Interaction Expressions – A Powerful Formalism for Describing Inter-Workflow Dependencies

Christian Heinlein, Peter Dadam

Dept. Databases and Information Systems
University of Ulm, Germany
{heinlein,dadam}@informatik.uni-ulm.de

Abstract

Current workflow management technology does not provide adequate means for describing and implementing *workflow ensembles*, i. e., dynamically evolving collections of more or less independent workflows which have to synchronize only now and then. Interaction expressions are proposed as a simple yet powerful formalism to remedy this shortcoming. Besides operators for sequential composition, iteration, and selection, which are well-known from regular expressions, they provide parallel composition and iteration, conjunction, and several advanced features like parametric expressions, multipliers, and quantifiers. The paper introduces interaction expressions semi-formally, gives examples of their typical use, and describes their implementation and integration with state-of-the-art workflow technology. Major design principles, such as orthogonality and implicit and predictive choice, are discussed and compared with several related approaches.

1. Introduction

Everyone of us is familiar with phrases like the following frequently found in envelopes from insurance companies or authorities: “Our apologies if you receive multiple letters from us today. The cost for merging them, however, would be higher than the additional postage.”

In-patients of hospitals might be familiar with even more unpleasant happenings: The nurse comes again to take blood because another test has been ordered; examinations have to be repeated because they have been performed in an incompatible order; an appointment has to be cancelled because the temporary nurse forgot to keep the patient sober, and so on.

The reason for this trouble is the same in both scenarios: Work processes run concurrently and *independently* despite the fact that they are actually interdependent. Even if the *individual* processes (or “workflows”) are controlled by a workflow management system (WfMS), there remains a lack of *inter-workflow* coordination which is not addressed by current workflow technology. When trying to solve these coordination problems with state-of-the-art approaches, it turns out that the language constructs typically provided by workflow description languages (such as sequence, branch, and loop) are not adequate to describe workflow *ensembles*, i. e., dynamically evolving collections of more or less *loosely coupled* workflows.

Taking as an example the different examinations of a patient during an inpatient treatment, it is sensible to model each of them as a separate workflow because it is a self-contained, comprehensible and frequently recurring process by itself. If several such workflows (for the same patient) run in

parallel, most of their activities (or “steps”) are still independent of each other and thus can be executed by themselves, but some of them must be coordinated across workflow boundaries. For an ad-hoc solution of this problem, all affected workflows might be augmented with explicit “coordination steps,” comparable to semaphor operations in parallel programming. It is well-known from that domain, however, that such an “entanglement” of application and synchronization matters leads to very unwieldy and error-prone solutions in practice. Therefore, it is desirable to re-apply a basic principle of workflow management, viz the separation of the control and data flow description from the individual application modules, one level higher: to separate the specification of inter-workflow dependencies from the individual workflow descriptions.

However, since usually neither the number nor the actual types of the workflows “participating” in an ensemble are known in advance, *explicit* descriptions like “Activity A of workflow W_1 must follow activity B of workflow W_2” or “If A has been executed then ... else ...” will lead to an unmanageable number of cases. In order to describe the coordination requirements in a compact and elegant means, a more *implicit* and generic formalism is needed which allows a whole range of possibilities to be described by a single expression or statement.

Regular expressions are a well-known and proven instance of such a formalism from the area of compiler construction. Substituting actions or activities for input characters or tokens, they can be employed to describe *permissible* (or impermissible) *sequences* of executions. Augmenting them with operators for expressing concurrent executions (together with some more advanced features) directly leads to interaction expressions as a conceptually simple yet powerful formalism for describing inter-workflow dependencies.

Despite their conceptual simplicity, interaction (or even regular) expressions tend to become incomprehensible quite quickly when applied to real-world problems. Therefore, a more user-friendly (e. g., graphical or verbal-propositional) notation is needed in order to improve their readability, especially for “computer science laymans” such as workflow designers. The actual appearance of such a notation is irrelevant for and beyond the scope of this paper, but it should be kept in mind throughout the following sections that an actual user will not be burdened with the mathematical formalism presented here.

