes the exact set of the concurrent actions on the first level of the action $\underline{\alpha}$. Often the R in the notation is omitted for brevity.

Basis:

a) When α is a basic action a of $\mathcal{A}_B$ it is immediately proven to be in canonical form just by looking at the definition of the canonical form. Action a is in canonical form with the set R containing only one element, namely a and the $\cdot$ constructor is applied to a and to skip action $\mathbf{1}$ ($a \cdot \mathbf{1} = a$). Note that the choice ($+$) of only one action ($+_a\, a$) should be understood as choice among fail action $\mathbf{0}$ and a ($+_a\, a \stackrel{def}{=} \mathbf{0} + a = a$).

b) The special actions $\mathbf{1}$ and $\mathbf{0}$ are considered by definition to be in canonical form.

In the inductive step we consider only one step of the application of the constructors; the general compound actions should follow from the associativity of the constructors.

Inductive steps:

a) Consider $\alpha = \beta + \beta'$ is a compound action obtained by applying once the $+$ constructor. By the induction hypothesis β and β' are in canonical form. It means that $\beta = +_{b_i}\, b_i \cdot \beta_i$ and $\beta' = +_{b'_j}\, b'_j \cdot \beta'_j$. Because of the associativity and commutativity of $+$, $\beta + \beta'$ is also in canonical form:

$$\beta + \beta' = \underset{b_i \in B}{+}\, b_i \cdot \beta_i \;+\; \underset{b'_j \in B'}{+}\, b'_j \cdot \beta'_j = \underset{a \in B \cup B'}{+}\, a \cdot \beta_a$$

where a and β_a are related in the sense that if $a = b_i$ then $\beta_a = \beta_i$ and if $a = b'_j$ then $\beta_a = \beta'_j$. Because the inductive hypothesis states that all β_i and β'_j are in canonical form it follows that also β_a (which is just a change of notation) are in canonical form.

b) Consider $\alpha = \beta \cdot \beta'$ with $\beta = +_{b_i}\, b_i \cdot \beta_i$ and $\beta' = +_{b'_j}\, b'_j \cdot \beta'_j$ in canonical form. We now make use of the distributivity of $\cdot$ over $+$, and of the

associativity of $\cdot$ and $+$ constructors. α transforms in several steps into a canonical form. In the first step α is:

$$\alpha = \beta \cdot \beta' = (\underset{b_i \in B}{+} b_i \cdot \beta_i) \cdot (\underset{b'_j \in B'}{+} b'_j \cdot \beta'_j)$$

and if we consider $|B| = m$ then α becomes:

$$\alpha = b_1 \cdot \beta_1 \cdot (\underset{b'_j \in B'}{+} b'_j \cdot \beta'_j) + \ldots + b_m \cdot \beta_m \cdot (\underset{b'_j \in B'}{+} b'_j \cdot \beta'_j)$$

Subsequently α distributes the $\cdot$ over all the members of the choice actions. In the end α becomes a choice of sequences; when we consider $|B| = m$ and $|B'| = k$.

$$\alpha = b_1 \cdot \beta_1 \cdot b'_1 \cdot \beta'_1 + \ldots + b_m \cdot \beta_m \cdot b'_k \cdot \beta'_k$$

This is clearly a canonical form because all actions $\beta_i \cdot b'_j \cdot \beta'_j$ are in canonical form due to the inductive hypothesis.

c) Consider $\alpha = \beta \times \beta'$ with $\beta = +_{b_i} b_i \cdot \beta_i$ and $\beta' = +_{b'_j} b'_j \cdot \beta'_j$ in canonical form. The proof of this case is fairly lengthy and we show here only a simple particular case.

Let us consider actions $\beta = b \cdot \beta'$, $\gamma = c \cdot \gamma'$, and $\delta = d \cdot \delta'$ in canonical form. They are the components of $\alpha = (\beta + \gamma) \times \delta$. We apply the distributivity of $\times$ with respect to $+$ and get:

$$\alpha = \beta \times \delta + \gamma \times \delta = (b \cdot \beta') \times (d \cdot \delta') + (c \cdot \gamma') \times (d \cdot \delta')$$

By applying equation (17) we get:

$$\alpha = b \times d \cdot \beta' \times \delta' + c \times d \cdot \gamma' \times \delta'$$

This shows that α is in canonical form because by the inductive supposition $\beta' \times \delta'$ and $\gamma' \times \delta'$ are in canonical form.

□

For the actions inside the deontic modalities one important notion is that of *action negation*. There have been a few works related to negation of actions [Mey88, HKT00, LW04, Bro03]. In [Mey88], the same as in [HKT00] action negation is with respect to the universal relation which, for example for PDL gives undecidability. Decidability of PDL with negation of only atomic actions has been achieved in [LW04]. A so called "relativized action complement" is defined in [Bro03] which is basically the complement of an action (not with respect to the universal relation but) with respect to a set

formed of atomic actions closed under the application of the action operators. This kind of negation still gives undecidability when several action operators are involved.

In our view *action negation* says that the negation $\overline{\alpha}$ of action α is the action given by all the immediate actions that *take us outside* the tree of α [BWM01].[3] With $\underline{\alpha}$ it is easy to formally define $\overline{\alpha}$. Note that our definition is not with respect to the universal relation and it is defined for all actions (not only the basic actions).

Definition 2.14 (action negation) *The* action negation *is a derived operator denoted by $\overline{\alpha}$ and is defined as a function $\overline{\ }: \mathcal{A}^{\mathcal{D}} \to \mathcal{A}^{\mathcal{D}}$ (i.e. action negation is not a principal combinator for the actions) which works on the equivalent canonical form $\underline{\alpha}$ as:*

$$\overline{\alpha} = \overline{\underset{i\in I}{+}\ \alpha^i_\times \cdot \underline{\alpha^i}} = \underset{\beta_\times \in \overline{R}}{+} \beta_\times \ + \ \underset{j\in J}{+}\ \gamma^j_\times \cdot \overline{\underset{i\in I'}{+}\ \underline{\alpha^i}}$$

Consider $R = \{\alpha^i_\times \mid i \in I\}$. The set $\overline{R}$ contains all the concurrent actions $\beta_\times$ with the property that $\beta_\times$ is not more demanding than any of the actions $\alpha^i_\times$:

$$\overline{R} = \{\beta_\times \mid \beta_\times \in \mathcal{A}^\times_B \text{ and } \forall i \in I, \alpha^i_\times \not\leq_\times \beta_\times\},$$

and $\gamma^j_\times \in \mathcal{A}^\times_B$ and $\exists \alpha^i_\times \in R$ s.t. $\alpha^i_\times \leq_\times \gamma^j_\times$. The indexing set $I' \subseteq I$ is defined for each $j \in J$ as:

$$I' = \{i \in I \mid \alpha^i_\times \leq_\times \gamma^j_\times\}.$$

Negation of $\mathbf{1}$ *is* $\mathbf{0} = \overline{\mathbf{1}}$ *and negation of* $\mathbf{0}$ *is* $\mathbf{1} = \overline{\mathbf{0}}$.

Remark: The negation operation formalizes the fact that an action is not performed. In an active system this boils down to performing the action given by $\overline{\alpha}$. In other words *not performing* action α means either not performing any of its immediate actions $\alpha^i_\times$, or by performing one of the immediate actions and then not performing the remaining action. Note that to perform an action $\alpha^i_\times$ means to perform any action that includes $\alpha^i_\times$ (this is encoded by the demanding relation $<_\times$). Therefore in the negation we may have actions $\gamma^j_\times$ which include more immediate actions, e.g. $\alpha = a \cdot b + c \cdot d$ and may perform $\gamma^j_\times = a \times c$. At this point we need to look at both actions b and d in order to derive the negation, e.g. performing now d means that α was done, whereas performing c means that α was not done (and $a \times c \cdot c$ must be part of negation).

Proposition 2.14 (action negation is an action)

1. *The negation operator returns an action.*
2. *The negation of an action α is in canonical form.*

[3] Intuitively, action negation encodes the violation of an obligation in legal contracts.

Proof: For 1 we have to prove that there is no infinite branching and also no infinite depth in the action returned by the negation operation $\overline{\ }$ (as actions of $\mathcal{CA}$ are finite terms). This is fairly simple to see.

The first possible source of infinite branching is $\overline{R}$. This is ruled out because $\overline{R}$ is a finite set as it contains elements of $\mathcal{A}_B^{\times}$ which is finite. The indexing set J is finite (having maximum size of $|\mathcal{A}_B^{\times}|$) thus ruling out this second possible source of infinite branching. Lastly, because $\gamma_{\times}^{j}$ is an action from $\mathcal{A}_B^{\times}$ it is finite, and because the demanding relation $<_{\times}$ is a partial order over $\mathcal{A}_B^{\times}$ (cf. Proposition 2.1) then there is only a finite number of $\alpha_{\times}^{i}$ less demmanding then $\gamma_{\times}^{j}$, which makes the indexing set I' also finite. In any way, because the actions α are finite then they have finite branching, thus the indexing set I is finite. Therefore, I' is finite as clearly $|I'| \leq |I|$.

It remains to argue that $\overline{\alpha}$ has finite depth. The first part of the action negation (i.e. $+_{\beta_{\times} \in \overline{R}} \beta_{\times}$) introduces only branches of finite depth. Actually this is the part which ends each branch of the action negation. Therefore the only source of infinite depth may be the second part (i.e. $+_{j \in J}\ \gamma_{\times}^{j} \cdot \overline{+_{i \in I'} \underline{\alpha^i}}$). Note that the definition of action negation is a recursive one and also note that it goes stepwise; meaning at each recursion step it decreases the length of the action it applies to. Naturally at some point (because the action α has finite depth) α^i will become $\mathbf{1}$. At this point the recursive application of $\overline{\ }$ stops in a $\mathbf{0}$ action.

For 2 the action negation respects the canonical form because it is a choice between sequences of a action (i.e. γ or $\alpha_{\times}^{i}$) which is only a basic action or a concurrent action; and each of these are followed by an action which is also in canonical form (i.e. respectively $\mathbf{1}$ and $\overline{\alpha^i}$ which is in canonical form by structural induction). □

Definition 2.15 (representation independence) *Consider a set S equipped with an equivalence relation $\equiv$. We call a function $f : S \to S$* independent of the representation *iff $\forall a, b \in S$ then if $a \equiv b$ implies $f(a) \equiv f(b)$.*

If a function is independent of the representation then the result of the application of the function on any of the elements of an equivalence class is *equivalent* with the result of the application on the representant element of the equivalence class. The next result is a negative one as it shows that the negation operator $\overline{\ }$ is *not* independent of the representation. This negative result is because we do not have a notion of unique normal form for the actions that the negation is applied to.

Proposition 2.15 *The function $\overline{\ } : \mathcal{A} \to \mathcal{A}$ is not independent of the representation of the actions of $\mathcal{A}$.*

Proof: The proof is immediate by considering a counterexample. Take the actions a and $a + b \cdot \mathbf{0}$ where $\mathcal{A}_B = \{a, b, c\}$ contains three basic actions.

Obviously $a = a + b \cdot \mathbf{0}$ by axioms (7) and (3). Then by Definition 2.14 we have that $\overline{a} = c + b + b \times c + a \cdot \mathbf{0} \overset{(7),(3)}{=} c + b + b \times c$ and that $\overline{a + b \cdot \mathbf{0}} = c + a \cdot \mathbf{0} + b \cdot \mathbf{1} \overset{(7),(3),(6)}{=} c + b$. Note that the first $=$ sign is to show what returns the $\overline{\ }$ function, where the second $=$ sign represents the equality over actions from the $\mathcal{CAT}$ algebra. Obviously $c + b + b \times c \neq c + b$ which ends our proof. □

3 Synchronous Kleene Algebra with Boolean Tests

We follow the work of D. Kozen [Koz97b] on defining Kleene algebras with tests.

Definition 3.1 *Synchronous Kleene algebra with tests is given by* $\mathcal{SKAT} = (\mathcal{A}, \mathcal{A}^?, +, \cdot, \times, ^*, \neg, \mathbf{0}, \mathbf{1})$ *which is an order sorted algebraic structure with* $\mathcal{A}^? \subseteq \mathcal{A}$ *that combines the previously defined* $\mathcal{SKA}$ *with a Boolean algebra in a special way we see in this section. The structures* $(\mathcal{A}^?, +, \cdot, \neg, \mathbf{0}, \mathbf{1})$ *and* $(\mathcal{A}^?, +, \times, \neg, \mathbf{0}, \mathbf{1})$ *are Boolean algebras and the Boolean negation operator* $\neg$ *is defined only on Boolean elements of* $\mathcal{A}^?$.

Notation: The elements of the set $\mathcal{A}^?$ are called *tests* (or *guards*) and are included in the set of actions of the $\mathcal{SKA}$ algebra (i.e. tests are special actions; $\mathcal{A}^? \subseteq \mathcal{A}$). As in the case of actions, the Boolean algebra is generated by a finite set $\mathcal{A}^?_B$ of *basic tests*. We denote tests by $\phi, \varphi, \ldots$ and basic tests by $p, q, \ldots$. Note the overloading of the functional symbols $+, \cdot, \times$, and functional constants $\mathbf{0}, \mathbf{1}$: over arbitrary actions they have the meaning as in the previous section; whereas, over tests they take the meaning of the classical disjunction (for $+$, which sometimes we denote by $\vee$), conjunction (for $\cdot$ and $\times$ which sometimes we denote by $\wedge$), falsity and truth (for $\mathbf{0}, \mathbf{1}$ which we often denote by $\bot$ and $\top$). In this richer context the elements of $\mathcal{A}$ (i.e. the actions and tests) are the syntactic terms constructed with the grammar below:

$$\begin{array}{rcll} \alpha & ::= & a \mid \phi \mid \alpha + \alpha \mid \alpha \cdot \alpha \mid \alpha \times \alpha \mid \alpha^* & \textit{actions} \\ \phi & ::= & p \mid \mathbf{0} \mid \mathbf{1} \mid \phi + \phi \mid \phi \cdot \phi \mid \phi \times \phi \mid \neg\phi & \textit{tests} \end{array}$$

The intuition behind tests is that for an action $\phi \cdot \alpha$ it is the case that action α can be performed only if the test succeeds (the guard ϕ is satisfied). We do not go into details about the properties of a Boolean algebra as these are classical results. For a thorough understanding see [Koz97b] and references therein.

Remarks: Note that the preference relation $\leq_+$ is defined over tests also and $\mathbf{1}$ is the most preferable test; i.e. $\forall \phi \in \mathcal{A}^?, \phi \leq_+ \mathbf{1}$. It is natural to think

of **1** as $\top$ because testing a tautology always succeeds; i.e. $\top \cdot \alpha = \mathbf{1} \cdot \alpha = \alpha$, which says that the action α can always be performed after a $\top$ test. The dual is $\bot = \mathbf{0}$ meaning that testing a falsity never succeeds, and thus, any following action α is never performed; i.e. $\mathbf{0} \cdot \alpha = \mathbf{0} = \bot \cdot \alpha = \bot$ (the action sequence stops when it reaches the $\bot$ test).

Definition 3.2 (extra axiom) *We give the equivalent of axiom (17) for tests:*

$$(17') \quad (\phi \cdot \alpha) \times (\varphi \cdot \beta) = (\phi \times \varphi) \cdot (\alpha \times \beta) \quad \forall \phi, \varphi \in \mathcal{A}^?.$$

Remark: Note that $\top \in \mathcal{A}^?$ and therefore this axiom allowes sequences of actions with **1**, which was not the case in axiom (17). On the other hand **1** is dealt with only in conjunction with another test, and not with another action. Particular instances of this axiom are $\alpha \times \phi = \phi \cdot \alpha$ and $\phi \times \alpha = \phi \cdot \alpha$. Note that $(\phi \cdot \alpha) \times (\varphi \cdot \beta)$ is $(\phi \wedge \varphi) \cdot \alpha \times \beta$.

3.1 Interpretation over regular sets of guarded concurrent strings

Guarded strings have been introduced in [Kap69] and have been used to give interpretation to the Kleene algebra with tests [Koz97b]. Here we need an extension to *guarded concurrent strings* similar to the extension we gave in Section 2.2 from strings to concurrent strings. We intentionally overload several symbols as they have the same intuitive meaning but the particular definitions (adapted to guarded concurrent strings) are different.

Definition 3.3 (guarded concurrent strings) *Over the set of basic tests $\mathcal{A}_B^?$ we define* atoms *as functions $\nu : \mathcal{A}_B^? \to \{0,1\}$ assigning a Boolean value to each basic test. Consider the finite alphabet $\mathcal{P}(\mathcal{A}_B)$ of all the subsets of basic actions (denoted by x, y). A* guarded concurrent string *(denoted by u, v, w) is a sequence*

$$w = \nu_0 x_1 \nu_1 \ldots x_n \nu_n, \qquad n \geq 0,$$

where ν_i are atoms and $x_i \in \mathcal{P}(\mathcal{A}_B)$ are sets of basic actions. We define $first(w) = \nu_0$ and $last(w) = \nu_n$.

Notation: Denote by $Atoms = \{0,1\}^{\mathcal{A}_B^?}$ the set of all atoms ν. There are $|Atoms| = 2^{|\mathcal{A}_B^?|}$ atoms in total. We say that an atom *satisfies* a test ϕ (denote it by $\nu \Rightarrow \phi$) iff the truth assignment of the atom ν to the basic tests makes ϕ true. Note that for basic tests $\nu \Rightarrow p$ iff $\nu(p) = 1$. Note that a guarded concurrent string is starting with an atom and ends in an atom. We define two mappings over guarded concurrent strings: τ which returns the associated (unguarded) concurrent string; i.e. $\tau(w) = x_1 \ldots x_n$ and π

which returns the sequence of guards; i.e. $\pi(w) = \nu_0\nu_1 \dots \nu_n$. Consider $Pref(\pi(w))$ to be the set of all prefixes of $\pi(w)$. Recall that when the concurrent action $\alpha_\times$ is known or important then we use the notation $\{\alpha_\times\} \in \mathcal{P}(\mathcal{A}_B)$ instead of x_i.

Definition 3.4 (sets of guarded concurrent strings) *Consider sets of guarded concurrent strings, denoted A, B, C. We define the following operations:*

$$\begin{aligned} \mathbf{0} &\triangleq \emptyset \\ \mathbf{1} &\triangleq Atoms \\ A + B &\triangleq A \cup B \\ A \cdot B &\triangleq \{uv \mid u \in A, v \in B\} \\ A \times B &\triangleq \{u \times v \mid u \in A, v \in B\} \\ A^* &\triangleq \bigcup_{n \geq 0} A^n \\ \neg A &\triangleq Atoms \setminus A, \quad \forall A \subseteq Atoms \end{aligned}$$

where $u = \nu_0^u x_1 \nu_1^u \dots x_n \nu_n^u$ and $v = \nu_0^v y_1 \nu_1^v \dots y_m \nu_m^v$ are guarded concurrent strings. The operation $\cup$ is just union over sets. The fusion product *uv of two guarded concurrent strings is defined iff $last(u) = first(v)$ and is $uv = \nu_0^u x_1 \dots x_n \nu_0^v y_1 \nu_1^v \dots y_m \nu_m^v$. The concurrency operation on guarded concurrent strings $u \times v$ is defined iff $\pi(u) \in Pref(\pi(v))$ when $m \geq n$ (whereas when $n \geq m$ we interchange u and v) and is:*

$$u \times v \triangleq \nu_0^v (x_1 \cup y_1) \nu_1^v (x_2 \cup y_2) \nu_2^v \dots (x_n \cup y_n) \nu_n^v y_{n+1} \nu_{n+1}^v \dots y_m \nu_m^v$$

Remark: Note that when $n, m = 0$, i.e. the guarded concurrent strings are just atoms then the two operations $\cdot$ and $\times$ over sets become the equivalent of $\cap$.