Section 2 commences by introducing *actions* and *activities* and the six basic operators of interaction expressions. It proceeds with describing more advanced features, such as *multipliers* and *quantifiers*, and demonstrates their use with several examples. It concludes with some remarks about the implementation of interaction expressions by means of an *interaction manager*.

Section 3 uses the example of inpatient examinations to explain the specification of inter-workflow dependencies with interaction expressions. Furthermore, it describes the possibilities for integrating the interaction manager with current workflow technology in order to actually enforce specified constraints.

Section 4 compares interaction expressions with several related approaches, while section 5 concludes with some remarks about current and future work.

2. Interaction Expressions

2.1 Actions and Activities

Activities (or “steps”) are the basic building blocks of workflow descriptions whose *execution* is requested by an external (not necessarily human) user. In order to describe inter-workflow dependencies with interaction expressions, it seems reasonable, therefore, to base them on activities, too. It turns out, however, that things become conceptually easier if interaction expressions are based on *actions* instead, whose executions are assumed to be *instantaneous* events (i. e., to take no time) which *cannot overlap in time*. This does not restrict generality, since it is always possible to describe an acitivity A (having a positive duration in time) by a sequential composition (see section 2.2) of two

actions, A_{start} and A_{term}, representing its start and termination, respectively. By that means it is possible to formally reduce a concurrent execution of activities to a sequential execution of actions which can be captured with formal (e. g., language-theoretic) methods much easier.

2.2 Basic Expressions

Interaction expressions (or simply *expressions*) are used to describe *permissible sequences* of action executions. An expression itself is "executed" by executing a sequence of actions permitted by the expression. If several such sequences exist (which is common), we assume — as a mental model — that the expression always chooses the "right" one, i. e., the sequence the external user has in mind. For a real implementation this means, of course, that it must consider *all* possible alternatives in order to finally accept the user's sequence if it is permissible.

In the following, interaction expressions are defined according to this simple, "intuitive" model. A more precise definition, based on formal language theory, is possible, but beyond the scope of this paper.

An *atomic expression* a is executed by executing the action a.

A *selection* or *disjunction* $x \mid y$ is executed by executing *one* of the expressions x or y.

A *sequential composition* $x - y$ is executed by first executing expression x and afterwards executing expression y.

A *sequential iteration* $x*$ is executed by sequentially executing expression x an arbitrary number of times. This is equivalent to the expression $\lambda \mid x \mid x - x \mid x - x - x \mid \ldots$, where λ represents an empty sequence of actions.

A *parallel composition* $x + y$ is executed by executing the expressions x and y *concurrently*, i. e., by arbitrarily *interleaving* their executions.

A *parallel iteration* $x\#$ is executed by executing an arbitrary number of expressions x in parallel. In analogy to the sequential iteration, this is equivalent to the expression $\lambda \mid x \mid x + x \mid x + x + x \mid \ldots$

A (strict) *conjunction* $x \mathbin{\&\&} y$ is executed by executing a sequence of actions permitted by both expressions, x and y. In practice, this means that actions $x_1, x_2, \ldots$ ($y_1, y_2, \ldots$, respectively) which occur only in expression x (y) but not in expression y (x) will never be permitted by the conjunction $x \mathbin{\&\&} y$. Since this would severely restrict the practical applicability of the conjunction operator, a weaker definition is used in the following which makes sure that an action occuring in one "branch" only is always permitted by the other branch. Formally, this weak conjunction can be defined as follows:

$$x \mathbin{\&} y = (x + y_1* + y_2* + \ldots) \mathbin{\&\&} (y + x_1* + x_2* + \ldots).$$

As a simple example, the expression $(a - b) \mathbin{\&} (a - c)$ is equivalent to $((a - b) + c*) \mathbin{\&\&} ((a - c) + b*)$ whose first (second, resp.) branch permits the sequence ab (ac) with arbitrary interleavings of c's (b's). Therefore, the complete expression permits the sequences abc and acb corresponding to the intuitive meaning of $(a - b) \mathbin{\&} (a - c)$ which simply says that a has to be executed before b and before c.