Theorem 3.1 *Any set of guarded concurrent sets containing $\emptyset$, Atoms, and the sets corresponding to $\mathcal{A}_B$ and $\mathcal{A}_B^?$ (i.e. those sets returned by the $I_{\mathcal{SKAT}}$ of Definition 3.5 below), and closed under the operations $+, \cdot, \times, ^*, \neg$ of Definition 3.4 is a synchronous Kleene algebra with tests. Call these the class of* algebras of guarded concurrent sets.

Proof : Simply by routine check of the axioms of $\mathcal{SKA}$ of Table 1 together with the extra axiom for tests (17′). Moreover the axioms for the two Boolean algebras of Definition 3.1 are also respected. □

Remark: The *smallest algebra of guarded concurrent sets* is called the *algebra of regular guarded concurrent sets* and is denoted $\mathcal{ARGCS}$. We call it "regular" because it can be constructed using the standard interpretation $\hat{I}_{\mathcal{SKAT}}$ defined below.

Definition 3.5 (interpretation) *We define the interpretation of the actions as the homomorphism $\hat{I}_{\mathcal{SKAT}}$ from $\mathcal{SKAT}$ into the sets of guarded concurrent strings. $\hat{I}_{\mathcal{SKAT}}$ is the homomorphic extension of the map $I_{\mathcal{SKAT}} : \mathcal{A}_B \cup \mathcal{A}_B^? \cup \{\mathbf{0}, \mathbf{1}\} \to \mathcal{ARGCS}$ which maps the generators of $T_{\mathcal{SKAT}}$ into regular sets of guarded concurrent strings as follows:*

$$
\begin{aligned}
I_{\mathcal{SKAT}}(a) &= \{\nu\{a\}\nu' \mid \nu, \nu' \in Atoms\},\ \forall a \in \mathcal{A}_B \\
I_{\mathcal{SKAT}}(p) &= \{\nu \in Atoms \mid \nu(p) = 1\},\ \forall p \in \mathcal{A}_B^? \\
I_{\mathcal{SKAT}}(\mathbf{0}) &= \emptyset \\
I_{\mathcal{SKAT}}(\mathbf{1}) &= Atoms
\end{aligned}
$$

The homomorphic extension is classical:

$$
\begin{aligned}
\hat{I}_{\mathcal{SKAT}}(\alpha) &= I_{\mathcal{SKAT}}(\alpha),\ \forall \alpha \in \mathcal{A}_B \cup \mathcal{A}_B^? \cup \{\mathbf{0}, \mathbf{1}\} \\
\hat{I}_{\mathcal{SKAT}}(\alpha + \beta) &= \hat{I}_{\mathcal{SKAT}}(\alpha) + \hat{I}_{\mathcal{SKAT}}(\beta) \\
\hat{I}_{\mathcal{SKAT}}(\alpha \cdot \beta) &= \hat{I}_{\mathcal{SKAT}}(\alpha) \cdot \hat{I}_{\mathcal{SKAT}}(\beta) \\
\hat{I}_{\mathcal{SKAT}}(\alpha \times \beta) &= \hat{I}_{\mathcal{SKAT}}(\alpha) \times \hat{I}_{\mathcal{SKAT}}(\beta) \\
\hat{I}_{\mathcal{SKAT}}(\alpha^*) &= \hat{I}_{\mathcal{SKAT}}(\alpha)^* \\
\hat{I}_{\mathcal{SKAT}}(\neg\phi) &= \neg\hat{I}_{\mathcal{SKAT}}(\phi)
\end{aligned}
$$

3.2 Automata on guarded concurrent strings

Automata on guarded strings have been introduced in [Koz03a] as an extension of classical finite state automata with transitions labeled with any test of $\mathcal{A}^?$. These automata accept regular languages of guarded strings. We give here an alternative presentation of automata on guarded strings to suite better our extension to recognize guarded concurrent strings.

Proposition 3.2 (automata for guards) *For the set of Atoms there exists a class of finite state automata which accept all and only the subsets of atoms.*

Proof: This is a folk result. Recall that the set *Atoms* is the finite set of functions $\nu : \mathcal{A}_B^? \to \{0, 1\}$ from the finite domain of basic tests $\mathcal{A}_B^?$ to the truth set $\{0, 1\}$. We define a class of finite automata which recognize functions $f : A \to B$ with finite domain and codomain, and thus, the languages accepted are subsets of the finite set of all possible functions f and nothing else. As a particular case when $A = \mathcal{A}_B^?$ and $B = \{0, 1\}$ this class of finite automata will accept languages of atoms.

Take a finite state automaton $M = (S, \Sigma, S_0, \rho, F)$ with the following restrictions:

1. $S_0 \subseteq S$, $F \subseteq S$ the set of initial and final states;
2. $\Sigma = A \cup B$ the two sorted (disjoint) alphabet;

3. $\rho = \rho_A \cup \rho_B$ the transition relation as the union of the two disjoint relations $\rho_A \subseteq S \times A \times S$ and $\rho_B \subseteq S \times B \times S$;
4. Any final trace (i.e. trace which starts in a start state and ends in a final state) of the automaton is of length $2 * |A|$ and no element of A appears twice on a final trace (i.e. no two transitions in the final trace are labeled with the same $a \in A$);
5. $\forall (s_1, b, s_2) \in \rho_B$ then $s_1 \notin S$ and $\nexists (s_2, b', s_3) \in \rho_B$;
6. $\forall (s_1, a, s_2) \in \rho_A$ then $s_2 \notin F$ and $\nexists (s_2, a', s_3) \in \rho_A$.

Denote the set of all such automata by $\mathcal{M}$.

We need to show that the automata with the restrictions above cannot accept something that is not an encoding of a total function. The condition 2 ensures that any string accepted is from $(A + B)^*$. Condition 5 ensures that no string starts with an element of B and that there cannot be two consecutive elements from B in any string. Condition 6 ensures that no string ends with an element from A and that there cannot be two consecutive elements from A in any string. Therefore, all strings accepted start with an element of A and end with an element from B and they are strict alternations of elements from A and B. Condition 4 ensures that the strings accepted have length $2 * |A|$ therefore, it has n elements from A each followed by one element from B. Moreover, condition 4 ensures that no element of A appears twice in the accepted strings and therefore, all elements of A appear. Therefore this is the representation of a total function from A to B. (The representation of a function from A to B is a set of n pairs $(a, b) \in A \times B$ where $n = |A|$, there are no two pairs with the same first element, and for each element from A there is a pair with this being the first element.)

We show that for any function $f : A \to B$ we can build an automaton which accepts it; i.e. its encoding as a string from $(AB)^n$. The automaton starts with a transition labelled from A and follows with a transition labelled from B and continues like this for n times ending in F; each time using a different element from A. It is clear that the automaton respects the conditions above as no transition labelled from B starts in an initial state, and no transition labelled from A ends in F. The trace constructed has length $2 * n$. Moreover, the restriction that no label from A is repeated on the trace is respected. If we want to accept a set of functions than we simply enumerate in this manner all functions (or equivalently make the union of the automata for each individual function).

There may be several different automata accepting the same function. Moreover, one automaton may accept two different strings encoding the same function. This is because we do not have an order on the elements of A. If we require an arbitrary total order on A which must be respected by each of the final traces of the automata, then the uniqueness of the representation is guaranteed. □

Corollary 3.3 *Automata as defined in Proposition 3.2 but on the particular alphabet* $\Sigma = \mathcal{A}_B^? \cup \{0,1\}$ *accept all and only the sets of atoms.*

Corollary 3.4 *Automata of Proposition 3.2 are closed under the classical operations on finite automata union (denoted* $\cup$*) and intersection (denoted* $\cap$*); and correspond respectively to union of sets of functions and intersection of sets of functions. Automata of Proposition 3.2 are not closed under concatenation.*

Definition 3.6 (automata on guarded concurrent strings) *Consider a two level finite automaton* $N = (S, \mathcal{P}(\mathcal{A}_B), S_0, \rho, F, \lceil\cdot\rceil)$*. It consists at the first level of a finite automaton on concurrent strings as in Definition 2.8; i.e.* $(S, \mathcal{P}(\mathcal{A}_B), S_0, \rho, F)$*, together with a map* $\lceil\cdot\rceil : S \to \mathcal{M}$*. The mapping associates with each state of the first level an automaton* $M \in \mathcal{M}$ *as defined in Proposition 3.2 which accepts atoms. The automata in the states make the second (lower) level. Denote the language of atoms accepted by* $\lceil s\rceil$ *with* $\mathcal{L}(\lceil s\rceil)$*.*

Definition 3.7 (acceptance) *Take the definitions and notations for automata on concurrent strings from Section 2.3. We say that* a guarded concurrent string w is accepted by an automaton N *iff there exists an accepting run of the first level automaton which accepts* $\tau(w)$ *and for each state* s_i *of the run there exists an accepting run of the automaton* $\lceil s_i\rceil$ *which accepts the corresponding atom* ν_i *of* w*.*

Remark: It is easy to see that the automata on guarded concurrent strings can be considered as ordinary finite state automata. The two level definition that we give is useful in defining the *fusion product* and *concurrent composition* operations over these automata. Henceforth we denote automata on guarded concurrent strings (as defined in Definition 3.6) by GNFA.

Definition 3.8 (fusion product) *Consider two (disjoint) automata over guarded concurrent strings given by* $N_1 = (S_1, \mathcal{P}(\mathcal{A}_B), S_0^1, \rho_1, F_1, \lceil\cdot\rceil_1)$ *and* $N_2 = (S_2, \mathcal{P}(\mathcal{A}_B), S_0^2, \rho_2, F_2, \lceil\cdot\rceil_2)$*. Define the fusion product automaton*

$$N_{12} = (S, \mathcal{P}(\mathcal{A}_B), S_0, \rho, F, \lceil\cdot\rceil)$$

where:

- $S = (S_1 \setminus F_1) \cup (S_2 \setminus S_0^2) \cup S'$*;*
- $S' = F_1 \times S_0^2$ *and for* $s \in S'$ *denote* $s|_{F_1} \in F_1$ *the first component, and* $s|_{S_0^2}$ *the second component;*
- $S_0 = S_0^1$*;*

- $F = F_2$;
- $\rho = (\rho_1 \setminus \{(s_1, a, s_2) \in \rho_1 \mid s_2 \in F_1\}) \cup (\rho_2 \setminus \{(s_1, a, s_2) \in \rho_2 \mid s_1 \in S_0^2\})$
  $\cup \{(s_1, a, s) \mid s \in S' \text{ and } (s_1, a, s|_{F_1}) \in \rho_1\}$
  $\cup \{(s, a, s_1) \mid s \in S' \text{ and } (s|_{S_0^2}, a, s_1) \in \rho_2\}$;
- $\forall s \in S'$, $\lceil s \rceil = \lceil s|_{F_1} \rceil \cap \lceil s|_{S_0^2} \rceil$.

The first two conditions ensure that we combine the final states of the first automaton with all the initial states of the second automaton, so to get all possible concatenations. The next two conditions set the initial states to be the initial states of N_1 and the final states to be the final states of N_2. The condition on the transition relation keeps all the transitions from both automata and modifies accordingly those transitions that have to do with the old final states of N_1 and the old initial states of N_2. The last condition makes sure that we concatenate only guarded concurrent strings that have the same atoms *last* and $first$ (i.e. we keep in the new nodes of S only those atoms that correspond to both the old final nodes of N_1 and to the old initial nodes of N_2).

Definition 3.9 (concurrent composition) *For two (disjoint) automata over guarded concurrent strings $N_1 = (S_1, \mathcal{P}(\mathcal{A}_B), S_0^1, \rho_1, F_1, \lceil\cdot\rceil_1)$ and $N_2 = (S_2, \mathcal{P}(\mathcal{A}_B), S_0^2, \rho_2, F_2, \lceil\cdot\rceil_2)$ take the concurrent composition operation as defined in Theorem 2.5 for finite automata over concurrent strings. This construction is applied here to the top level automata. For the lower level automata of the nodes do their intersection: $\forall (s_1, s_2) \in S_1 \times S_2$, $\lceil (s_1, s_2) \rceil = \lceil s_1 \rceil \cap \lceil s_2 \rceil$.*

Proposition 3.5 *Automata on guarded concurrent strings are closed under union, fusion product, and concurrent composition.*

Proof: The union operation for automata on guarded concurrent strings is the same as given in Theorem 2.5 for automata on concurrent strings where any new first level nodes that are added have as automata on second level accepting anything (i.e. accepting the set *Atoms*). For the fusion product it is clear that the top level automaton remains an automaton on concurrent strings and in each nodes the intersection of the automata for guards gives also an automaton for guards (because of closure under intersection; see Corollary 3.4). The same observation applies to the concurrent composition. □

Theorem 3.6 (translation into automata) *For any action $\alpha \in \mathcal{SKAT}$ we can construct an automaton $GNFA(\alpha)$ which accepts the regular set $\hat{I}_{\mathcal{SKAT}}(\alpha)$.*

Proof: We follow a similar recursive construction of the automaton based on the structure of the actions as we did in Theorem 2.5. We give constructions for the basic actions (here the recursion stops) and for each action constructor of $\mathcal{SKAT}$. In each case we have to show that the automaton accepts the same set of guarded concurrent strings as the interpretation of the actions. To show this we use an inductive argument.

From Proposition 3.2 we know how to construct an automaton for guards to recognize a given set of atoms. More precisely we know how to construct an automaton for guards which recognizes the set of atoms which make only the basic test p true; or make only *some* or *none* of the basic tests true.

Basis: For a basic test p the automaton consists of only one top level state s which is both the initial and the final state. The second level automaton $\lceil s \rceil$ is such constructed to accept all and only the atoms which make the basic test p true. It is clear that this automaton accepts the set of guarded concurrent strings $\{\nu \in Atoms \mid \nu(p) = 1\}$ which corresponds to $\hat{I}_{\mathcal{SKAT}}(p)$.

For the special actions $\mathbf{0}$ and $\mathbf{1}$ the construction is similar to that for tests. For $\mathbf{0}$ the automaton $\lceil s \rceil$ accepts the empty set thus the whole automaton accepts $\hat{I}_{\mathcal{SKAT}}(\mathbf{0}) = \emptyset$. For $\mathbf{1}$ the automaton $\lceil s \rceil$ accepts all possible strings (i.e. an universal automaton) encoding all possible atoms; thus the automaton $GNFA(\mathbf{1})$ accepts $Atoms = \hat{I}_{\mathcal{SKAT}}(\mathbf{1})$.

For a basic action $a \in \mathcal{A}_B$ we construct an automaton which at the top level is as the automaton in Fig.2(i) and at the second level the automata $\lceil s_1 \rceil$ and $\lceil s_2 \rceil$ both accept $Atoms$; thus $GNFA(a)$ accepts $\{\nu\{a\}\nu' \mid \nu, \nu' \in Atoms\} = \hat{I}_{\mathcal{SKAT}}(a)$.

Induction: Corresponding to the action constructors $\cdot$ and $\times$ we have respectively the constructions of fusion product and concurrent composition on automata given in Definition 3.8 and Definition 3.9.

Consider $\alpha = \alpha_1 \cdot \alpha_2$. By the inductive hypothesis we have $\mathcal{L}(GNFA(\alpha_1)) = \hat{I}_{\mathcal{SKAT}}(\alpha_1)$ and $\mathcal{L}(GNFA(\alpha_2)) = \hat{I}_{\mathcal{SKAT}}(\alpha_2)$. From Definition 3.5 we know $\hat{I}_{\mathcal{SKAT}}(\alpha) = \hat{I}_{\mathcal{SKAT}}(\alpha_1) \cdot \hat{I}_{\mathcal{SKAT}}(\alpha_2)$ where $\cdot$ is as in Definition 3.4. The construction for fusion product of Definition 3.8 generates $GNFA(\alpha)$ s.t. it accepts $w = w_1 w_2$ where $w_1 \in GNFA(\alpha_1)$ and $w_2 \in GNFA(\alpha_2)$ (i.e. $w_1 \in \hat{I}_{\mathcal{SKAT}}(\alpha_1)$ and $w_2 \in \hat{I}_{\mathcal{SKAT}}(\alpha_2)$). Moreover, $last(w_1) = first(w_2)$ because of the last constraint of Definition 3.8. Therefore, w is contained in $\hat{I}_{\mathcal{SKAT}}(\alpha_1) \cdot \hat{I}_{\mathcal{SKAT}}(\alpha_2)$ cf. Definition 3.4.

It remains to prove the opposite inclusion; i.e. that for any two $w_1 \in \hat{I}_{\mathcal{SKAT}}(\alpha_1)$ and $w_2 \in \hat{I}_{\mathcal{SKAT}}(\alpha_2)$ we have that if $w_1 w_2 \in \hat{I}_{\mathcal{SKAT}}(\alpha_1 \cdot \alpha_2)$ then $w_1 w_2 \in \mathcal{L}(GNFA(\alpha_1 \cdot \alpha_2))$. From the same inductive hypothesis we know that $w_1 \in GNFA(\alpha_1)$ and $w_2 \in GNFA(\alpha_2)$. Because $w_1 w_2 \in \hat{I}_{\mathcal{SKAT}}(\alpha_1 \cdot \alpha_2)$ then we know (cf. Definition 3.4) that $last(w_1) = first(w_2)$. According to Definition 3.8 the last condition is satisfied for the w_1 and w_2 and thus the word $w_1 w_2$ is accepted by the fusion product of $GNFA(\alpha_1)$ and $GNFA(\alpha_2)$;

thus the conclusion.

Consider $\alpha = \alpha_1 \times \alpha_2$. We consider only the inclusion $\mathcal{L}(GNFA(\alpha)) \subset \hat{I}_{\mathcal{SKAT}}(\alpha)$; the opposite inclusion is simple and follows a similar reasoning as in the case before. By the inductive hypothesis we have $\mathcal{L}(GNFA(\alpha_1)) = \hat{I}_{\mathcal{SKAT}}(\alpha_1)$ and $\mathcal{L}(GNFA(\alpha_2)) = \hat{I}_{\mathcal{SKAT}}(\alpha_2)$. The $\hat{I}_{\mathcal{SKAT}}(\alpha_1 \times \alpha_2)$ is constructed as in Definition 3.4 and $GNFA(\alpha_1 \times \alpha_2)$ is the concurrent composition as in Definition 3.9 of the two smaller automata. From Theorem 2.5 we know that the word accepted by $GNFA(\alpha_1 \times \alpha_2)$ comes from the synchronous composition of two words accepted by the smaller automata $GNFA(\alpha_1)$ and $GNFA(\alpha_2)$. For our case of *guarded* concurrent strings the synchronous composition is restricted only to strings which have the same guards. In the automata GNFA this is achieved by the intersection of all the lower level automata for each new nodes.