Weak conjunction is similar, but different from parallel composition: If the "alphabets" (i. e., action sets) of the expressions x and y are disjoint, the expressions $x \mathbin{\&} y$ and $x + y$ are equivalent, i. e., x and y can be executed independently. Otherwise, their execution must be synchronised at every common action.

In order to make complex expressions involving different operators unambiguous, the following precedence rules are defined. The unary operators * and # have highest precedence, followed, in decreasing order, by the binary operators –, +, |, and & (&& is not used in practice and is only needed to define &). Since both unary operators are applied postfix and all binary operators are associative, no rules for implicit bracketing are necessary. Parenthesis can be used, of course, for explicit grouping or improving readability.

2.3 Multipliers

Multipliers are a straight-forward extension of binary operators, similar to the mathematical sum and product operators, Σ and Π. If @ is one of the operators $-$, $+$, $|$, or $\&$, the following definitions shall hold:

$$\overset{n}{@}\, x = x \;@\; \ldots \;@\; x \text{ (n times x) for } n > 0; \quad \overset{n}{@}\, x = \lambda \text{ for } n \leq 0;$$

$$\overset{n}{\underset{k=m}{@}}\, x_k = x_m \;@\; \ldots \;@\; x_n \text{ for arbitrary integers } m, n.$$

In the first line, n is called *factor*, while m and n are called lower resp. upper *bound* in the second line. k is called *index* and can be used as a factor or bound of nested multipliers in the expression x_k. As a simple example, the expression $\overset{k}{-}\, x$ states that x must be executed (sequentially) exactly k times, while $\overset{n}{\underset{k=m}{|}}\, \overset{k}{-}\, x$ allows m to n repetitions of x.

Formally, multipliers are just "syntactic sugar:" they can always be replaced by ordinary binary operators according to their definition. However, multipliers can help to *significantly* reduce the size and, simultaneously, improve the readability of complex expressions. Furthermore, an implementation of interaction expressions might be able to process a multiplier expression more efficiently than its "unfolded" counterpart. Finally, multipliers can be generalized to quantifiers (see section 2.6) which can no longer be directly reduced to the original binary operators.

2.4 Implicit and Predictive Choice

The operators for sequential composition, selection, and sequential iteration resemble well-known constructs of programming or workflow description languages: *sequence*, *branch*, and *loop*. There is a fundamental difference, however: there are no explicit *conditions* in interaction expressions. Given a selection $a \mid b$, for instance, no Boolean expression is used to decide which branch to follow, but simply the fact which action is executed first. The same holds for the termination of an iteration like $a*$: The iteration continues as long as instances of a are executed. As soon as a permissible subsequent action ist executed (e. g., b in the expression $a* - b$), the iteration terminates. We call this policy *implicit choice*.

Some of these "decisions" cannot be made "immediately," however. Given an expression like $a - b \mid a - c$, for example, and executing action a, it is not clear which branch of the selection to choose. Therefore, the decision is "postponed," i. e., both b and c are permitted next. If one of them is executed later, the decision is made "retroactively." The same is true for an expression like $a* - a - b$. If an a is executed, it might correspond to the a in the iteration or to the subsequent one. The decision always "limps one step behind:" if another a is executed, the previous one belonged to the iteration; if b is executed, it corresponded to the subsequent a.

As already mentioned in section 2.2, one can assume as a mental model, that – in analogy to nondeterministic automata – the expression always makes the "right" decision. We call this behavior *predictive choice*.

The combination of these two principles leads to expressions which are very elegant and compact compared to solutions based on explicit conditions.

2.5 Examples

In order to substantiate the statement just made, a few examples shall give an impression of the typical use and expressiveness of interaction expressions.

Activities

As already mentioned in section 2.1, an activity A can be described by the sequential composition $A_{\text{start}} - A_{\text{term}}$. Since, in practice, activities occur so frequently, the name of an activity A may always be used as an abbreviation for this sequence in interaction expressions.

Producer and consumer

Let activity *produce* produce an object which will then be consumed by activity *consume*. If the "transport channel" connecting them can hold only one object, the activities must be executed in alternating sequence starting with *produce*:

$$(produce - consume)\,*.$$

If, on the other hand, the channel's capacity is unlimited, an arbitrary number of *produce-consume* sequences might be executed in parallel:

$$(produce - consume)\,\#.$$

This expression permits an arbitrary number of concurrent producers, but guarantees that the number of active consumers never exceeds the number of completed producers.

If, finally, the channel can hold at most n objects, the number of concurrent *produce-consume* sequences must be limited to n:

$$\overset{n}{+}(produce - consume)\,* = \overset{n}{+}((produce - consume)\,*).$$

Mutual exclusion

If several activities (or complex expressions), A, B, ..., access a common resource and thus must not execute concurrently, they can be synchronized with the expression $(A \mid B \mid \ldots)\,*$.

Readers and writers

If the common resource just mentioned is an object which can be either read or written, concurrency can be increased by distinguishing readers (activity *read*) from writers (activity *write*) and allowing several readers simultaneously:

$$(read\,\# \mid write)\,*.$$

Arbitrary sequence

To describe in a compact form that a number of activities (or complex expressions), A, B, ..., should be executed sequentially in arbitrary order, is quite hard with typical „imperative" control constructs such as sequence, branch, and loop. Using parallel composition, it is possible to prescribe that each activity is executed exactly once: $A + B + \ldots$ Combining this with the above expression for mutual exclusion, immediately yields the desired behavior:

$$(A + B + \ldots)\ \&\ (A \mid B \mid \ldots)\,*.$$

To specify that only m of the activities should be executed, the first part of the expression is replaced by $(A\,? + B\,? + \ldots)$ where $x\,?$ is an abbreviation for $x \mid \lambda$. This guarantees that each activity is executed *at most once*. To express that exactly m activities should be executed, the indefinite iteration

of the second part is replaced by $\overset{m}{-}(A \mid B \mid \ldots)$, yielding the expression

$$(A\,? + B\,? + \ldots)\ \&\ \overset{m}{-}(A \mid B \mid \ldots).$$

If, finally, m_1 to m_2 activities should be executed, the following expression might be used:

$$(A\,? + B\,? + \ldots)\ \&\ \Big|_{m=m_1}^{m_2} \overset{m}{-}(A \mid B \mid \ldots).$$

2.6 Parameters and Quantifiers

In practical applications of, e. g., the readers and writers problem, there is, of course, not only one, but a large (usually unknown) number of objects on which access has to be synchronized *independently*. Therefore, it is reasonable to augment the activities *read* and *write* with a *parameter*, say o, representing the object to be accessed. Then, for every instance of the parameter o, a separate instance of the expression $(read(o)\,\# \mid write(o))\,*$ is needed. This can be specified as follows, using an intuitive generalisation of the multiplier concept:

$$\underset{o}{+}(read(o)\,\# \mid write(o))\,*.$$

By leaving the range of the parameter o unspecified, it is indicated that this range is usually neither known exactly nor of particular interest here. "Virtually," an expression $\underset{p}{+}\,x(p)$ might be imagined as a parallel composition with an infinite number of components:

$$\underset{p}{+}\,x(p) = \overset{\infty}{\underset{i=1}{+}}\,x(p_i),$$

where $\{\, p_1, p_2, \ldots \}$ represents the set of all possible values of the parameter p. (Since parallel composition is commutative and associative, the ordering of the p_i is irrelevant.) Practically, this means that the expression $\underset{p}{+}\,x(p)$ will "create on demand" a new instance of the subexpression $x(p)$ for every new value of the parameter p.

Similarly, it is possible to "define" infinite selections as

$$\Big|_{p}\, x(p) = \Big|_{i=1}^{\infty}\, x(p_i).$$

Due to their conceptual similarity to the universal and existential quantifiers known from predicate logic, $\underset{p}{+}$ and $\Big|_{p}$ are called *quantifiers*, too.