For the action constructors $+$ and * we have the classical constructions from Fig.2(i) respectively Fig.2(iii) defined in Theorem 2.5. We need to take care that now we do not have at our disposal the empty transitions (for working with ε-transitions we need to have a convention similar to $\varepsilon a = a$). □

3.3 Completeness and Decidability

Theorem 3.7 (Completeness) *For any two actions α and β of $\mathcal{A}$ then $\alpha = \beta$ is a theorem of $\mathcal{SKAT}$ iff the corresponding regular sets of guarded concurrent strings $\hat{I}_{\mathcal{SKAT}}(\alpha)$ and $\hat{I}_{\mathcal{SKAT}}(\beta)$ are the same.*

Proof : The forward implication (or soundness) comes as a consequence of Theorem 3.1; i.e. from the fact that $\mathcal{ARGCS}$ is a synchronous Kleene algebra with tests. We use induction on the structure of the actions from $\mathcal{SKAT}$. □

3.4 $\mathcal{SKAT}$, Hoare logic, and Concurrency

Propositional Hoare Logic (PHL) is the version of Hoare logic which does not involve the assignment axiom explicitly [Koz00]. PHL reasons about programs at a more abstract level where the assignment axiom instances are just particular cases of atomic programs. Kleene algebra with tests (KAT) subsumes PHL and has the same complexity (PSPACE-complete for the Horn theory with premises of the form $\alpha = \mathbf{0}$). Moreover, KAT is complete for relational valid Horn formulas (i.e. all the rules of PHL are theorems of KAT), whereas PHL is incomplete. The following valid inference cannot be derived:

$$\frac{\{\psi\}\ \mathbf{if}\ \phi\ \mathbf{then}\ \alpha\ \mathbf{else}\ \alpha\ \{\psi\}}{\{\psi\}\ \alpha\ \{\psi\}}$$

Extensions of Hoare logic exist which treat procedure calls, goto jumps, pointers, or aliasing. Similar extensions can be devised for Kleene algebra with tests, e.g. some form of higher-order functions [AHK06b], non-local flow of control [Koz08], or local variables [AHK08]. We do not know about nested procedure calls and returns which is known to be a context free property (and not a regular property as is the style of KAT).

About expressiveness, KAT can encode the while programs (and more) which correspond to the notion of *tail recursion* (or *iteration*) from programming languages. It is known that tail recursion is strictly less expressive than (full) recursion. Recursion can be encoded with while programs and a stack. This corresponds to the context-free languages as opposed to the regular languages where KAT lies. The stack can be expressed in the First-Order Dynamic Logic (which is undecidable in general). KAT relates to the Propositional Dynamic Logic.

All these expressiveness issues hold for our $\mathcal{SKAT}$ too. Encoding partial correctness assertions (PCAs) into $\mathcal{SKAT}$ is done as in [Koz00,HKT00]. The PCA $\{\phi\}\alpha\{\psi\}$ intuitively says that if the program α starts in a state where ϕ holds (i.e. the precondition is true) then whenever the program terminates[4] the postcondition ψ will hold. There are two equivalent ways of encoding PCAs in $\mathcal{SKAT}$: $\phi\alpha\neg\psi = \mathbf{0}$ (it is not possible that program α starts with precondition ϕ and terminates with postcondition $\neg\psi$) or $\phi\alpha = \phi\alpha\psi$ (testing the postcondition ψ after termination of α started with precondition ϕ is superfluous).

Our notion of synchrony is based on the assumption of *interference freedom* of Owicki and Gries [OG76]. The definition of interference freedon given in [OG76] says that an action b does not interfere with another action α iff first, it does not change the postcondition of α and second, if interleaved at any point in α it does not change the precondition of the remaining actions (to be executed) of α. Note that the b actions in [OG76] are only the statements that could change the state of the system (i.e. the `await` and `assignment`). In our case these correspond to the basic actions.

Interference freedom in $\mathcal{SKAT}$ is given by the $\sim_{\mathcal{C}}$ relation (defined in terms of $\#_{\mathcal{C}}$) on the basic actions of $\mathcal{A}_B$. The difference is that in our case $\#_{\mathcal{C}}$ is given by an "oracle" whereas in [OG76] it is given by rules based on the syntax (i.e. the assignment axiom schemata; which in practice reduces to the instances of the axiom for each particular assignment). In our case we assume an oracle because we abstract away and the basic actions have no assumed structure. If particularized to assignments than the oracle giving the $\#_{\mathcal{C}}$ relation becomes the axiom schemata.

Example 3.1 *Consider two programs: a conditional* **if** ϕ **then** α **else** β *and a sequence* $\gamma \cdot \gamma'$. *Writing these into* $\mathcal{SKAT}$ *is, respectively* $\phi\alpha + \neg\phi\beta$

[4] To talk about termination (total correctness) we need to use an extended version of the Hoare logic (not in this paper).

*and $\gamma\gamma'$. Now put these two programs to run in parallel (synchronously) and therefore write $(\phi\alpha + \neg\phi\beta)\times(\gamma\gamma')$. The following sequence of equalities follow from the axioms of $\mathcal{SKAT}$: $(\phi\alpha)\times(\gamma\gamma') + (\neg\phi\beta)\times(\gamma\gamma') = (\phi\alpha)\times(\mathbf{1}\gamma\gamma') + (\neg\phi\beta)\times(\mathbf{1}\gamma\gamma')$ which by axiom (1′) and rules of Boolean algebra becomes $\phi(\alpha\times\gamma)\gamma' + \neg\phi(\beta\times\gamma)\gamma' = (\phi(\alpha\times\gamma) + \neg\phi(\beta\times\gamma))\gamma'$. If we write now back into the while language we get: (**if** ϕ **then** $\alpha\times\gamma$ **else** $\beta\times\gamma$); γ'.*

Note that this holds under the interference freedom assumption about the programs α and γ and β and γ which in $\mathcal{SKAT}$ is given by the compatibility relation $\alpha \sim_{\mathcal{C}} \gamma$ and $\beta \sim_{\mathcal{C}} \gamma$.

4 Final Considerations

4.1 Synchronous Kleene Algebra and True Concurrency Models

4.1.1 Mazurkiewicz trace theory

Definition 4.1 (Mazurkiewicz traces) *Consider a symmetric and irreflexive binary relation $I_{\mathcal{A}_B}$ called the independence relation (i.e. not causal) on a set of basic actions, say $\mathcal{A}_B$. Define $\equiv_{\mathcal{A}_B}$ as the least congruence in the monoid of strings over $\mathcal{A}_B$, i.e. $(\mathcal{A}_B^*, \cdot, \mathbf{1})$ s.t. if $(a,b) \in I_{\mathcal{A}_B}$ then $ab \equiv_{\mathcal{A}_B} ba$. For arbitrary strings we say that $u \equiv_{\mathcal{A}_B} v$ iff $\exists w_1 \ldots w_n$ with $u = w_1$ and $v = w_n$ and $\forall i, \exists w', w'', \exists a, b$ s.t. $(a,b) \in I_{\mathcal{A}_B}$ and $w_i = w'abw''$ and $w_{i+1} = w'baw''$. One equivalence class generated by $\equiv_{\mathcal{A}_B}$ is called a (Mazurkiewicz)* trace *and is denoted by $[w]_{\equiv_{\mathcal{A}_B}}$ (the equivalence class is generated by the class representant w).*

If $u \equiv_{\mathcal{A}_B} v$ then u is a permutation of v. A trace represents a run of a concurrent system. On the other hand a trance encodes several possible sequential runs which are considered equivalent due to the independence of some of the basic actions involved. From this point of view Mazurkiewicz traces talk about a special form of interleaving. The independence relation makes two basic actions *globally* independent; i.e. the basic actions are independent of each other no matter of their position on the sequential runs.

Take the same view (with set of equivalent interleavings) in $\mathcal{SKA}$: a concurrent action $\alpha_\times \in \mathcal{A}_B^\times$ is interpreted as the set of its basic actions that compose it, e.g. $\{a,b,c\}$. It encodes all the possible interleavings of these basic actions, e.g. $\{abc, acb, bac, bca, cab, cba\}$. Therefore, in the context of $\mathcal{SKA}$ the following definitions apply to the independence relation of Mazurkiewicz traces.

Definition 4.2 *Consider $I_{\mathcal{A}_B}$ to respect the following: $\forall a, b \in \mathcal{A}_B$ if $a \times b$ then $(a,b) \in I_{\mathcal{A}_B}$. Extend this to concurrent actions $\alpha_\times$ to say that if $\alpha_\times \times \beta_\times$ then $\forall a \in \{\alpha_\times\}, b \in \{\beta_\times\}$, $(a,b) \in I_{\mathcal{A}_B}$.*

Proposition 4.1 *For a concurrent action* $\alpha_\times = a_1 \times \ldots \times a_n$ *then* $I_{\mathcal{A}_B}$ *restricted to the basic actions of* $\alpha_\times$ *is a total relation; i.e.* $\forall\ 0 < i, j \leq n$ *then* $(a_i, a_j) \in I_{\mathcal{A}_B}$.

This proposition shows a first difference between the concurrency modelled in $\mathcal{SKA}$ and the concurrency of Mazurkiewicz traces. In the later the independence relation is not necessarily total (i.e. it may be defined as partial); this fact allowes for some basic action to move back and forth along the sequence of actions depending on which actions it is independent of. Therefore $\mathcal{SKA}$ cannot capture the concurrent behavior of Mazurkiewicz traces.

For general actions of $\mathcal{SKA}$ generated using also the $\cdot$ operator the definitions above are not sufficient any more. Consider the simple example: $(a \times b) \cdot a$ in $\mathcal{SKA}$ has the following intended sequential runs: $\{aba, baa\}$; whereas in Mazurkiewicz traces, because $(a, b) \in I_{\mathcal{A}_B}$ we get the following sequential runs: $[aba]_{\equiv_{\mathcal{A}_B}} = \{aba, baa, aab\}$. This shows that Mazurkiewicz traces cannot capture the concurrent behavior intended in $\mathcal{SKA}$ because we need the independence relation to be *local* to each sequential step of a concurrent run.

Remark: Mazurkiewicz traces have defined a *global* and *partial* independence relation. If we take the similar view in $\mathcal{SKA}$ we need a *local* and *total* independence relation. The locality comes from the perfect synchrony model we adopted, where all the concurrent actions are executed at each tick of a universal clock. The totality comes from our view of concurrent basic actions as forming a set.

4.1.2 Shuffle

Shuffle is an operation over regular languages (basically over words) which preserves regularity. Shuffle has been used to model concurrency in [Abr79, Gis84] with a position between the interleaving approach and the partial orders approach. Shuffle is a generalization of interleaving similar to what we discussed for the Mazurkiewicz traces but it does not take into considerration any other relation on the actions(events) which it interleaves. If we were to ignore the branching information in our actions then we can view $\times$ as some kind of *ordered shuffle*. The shuffling of two sequences of actions in $\mathcal{SKA}$ walks step by step (on the $\cdot$ operation) and shuffles the basic actions found (locally).