Despite the fact that parametric expressions and quantifiers are of essential importance in practice, space does not permit to treat them in more detail here nor to give more formal and sound definitions of them. The same is true for additional features of interaction expressions like *substitution* (an action can be declared as a substitute for one or more other actions), *explicit conditions* (if the principle of implicit choice is not appropriate or sufficient for a particular application), and *temporal aspects* (conditions whose truth value changes with time).

2.7 Interaction Manager

Interaction expressions can be implemented by an *interaction manager* as follows. First, an expression x is transformed to a graph-based internal representation $G(x)$, similar to [34]. Based on this graph, a notion of *state* is defined: An *atomic state* corresponds to a node of the graph and indicates that the action contained in that node is currently permissible. A *product state*, which is introduced to deal with concurrency, is a tuple of atomic states indicating that *all* corresponding actions are currently permissible. Finally, a *sum state*, which is necessary to deal with

nondeterminism and to implement the principle of predictive choice, is a set of product states indicating that *one* of these product states is the "right" one corresponding to the external user's intention.

Given an expression x, the interaction manager maintains a current (sum) state $S(x)$ and a set of permissible actions, $P(x)$, consisting of all actions contained in the nodes of $S(x)$. As a simple example, the expression $x = (a - b \mid c - d) + (e - f)$ is transformed to the graph of figure 1. Its initial state is $S(x) = \{ (a, e), (c, e) \}$, and thus the set of permissible actions is $P(x) = \{ a, c, e \}$. If an action of $P(x)$ is executed by a user, e. g., a, a new state is determined according to a state transition function derived from the graph (details are beyond the scope of this paper) leading to a new set of permissible actions. In the example, the atomic state a is replaced by its successor b (and consequently, (a, e) is replaced by (b, e)), while the atomic state c (and consequently (c, e)) is discarded because this alternative has proven false. Therefore, the new sum state becomes $S(x) = \{ (b, e) \}$, and consequently $P(x) = \{ b, e \}$.

If more than one expression is given to the interaction manager, an action a should be permissible if and only if it is permitted by all expressions containing a. Formally, this is achieved by combining the expressions to a single expression using (weak) conjunction. In practice, however, it is easier to maintain a separate graph, state, and set of permissible actions for every expression, especially if expressions shall be added and removed dynamically.

The principle of predictive choice which forces the interaction manager to pursue all possible interpretations of an action execution, might lead to intractable state explosions when implemented naively. In order to surmount this problem, it is necessary to introduce an *equivalence relation* over product states which is able to reduce otherwise exponentially large sets of equivalent product states to a single representative. By that means, it becomes possible to process even complex expressions quite efficiently. Precise complexity analyses are a topic of future work, however.

Control Flow Server

Even though interaction expressions are primarily intended for describing *inter*-workflow dependencies, they might be used to model the "behavioral aspect" (control flow) of *individual* workflows, too. In that case, the interaction manager can be used as the "control flow server" in a modular workflow management system architecture like MOBILE [5, 21]. Other aspects, like organizational modeling or data flow, are described with other appropriate formalisms and implemented by corresponding servers in such an architecture.

3. Inter-Workflow Dependencies

3.1 Specification

To make sure that individual workflow descriptions and inter-workflow dependencies will harmonize properly, it is necessary to define a "vocabulary" of activities (workflow steps) in advance which serves as a common basis for both kinds of descriptions.

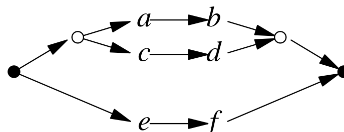

Figure 1: Graph-based representation $G(x)$ for expression $x = (a - b \mid c - d) + (e - f)$.

For the domain of inpatient examinations mentioned in the introduction, the following set of activities might be useful:

- *order*: an examination (e. g., an ultrasonography or an X-ray) is ordered by a ward doctor;
- *appoint*: an appointment is made with the department supplying the examination (e. g., internal medicine or radiology);
- *prepare*: the patient is prepared by a ward nurse;
- *inform*: the patient is informed by a ward doctor about potiential risks of the examination;
- *call*: the patient is called to the examination room;
- *examine*: the examination is actually performed;
- *report*: a report is written (or dictated) by the examining physician;
- *read*: the report is read by the ward doctor.