4.1.3 Pomsets

Pomsets have been long advocated by Pratt [Pra86] and many of the initial theoretical results were published as [Gis84]. The theory of pomsets is among

the first in concurrency theory to make a distinction between events (E) and actions (A). A pomset is a partially ordered set of events labeled (non-injective) by actions. Pomsets extend the idea of strings which are linearly ordered multisets to partially ordered multisets. Normally a multiset is $\mathbb{N}^A$ and assigns to each action of A a multiplicity from $\mathbb{N}$. In pomset theory they are more: E^A which assigns to each action of A a set of events from E, and more, events are ordered by the temporal partial order. Thus, an action may be executed several times and each execution of an action is an event. Formally a *pomset* is the isomorphism class (w.r.t. the events) of the structure $(E, A, <, \mu)$ where $\mu : E \to A$ is the labeling function of the events by action names.

Two events which are incomparable by $<$ are permitted to occur concurrently. An important feature of the pomset theory is that it is independent of the *granularity of the atomicity*; i.e. events may be either atomic or may have a even more elaborated structure (in [Gis84] operations over pomsets are defined by replacing events (with the same action name) by new pomsets). Moreover, the view of time does not matter as events may occupy time points or time intervals with no difference to the theory. There is also a large number of operations defined over pomsets (see [Pra86]), more than in the other theories we have seen.

A pomset describes only one execution of the concurrent system. A set of pomsets is called a *process* and describes the whole set of concurrent behaviors or a system (or process). Pomsets are more expressive then our synchronous actions. We know that a synchronous action represents a set of concurrent strings (as we called them). Each concurrent string is a particular pomset; formally it is a pomset where the partial order respects the constraint:

$$\text{all } \textit{maximal independent sets} \text{ are disjoint,} \quad \begin{array}{r} \textit{uniquely labeled}, \text{ and} \\ \textit{completely ordered}, \end{array} \tag{23}$$

where an *independent set of events* is $X \subseteq E$ s.t. $\forall e_i, e_j \in X$ then $(e_i \not< e_j) \wedge (e_j \not< e_i)$. An independent set is *uniquely labeled* iff the labeling function is injective on X; i.e. $\mu|_X$ is injective. Two independent sets X_i, X_j are *completely ordered* iff if $\exists e_i \in X_i$, $\exists e_j \in X_j$ with $e_i < e_j$ then $\forall e_i \in X_i$, $\forall e_j \in X_j$ is the case $e_i < e_j$. Call this a *synchronous pomset*.

Theorem 4.2 *Concurrent strings are completely characterized by synchronous pomsets.*

Proof: We need to prove two implications: for any w a concurrent string as in Definition 2.6 there is a synchronous pomset simulating it; and for any synchronous pomset there is a concurrent string.

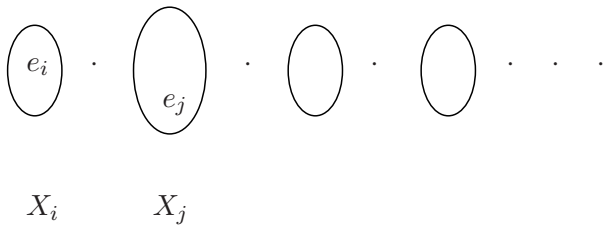

Figure 5: A concurrent string.

Consider a concurrent string as pictured in Fig.5. It is formed of sets of incomparable events named by unique actions (because of the axiom (14)). Each of these sets are disjoint and all the events in one set that follows after a $\cdot$ operator are in the relation $<$ with all the events that precede them (also because of the transitivity of the partial order). Therefore these sets of events are also maximal and are *independent sets* as in the definition above. The requirement of *completely ordered* is clearly satisfied.

For a synchronous pomset the fact that we consider the maximal independent sets to be disjoint gives much of the proof that we can consider it as a concurrent string. The requirement of completely ordered is needed to assure that there are no X-like pair of events in two independent sets. $\Box$

The good thing about the synchronous actions of $\mathcal{SKA}$ is that all the actions can be obtained (constructed) from a finite set of basic actions using a finite set of operations on actions. Is it the same situation for the pomsets?

Because of the $+$ operation the actions of $\mathcal{SKA}$ define *sets* of behaviors (of concurrent strings). Therefore we need to talk not of pomsets but of sets of pomsets (i.e. processes). The same $+$ operation exists for sets of pomsets (i.e. their union) which on synchronous pomsets behaves exactly like the $+$ in $\mathcal{SKA}$. For the $\cdot$ there is the ; on pomsets. The extension of ; to sets of pomsets is exactly the same as the extension of $\cdot$ to sets of concurrent strings. Similarly we find the Kleene * for pomsets.

For the $\times$ of $\mathcal{SKA}$ we did not find a straight forward equivalent for pomsets. Moreover, we are not sure if there is a *pomset definable* operation (as in terminology of [Gis84]). The first candidat was the concurrence operation $||$ but this breaks the *completely ordered* requirement. The *orthocurrence* operation on pomsets is also not good. We could not find a satisfactory new definition for $\times$ over pomsets because in order to enforce the conditions of synchronous pomsets we needed to look through the whole (infinite) structure of the partial order on events (a recursive destroying algorithm) starting with the smallest elements in the order.

4.1.4 Event structures

Event structures were introduced in [NPW79] as a model of concurrency based on events partially ordered by a *causal dependency* relation and with additional structure given by a *conflict* relation and an *enabling* relation. We base our presentation on [Win88]. Configurations (or computation states) are viewed as subsets of events (left-closed w.r.t. the causal dependency order) which for one event all the events it depends on are included. Note that parallel processes computations are modeled by causal independence between events. Event structures are based on the fundamental *axiom of finite causes* which basically states that any event depends on a finite number of events. We like to note that our interpretation of actions as trees mimic event structures with the $\cdot$ as a causal dependency relation and any node respects the axiom of finite causes.

The conflict relation of event structures is similar to our conflict relation $\#_{\mathcal{C}}$ and to the one found in Esterel. The conflict relation is defined over events and its intuitive interpretation is to express how the occurrence of an event rules out the occurrence of another. More general event structures $(E, \#, \vdash)$ are obtained by relaxing the partial order to a *enabling* relation $\vdash$ with the intuition that now it is sufficient for an event to be enabled by a single chain of enabled events starting with an event e_0 which is enabled by *no* event.

4.2 Related Work

4.2.1 More on SCCS

The well known CCS calculus [Mil95] is proven to be just a subcalculus of the asynchronous version of SCCS (named ASCCS). Milner defines asynchrony in terms of synchrony by using a special action for delay (denote it $\mathbf{d}$ in our context) which takes exactly one time step to execute. The ASCCS is comprised only of *asynchronous* processes. An asynchronous process is a synchronous process which can *delay indefinitely after each action* (but not before the first action); in our notation $a \cdot \mathbf{d}^* \cdot b \cdot \mathbf{d}^* \ldots$.

ASCCS is strictly more expressive than CCS as synchronous occurrence of actions are enforced by the $\times$ operation in conjunction with the action prefix (i.e. sequence) operation. In CCS any two actions which are not inverse must be interleaved. To achieve this the synchrony operation $\times$ is changed into the composition operation $||$ which is defined in terms of $\times$. The idea is to make it possible that for two processes composed in parallel $P||Q$ either one does an action and the other *may* delay or the other does an action and the first may delay. The fact that a process *may* delay includes that fact that the process may also *not* delay. Therefore, $||$ does include the case when the two processed execute inverse actions.

Remark: As far as we know, there is no result on completeness of ASCCS, which means that it is not known whether any asynchronous agent can be expressed in ASCCS. On the other hand the work of [dS85] establishes the relative completeness of SCCS and Meije languages.

4.2.2 The strong synchrony hypothesis of the Esterel family

Esterel [BC85, BG92] is a synchronous imperative programming language where the model of concurrency = synchrony + determinism + multi clocks. This is opposed to the classical view of concurrency = nondeterminism + universal clock. The strong synchrony hypothesis says that the control structures (like sequence) and the simple operations (like assignments) take no time (instantaneously); making the execution machine "infinitely fast". This means that the external environment of the system remains invariant during an execution of an action.

Esterel is used to model reactive real-time systems. An Esterel program only reacts on input events from the environment (producing some other output events). Communication is through instantaneous broadcasting of events, therefore all processes have the same view of the environment.

Determinism is most desirable for reactive systems as on the same input we want the system to give the same output. Deterministic systems are much easier to analyze and to reason about; and composition of deterministic concurrent systems is simple. Nondeterminism is needed for modelling communication in concurrent systems, whereas in Esterel the synchrony hypothesis takes care of this.

There is no internal clock of the processes but a process can handle several notions of time with several time units. The time considers a *time unit* to be just a repetition of an event (and each event gives another time unit; i.e. another clock).

The semantics of Esterel is based on SOS which does not fit into our algebraic purpose for logics of actions. On the other hand the Esterel programs are compiled into automata which is very similar to our translation of the actions into finite automata.

4.2.3 The mCRL2 language

The mCRL2 is a specification language for distributed systems built in the style of process algebras [GMR+07]. (mCRL2 is the successor of μCRL language [GP93]) The semantics is given as SOS rules and a strong bisimulation relation is defined to capture the equality of the processes. An axiomatization of the operators is given and (relative) completeness of the axiomatization w.r.t. the SOS semantics is shown. Recently a tool set has been released [GKM+08].

Many concepts of the synchronous Kleene algebra are found in mCRL2.

The building blocks of the language are a set of basic actions (parameterized by data types). The basic actions are grouped into sets of basic actions (called multiactions) which are assumed to occur at the same time. The operation on multiactions is the same as $\times$ on $\mathcal{A}_B^{\times}$ in $\mathcal{SKA}$. Over multiactions are defined the basic operators which are essentially the choice, sequence, and conditional (and a few nonessential for process references or for attaching time to a process in the form of a delay). There is no Kleene star concept but recursion is achieved as in process algebras through process definitions and process references. The rest of the operators are for parallel composition and synchronization (as in process algebra terminology), and additional for restriction, blocking, renaming, and communication.

Analyzing a mCRL2 specification means linearization of the specification into what is called a linear process specification (which uses only the basic operators of mCRL2 in a restricted way). This linearization concept is the same as our models for the actions as sets of concurrent strings. The linearizations of mCRL2 are very close to our concurrent strings, only that they need to have more specific notions like the timers on the processes (if any was specified in the original mCRL2 process) or the data arguments of the multiactions.

Overall $\mathcal{SKAT}$ is a simple and clean formalism but not as expressive as mCRL2. $\mathcal{SKAT}$ is tractable and when used in the logical formalism that we mentioned it still yields tractable logics. On the other hand we will need the addition of timing notions to our actions and addition of parameters to the actions (in the form of types as in mCRL2). For this it is wise if we start to investigate first the use of mCRL2 instead of the $\mathcal{SKAT}$.

4.3 Open Problems

1. Consider the equational system defining the $\mathcal{CA}$ algebra in Definition 2.11 (i.e. axioms (1)-(17) of Table 1). Is there an algorithm to decide the *unification problem* in $\mathcal{CA}$? And what is its complexity? A more simple unification problem is to give an algorithm to find the substitution *solution* to the following:

$$\forall \alpha\beta \in \mathcal{CA},\ \alpha \times X = \beta, \text{ where } X \text{ is a variable in } \mathcal{CA}.$$

2. Is there a *canonical form* for $\mathcal{SKA}$(or $\mathcal{SKAT}$); similar to what it was done in Section 2.4 for $\mathcal{CA}$?

3. Give a representation of the automata on concurrent strings in terms of matrices over the $\mathcal{SKA}$ and give an alternative proof of the completeness theorem 2.9 similar to what is done in [Koz94]. This involves the definition of an operation over matrices to simulate the concurrent composition of finite automata over concurrent strings.

4. Find a correct definition of an operation over the synchronous pomsets which would correspond to the $\times$ operation over $\mathcal{SKA}$ actions (or over the concurrent strings).

Acknowledgements

I would like to thank Sergiu Bursuc and Martin Steffen for fruitful discussions and many comments on earlier drafts of this work. Thanks go also to Gerardo Schneider.

References

[Abr79] Karl R. Abrahamson. Modal logic of concurrent nondeterministic programs. In Gilles Kahn, editor, *International Sympoisum on Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 21–33. Springer, July 1979.

[AHK06a] Kamal Aboul-Hosn and Dexter Kozen. Kat-ml: an interactive theorem prover for kleene algebra with tests. *Journal of Applied Non-Classical Logics*, 16(1-2):9–34, 2006.

[AHK06b] Kamal Aboul-Hosn and Dexter Kozen. Relational semantics for higher-order programs. In Tarmo Uustalu, editor, *8th International Conference Mathematics of Program Construction (MPC'06)*, volume 4014 of *Lecture Notes in Computer Science*, pages 29–48. Springer, 2006.

[AHK08] Kamal Aboul-Hosn and Dexter Kozen. Local variable scoping and kleene algebra with tests. *J. Log. Algebr. Program.*, 76(1):3–17, 2008.

[BC85] Gérard Berry and Laurent Cosserat. The esterel synchronous programming language and its mathematical semantics. In Stephen D. Brookes, A. W. Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448. Springer, 1985.

[BG92] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[Bro03] Jan Broersen. *Modal Action Logics for Reasoning About Reactive Systems*. PhD thesis, Vrije Universiteit Amsterdam, 2003.

[BWM01] Jan Broersen, Roel Wieringa, and John-Jules Ch. Meyer. A fixed-point characterization of a deontic logic of regular action. *Fundam. Inf.*, 48(2-3):107–128, 2001.

[CKS96] Ernie Cohen, Dexter Kozen, and Frederick Smith. The Complexity of Kleene Algebra with Tests. Technical report, Cornell University, 1996.

[Coh94] Ernie Cohen. Using kleene algebra to reason about concurrency control. Technical report, Telcordia, 1994.

[Con71] John Horton Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, 1971.

[dBdRR89] J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors. *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings*, volume 354 of *Lecture Notes in Computer Science*. Springer, 1989.

[dS85] Robert de Simone. Higher-Level Synchronising Devices in Meije-SCCS. *Theor. Comput. Sci.*, 37:245–267, 1985.

[FL77] Michael J. Fischer and Richard E. Ladner. Propositional modal logic of programs. In *9th ACM Symposium on Theory of Computing (STOC'77)*, pages 286–294. ACM, 1977.

[Gis84] Jay L. Gischer. *Partial Orders and the Axiomatic Theory of Shuffle*. PhD thesis, CS, Stanford University, 1984.

[GKM+08] Jan Friso Groote, Jeroen Keiren, Aad Mathijssen, Bas Ploeger, Frank Stappers, Carst Tankink, Yaroslav Usenko, Muck van Weerdenburg, Wieger Wesselink, Tim Willemse, and Jeroen van der Wulp. The mCRL2 Toolset. In *Workshop on Advanced Software Development Tools and Techniques*, 2008.

[GMR+07] Jan Friso Groote, Aad Mathijssen, Michel A. Reniers, Yaroslav S. Usenko, and Muck van Weerdenburg. The Formal Specification Language mCRL2. In Ed Brinksma, David Harel, Angelika Mader, Perdita Stevens, and Roel Wieringa, editors, *Methods for Modelling Software Systems (MMOSS'06)*, volume 06351 of *Dagstuhl Seminar Proceedings*, Germany, 2007. Internationales Begegnungs- Und Forschungszentrum Fuer Informatik (IBFI).

[GP93] Jan Friso Groote and Alban Ponse. Proof theory for mucrl: A language for processes with data. In D. J. Andrews, Jan Friso Groote, and C. A. Middelburg, editors, *Semantics of Specification Languages*, Workshops in Computing, pages 232–251. Springer, 1993.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[HMU00] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition, 2000.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hoa07] C. A. R. Hoare. Fine-grain concurrency. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering Series*, pages 1–19. IOS Press, 2007.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *POPL*, pages 14–26, 2001.

[Kap69] Donald M. Kaplan. Regular expressions and the equivalence of programs. *J. Comput. Syst. Sci.*, 3(4):361–386, 1969.

[Kle56] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, pages 3–41, 1956.

[Koz79] Dexter Kozen. On the duality of dynamic algebras and kripke models. In *Logic of Programs, Workshop*, volume 125 of *Lecture Notes in Computer Science*, pages 1–11. Springer-Verlag, 1979.

[Koz90] Dexter Kozen. On kleene algebras and closed semirings. In Branislav Rovan, editor, *Mathematical Foundations of Computer Science (MFCS'90)*, volume 452 of *Lecture Notes in Computer Science*, pages 26–47. Springer, 1990.

[Koz94] Dexter Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

[Koz97a] Dexter Kozen. *The Design and Analysis of Algorithms.* Springer-Verlag, New York, USA, 1997.

[Koz97b] Dexter Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS'97)*, 19(3):427–443, 1997.

[Koz98] Dexter Kozen. Typed kleene algebra. Technical Report 1669, Computer Science Department, Cornell University, March 1998.

[Koz00] Dexter Kozen. On hoare logic and kleene algebra with tests. *Transactions on Computational Logic*, 1(1):60–76, July 2000.

[Koz03a] Dexter Kozen. Automata on Guarded Strings and Applications. In John T. Baldwin, Ruy J. G. B. de Queiroz, and Edward H. Haeusler, editors, *Workshop on Logic, Language, Informations and Computation (WoLLIC'01)*, volume 24 of *Matematica Contemporanea*. Sociedade Brasileira de Matemática, 2003.

[Koz03b] Dexter Kozen. Kleene algebra with tests and the static analysis of programs. Technical report, Cornell University, November 2003.

[Koz08] Dexter Kozen. Nonlocal flow of control and kleene algebra with tests. In *23rd IEEE Symposium on Logic in Computer Science (LICS'08)*, pages 105–117. IEEE Computer Society, 2008.

[KP00] Dexter Kozen and Maria-Christina Patron. Certification of compiler optimizations using kleene algebra with tests. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic*, volume 1861 of *Lecture Notes in Computer Science*, pages 568–582. Springer, 2000.

[KS86] Werner Kuich and Arto Salomaa. *Semirings, Automata, Languages*. Springer-Verlag, Berlin, 1986.

[KW08] Adam Koprowski and Johannes Waldmann. Arctic termination ...below zero. In Andrei Voronkov, editor, *RTA*, volume 5117 of *Lecture Notes in Computer Science*, pages 202–216. Springer, 2008.

[LW04] Carsten Lutz and Dirk Walther. PDL with negation of atomic programs. In David A. Basin and Michaël Rusinowitch, editors, *2nd International Joint Conference on Automated Reasoning (IJCAR'04)*, volume 3097 of *Lecture Notes in Computer Science*, pages 259–273. Springer, 2004.

[Maz88] Antoni W. Mazurkiewicz. Basic notions of trace theory. In de Bakker et al. [dBdRR89], pages 285–363.

[Mey88] John-Jules Ch. Meyer. A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. *Notre Dame Journal of Formal Logic*, 29(1):109–136, 1988.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mil95] Robin Milner. *Communication and concurrency*. Prentice Hall, 1995.

[Möl97] Bernhard Möller. Calculating with pointer structures. In Richard S. Bird and Lambert G. L. T. Meertens, editors, *Algorithmic Languages and Calculi*, volume 95 of *IFIP Conference Proceedings*, pages 24–48. Chapman & Hall, 1997.

[NPW79] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains. In Gilles Kahn, editor, *Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 266–284. Springer, 1979.

[OG76] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.

[Pel85] David Peleg. Concurrent dynamic logic (extended abstract). In *7th ACM Symposium on Theory of Computing (STOC'85)*, pages 232–239. ACM, 1985.

[Pra79] Vaughan R. Pratt. Process logic. In *6th Symposium on Principles of Programming Languages (POPL'79)*, pages 93–100. ACM, 1979.

[Pra86] Vaughan R. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, Feb 1986.