Given such a vocabulary, it is possible, on the one hand, to use an arbitrary workflow definition language to define individual workflow types such as the "examination workflow" shown in figure 2. On the other hand, interaction expressions (resp. a more user-friendly notation of them; cf. section 1) can be used to specify "global constraints" which are independent of the particular use of activities in individual workflows. For example, if several examination workflows are performed in parallel for the same patient, their *call*-*examine* sequences must be serialized since a patient cannot be in two examination rooms at the same time. Furthermore, it is impossible to inform or prepare him for another examination while such a sequence is in progress. (In database terms, *call* and *examine* require *exclusive* access to the patient.) On the other hand, it is possible and even desirable to perform several preparations (e. g., taking blood) or informational talks simultaneously. These requirements can be summarized (for a single patient) with the following expression resembling the readers and writers problem:

$$((call - examine) \mid prepare\,\# \mid inform\,\#)\,*.$$

In analogy to this problem, the expression should be quantified over all patients, however, in order to guarantee individual synchronization for every patient (cf. section 2.6):

$$\mathop{+}_{p} ((call(p) - examine(p)) \mid prepare(p)\,\# \mid inform(p)\,\#)\,*.$$

3.2 Integration with Current Workflow Technology

To actually *enforce* inter-workflow constraints like the one just shown, two general problems have to be solved. First, the interaction manager implementing the expressions must be able to *monitor* all actions performed by the WfMS (i. e., start and termination of workflow steps) in order to adjust its state and set of permissible actions accordingly. Second, it must be able to *control* which actions a user might execute (i. e., which workflow steps he might start) in order to make sure that only permissible actions will be executed. This is especially difficult if actions alternate between being permissible and being not. The above expression, for example, initially permits to start any one of the

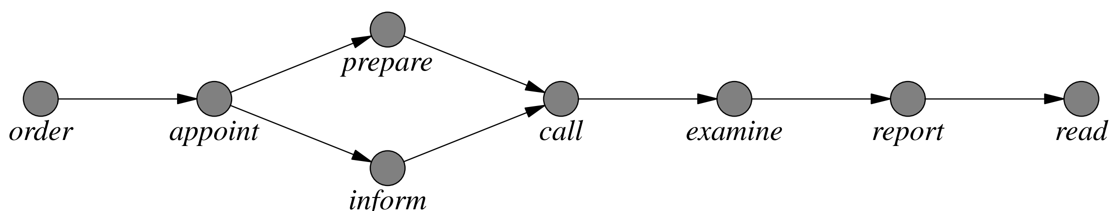

Figure 2: Simple examination workflow.

steps *call*, *prepare*, and *inform* for arbitrary patients, i. e., all of them are allowed to appear in appropriate users' worklists. As soon as one of them is executed for a particular patient, however, e. g., *prepare*(p_1), *call*(p_1) and *inform*(p_1) cease to be permissible and therefore should *disappear* from all worklists! Although this is very similar to the requirement that a single step should disappear from all worklists as soon as it is executed by one user, it is not supported for multiple steps by current WfMSs.

A specific solution to these general problems depends more or less on the peculiarities of the particular WfMS. To solve the monitoring problem, for example, it is necessary to identify some externally "observable" event that takes place whenever a step is started or terminated by a user. If the WfMS records such events in an externally accessible database system, for example, database triggers might be used to intercept them and to inform the interaction manager. Other systems, e. g., ProMInanD [22, 23], invoke an external program whenever a workflow instance is copied between the global workflow server and a user's client, and there is a one-to-one correspondence between copying of workflow instances and the execution of workflow steps: If an activity is started, the workflow instance is copied from the server to the client; if an activity is terminated, it is copied back to the server. Replacing the external copy program with a "wrapper" which informs the interaction manager before executing the original copy program is a safe yet simple method to solve the monitoring problem in that case.

Furthermore, it is a partial solution to the controlling problem, too, if the wrapper program does not only inform the interaction manager about an activity to be started, but also asks whether this activity is permissible. If it is not, the wrapper might "pull the emergency brake" by returning an error code instead of executing the copy program, causing the WfMS to tell the user that the step is currently not executable (although the actual error message might not be very instructive).

Of course, this is not an ideal solution from the user's point of view since it might happen that most of the steps appearing in his worklist are not actually executable. In order to fully solve the controlling problem, the interaction manager must be able to temporarily remove impermissible activities from users' worklists. Many WfMSs, including ProMInanD, automatically remove an activity from a user's worklist as soon as it has been started – or reserved – by another user. This behavior can be exploited as follows. If a particular activity *A* is executable from the WfMS's point of view and therefore would appear in appropriate users' worklists, but *A* is not permissible from the interaction manager's point of view, a priviliged pseudo-user "grabs" the activity by declaring that he is willing to execute it. This causes the WfMS to remove the activity from all other user's worklists. The pseudo-user does not actually execute the activity, however, but only waits until it becomes permitted by the interaction manager. Then he declares to give it back to the pool of authorized users, causing the WfMS to re-insert it into the corresponding users' worklists. The only prerequisite for this trick to work is to have an "omnipotent" user having the privilege to execute resp. reserve *every* activity in the system. Even if the WfMS does not directly support such a "super-user," it is usually possible to declare a user as a "temporary" substitute for all other users and by that means inherit all of their privileges.

4. Related Work

Interaction expressions are, of course, not the first attempt to describe dependencies between concurrent activities. Quite similar approaches are: regular expressions [19, 2], CoCoA ordering rules [12], "life cycles" [10], path expressions [6, 7], and synchronization expressions [14, 15]. Modeling techniques, such as Petri nets [31, 29], sequence diagrams [37], and statecharts [16], also have something in common with interaction expressions. So why "yet another" formalism?

Well, none of the alternatives just mentioned is complete with respect to the approach presented here. Regular expressions, in their basic form, do not provide any constructs for expressing concurrency. The same is true for the ordering rules of CoCoA and for sequence diagrams which are just a graphical representation of regular expressions. Life cycles are an example of a straight-

forward extension of regular expressions with parallel composition. Similarly, statecharts and Petri nets provide parallel composition, but like almost all other approaches, they do neither provide parallel iteration nor conjunction, which have turned out to be quite important constructs in our experience (confer the examples in section 2.5). A very limited form of parallel iteration is supported by path expressions, where multiple instances of a simple expression (containing only selection and sequential composition) might be executed concurrently. Parallel iteration has also been suggested for synchronization expressions [14], but discarded in the final version [15]. Synchronization expressions provide conjunction, too, but only in its strict form which is unwieldy for practical applications (cf. section 2.2).

Despite their significance for real-world applications (cf. section 3.1), parametric expressions and quantifiers are usually not supported by related approaches. Path expressions are a partial exception: Since a path expression is part of an abstract data type (or “class”) definition, it is implicitly quantified over all instances of that type. This makes it impossible, however, to specify dependencies between “methods” of different classes. CoCoA ordering rules allow for parametric actions and provide a “forall” operator corresponding to our + quantifier.

Many approaches lack orthogonality, mostly to be easily implementable. Path expressions, for example, forbid nested iterations, while synchronization expressions do not allow the same action to occur in both branches of a parallel composition, inhibiting our simple interpretation of parallel iteration as $x\# = \lambda \mid x \mid x + x \mid \ldots$ Even though restrictions like these might seem reasonable at first glance (“Who will need such complicated expressions?”) and may simplify an implementation of the formalism, they should be avoided whenever possible, because they will generally impede the development of expressions (“Is this combination of operators allowed?”) and especially obstruct the modular composition of complex expressions from independently developed subexpressions.