[Pra90] Vaughan R. Pratt. Dynamic algebras as a well-behaved fragment of relation algebras. In Clifford H. Bergman, Roger D. Maddux, and Don L. Pigozzi, editors, *Algebraic Logic and Universal Algebra in Computer Science*, volume 425 of *Lecture Notes in Computer Science*, pages 77–110. Springer-Verlag, 1990.

[Pri08] Cristian Prisacariu. Deontic modalities over synchronous actions – technicalities. Technical Report 381, Univ. Oslo, Norway, 2008.

[PS07] Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In *FMOODS'07*, volume 4468 of *LNCS*, pages 174–189. Springer, 2007.

[Rey02] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74. IEEE Computer Society, 2002.

[Sal66] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of ACM*, 13(1):158–169, 1966.

[Seg82] Krister Segerberg. A deontic logic of action. *Studia Logica*, 41(2):269–282, 1982.

[Seg92] Krister Segerberg. Getting started: Beginnings in the logic of action. *Studia Logica*, 51(3/4):347–378, 1992.

[SM73] Larry J. Stockmeyer and Albert R. Meyer. Word Problems Requiring Exponential Time. In *5th Annual ACM Symposium on Theory of Computing (STOC'73)*, pages 1–9. ACM, 1973.

[vdM96] Ron van der Meyden. The dynamic logic of permission. *Journal of Logic and Computation*, 6(3):465–479, 1996.

[VW68] Georg Henrik Von Wright. *An Essay in Deontic Logic and the General Theory of Action*. North Holland Publishing Co., Amsterdam, 1968.

[Win88] Glynn Winskel. An introduction to event structures. In de Bakker et al. [dBdRR89], pages 364–397.

A Additional Proofs

We give here an alternative proof of Theorem 2.9 using the inverse of the interpretation.

Proof: The forward implication ($\Rightarrow$) can be rewritten as:

$$\mathcal{SKA} \vdash \alpha = \beta \Rightarrow \hat{I}_{\mathcal{SKA}}(\alpha) \doteq \hat{I}_{\mathcal{SKA}}(\beta)$$

where $\vdash$ means that the statement on the right can be deduced from the axioms of the algebra on the left and the classical rules of equational reasoning *reflexivity*, *symmetry*, *transitivity*, and *substitution*.

The proof follows from the fact that $\mathcal{ARCS}$ is a synchronous Kleene algebra (see Theorem 2.3). We use induction on the derivation and prove as base case that the implication holds for the axioms of $\mathcal{SKA}$. For the inductive case we consider the rules of equational reasoning.

For the converse implication ($\Leftarrow$) we prove that the standard interpretation $\hat{I}_{\mathcal{SKA}}$ is an isomorphism up to a congruence relation on actions induced by the equality on action terms. This means that when $\hat{I}_{\mathcal{SKA}}$ is applied to action α and returns a regular concurrent set then one can get by applying the inverse function another action α' which is congruent to the initial α. Having this isomorphism then from two actions α and β we get the same regular concurrent set (by the hypothesis of the theorem) from which we translate back to the same action γ congruent to both α and β which gives the conclusion. Recall that the term algebra $T_{\mathcal{SKA}}$ is free in the class of algebras over the generators $\{\mathbf{1}, \mathbf{0}\} \cup \mathcal{A}_B$. The fact that $\hat{I}_{\mathcal{SKA}}$ is an algebraic isomorphism makes the $\mathcal{ARCS}$ algebra also free in the class of algebras $\mathcal{SKA}$, which means that any property on the regular concurrent sets holds on the action terms and the converse.

As usual, consider a relation $\equiv \subseteq T_{\mathcal{SKA}} \times T_{\mathcal{SKA}}$ defined as: $\alpha \equiv \beta \Leftrightarrow \mathcal{SKA} \vdash \alpha = \beta$. The proof that $\equiv$ is a congruence is classical based on the deduction rules and we leave it to the reader. The rest of the proof is based on the following lemma which basically establishes the existence of the inverse function of the standard interpretation $\hat{I}_{\mathcal{SKA}}$ thus proving that $\hat{I}_{\mathcal{SKA}}$ is an isomorphism.

Lemma A.1 (existence of the inverse of the interpretation)
There exists a map $\hat{I}^{-1}_{\mathcal{SKA}} : \mathcal{ARCS} \to T_{\mathcal{SKA}}$ which is the inverse map up to $\equiv$ of $\hat{I}_{\mathcal{SKA}}$.

Proof: The proof of the lemma involves three parts:

1. $\forall A \in \mathcal{ARCS}$ then $\exists \alpha \in T_{\mathcal{SKA}}$ s.t. $\hat{I}^{-1}_{\mathcal{SKA}}(A) = \alpha$;
2. $\forall A = B$ then $\hat{I}^{-1}_{\mathcal{SKA}}(A) = \hat{I}^{-1}_{\mathcal{SKA}}(B)$;

   The first two guarantee that $\hat{I}^{-1}_{\mathcal{SKA}}$ is a correctly defined function and their proof will be part of the construction of $\hat{I}^{-1}_{\mathcal{SKA}}$.

3. $\hat{I}^{-1}_{\mathcal{SKA}} \circ \hat{I}_{\mathcal{SKA}} = Id/_{\equiv}$ i.e. $\forall \alpha \in T_{\mathcal{SKA}}$ then $\hat{I}^{-1}_{\mathcal{SKA}} \circ \hat{I}_{\mathcal{SKA}}(\alpha) = \alpha'$ and $\alpha \equiv \alpha'$.

We define $\hat{I}^{-1}_{\mathcal{SKA}}$ first for the generator sets and in a second stage it is extended homomorphic to all regular concurrent sets of $\mathcal{ARCS}$.

$$\begin{aligned}\hat{I}^{-1}_{\mathcal{SKA}}(\{\{a\}\}) &= a, \forall a \in \mathcal{A}_B\\ \hat{I}^{-1}_{\mathcal{SKA}}(\mathbf{0}) &= \mathbf{0}\\ \hat{I}^{-1}_{\mathcal{SKA}}(\mathbf{1}) &= \mathbf{1}\end{aligned}$$

Recall that we defined $\mathbf{0}$ and $\mathbf{1}$ to be respectively the regular sets $\emptyset$ and $\{\{\}\}$. The homomorphic extension of $\hat{I}^{-1}_{\mathcal{SKA}}$ to the operators on regular concurrent sets:

$$\begin{aligned}\hat{I}^{-1}_{\mathcal{SKA}}(A + B) &= \hat{I}^{-1}_{\mathcal{SKA}}(A) + \hat{I}^{-1}_{\mathcal{SKA}}(B)\\ \hat{I}^{-1}_{\mathcal{SKA}}(A \cdot B) &= \hat{I}^{-1}_{\mathcal{SKA}}(A) \cdot \hat{I}^{-1}_{\mathcal{SKA}}(B)\\ \hat{I}^{-1}_{\mathcal{SKA}}(A \times B) &= \hat{I}^{-1}_{\mathcal{SKA}}(A) \times \hat{I}^{-1}_{\mathcal{SKA}}(B)\\ \hat{I}^{-1}_{\mathcal{SKA}}(A^*) &= \hat{I}^{-1}_{\mathcal{SKA}}(A)^*\end{aligned}$$

Because $\mathcal{ARCS}$ is generated by $\{\mathbf{0}, \mathbf{1}\} \cup \{\{x\} \mid x \in \mathcal{A}_B\}$ the definition of $\hat{I}^{-1}_{\mathcal{SKA}}$ above is done for all sets of $\mathcal{ARCS}$; therefore item 1 of the lemma is proven. Now we have to prove that $\hat{I}^{-1}_{\mathcal{SKA}}$ returns a unique value for each input, in order to call it a function.

For the proof of the second item of the lemma we need to note that the two sets A and B can be decomposed in different ways depending on the different operators. To overcome this we require that $\hat{I}^{-1}_{\mathcal{SKA}}$ respects the following precedence on the set operators: $^* > + > \cdot > \times$. Therefore, for each set we have only one way of decomposing it and thus only one way of applying $\hat{I}^{-1}_{\mathcal{SKA}}$ and thus obtaining a unique action.

To prove the third item of the lemma we use induction on the structure of the actions. Take as basis the generator actions. For $a \in \mathcal{A}_B$ we have $\hat{I}^{-1}_{\mathcal{SKA}}(\hat{I}_{\mathcal{SKA}}(\alpha)) = \hat{I}^{-1}_{\mathcal{SKA}}(\{\{a\}\}) = a$ by the definitions of $\hat{I}^{-1}_{\mathcal{SKA}}$ and $\hat{I}_{\mathcal{SKA}}$. The inductive are one for each action operator. We take here only one: $\hat{I}^{-1}_{\mathcal{SKA}}(\hat{I}_{\mathcal{SKA}}(\alpha_1 + \alpha_2)) = \hat{I}^{-1}_{\mathcal{SKA}}(\hat{I}_{\mathcal{SKA}}(\alpha_1) + \hat{I}_{\mathcal{SKA}}(\alpha_2)) = \hat{I}^{-1}_{\mathcal{SKA}}(\hat{I}_{\mathcal{SKA}}(\alpha_1)) + \hat{I}^{-1}_{\mathcal{SKA}}(\hat{I}_{\mathcal{SKA}}(\alpha_2))$ which by the inductive hypothesis it is congruent to $\alpha_1 + \alpha_2$. □

To finish the proof of the theorem take $\hat{I}_{\mathcal{SKA}}(\alpha) = \hat{I}_{\mathcal{SKA}}(\beta)$ to be equal regular concurrent sets corresponding to two arbitrary actions α and β. By applying the inverse interpretation we obtain the same action $\hat{I}^{-1}_{\mathcal{SKA}}(\hat{I}_{\mathcal{SKA}}(\alpha)) = \gamma = \hat{I}^{-1}_{\mathcal{SKA}}(\hat{I}_{\mathcal{SKA}}(\beta))$ which in turn is congruent to both actions. Therefore, $\alpha \equiv \gamma \equiv \beta$ from which the goal of the theorem is derived. □