Besides these practically oriented approaches, there is a variety of related theoretical approaches, too: trace theory [1, 27], process algebra [4, 17], CSP [18], CCS [28], and maybe others. While these formalisms are well-suited for specifying, studying, and verifying properties of concurrent systems, their practical applicability to our problem domain is limited. Usually, they do not provide explicit iteration operators, but employ recursion equations for that purpose. While this is quite elegant from a mathematical point of view, we prefer “closed” and self-contained expressions for practical applications.

Workflow description languages usually provide only standard “imperative” control constructs, such as sequence, branch, loop, and possibly concurrency, which are too “explicit” and inflexible for modelling workflow ensembles, i. e., dynamically evolving collections of more or less independent workflows which have to synchronize only now and then (cf. sections 1 and 2.4). The same holds for most advanced transaction models [11, 20], such as ConTracts [35] or FlexTransactions [25, 26], whose emphasis and strength lie in different areas, such as reliability, (forward) recovery, etc. of indivudal workflows. These aspects are considered orthogonal to inter-workflow dependencies.

An approach that goes in line with ours is presented in [30]. However, since only regular expressions are allowed there to describe permissible or impermissible interleavings of subtransactions, the expressive power is quite limited.

Klein [24] has proposed two primitives to describe “inter-task dependencies” [3, 33]: order dependency and existence dependency. Similar to interaction expressions, it is possible to describe a whole range of permissible event sequences (corresponding to action sequences in our terminology) with a single expression based on these primitives. When comparing the expressiveness of the two approaches, it can be noticed that every “Klein expression” can be transformed to an equivalent interaction expression in principle (in the worst case by enumerating all permissible sequences, which is not very elegant, of course), but interaction expressions involving iteration cannot be expressed with the Klein primitives.

Some WfMSs provide an event mechanism [32, 36] which can be employed to synchronize activities across workflow boundaries, similar to the idea of “coordination steps” mentioned in section 1. While concepts like these might help to *implement* inter-workflow dependencies at run time, they appear to

be too primitive and restricted to be useful for *describing* them at build time. Furthermore, they are not able to fully solve the controlling problem mentioned in section 3.2, since events can only be used to *enable* permissible activities, but not to *disable* (i. e., remove from worklists) impermissible ones.

An approach which is directly concerned with interacting workflows is presented in [9]. The inter-workflow dependencies identified there, are quite similar to ours. A “condition-action dependency”, $\{ A_1, A_2, \ldots \} <_{ca} B$, for example, corresponds to the interaction expression $(A_1 \mid A_2 \mid \ldots) - B$. The concept of “workflow integration,” however, is proposed only as a “conceptual device” helping to “build an integrated frame in which interactions can be better understood.” Furthermore, it seems that only interactions between “cooperating” workflows are studied, whereas interaction expressions can deal with “competing” workflows equally well.

5. Summary and Outlook

Using the example of interrelated medical examinations as a representative, we have shown that inter-workflow dependencies cannot be ignored for practical applications. Since neither “ad-hoc” solutions like explicit coordination steps nor “hard-wired” specifications like if-then-else statements are really satisfactory in practice, we have introduced interaction expressions as a simple yet powerful formalism for describing inter-workflow dependencies in an adequate way. With only a few basic building blocks − sequential and parallel composition, the corresponding iterations, Boolean operators, and their generalisation to multipliers and quantifiers − it is possible to solve a broad spectrum of synchronization problems. The principles of implicit and predictive choice allow for very compact and elegant expressions compared to solutions based on more explicit constructs.

A prototypical implementation of interaction expressions by means of an interaction manager as well as an interface to the WfMS ProMInanD have been built as a feasibility study. As already mentioned in section 2.7, the implementation performs reasonably well even for complex expressions, but no detailed analyses of its worst-case complexity have been carried out so far.

In order to emphasize their practical usefulness, interaction expressions have been introduced rather informally. A theoretically founded definition based on formal languages is currently under construction and shall be completed in the near future. Based on the formal semantics, it is possible to develop a calculus which can be used to “optimize” expressions, i. e., transform an expression to an equivalent one which can be implemented more efficiently, and to recognize and eliminate “malignant” expressions (e. g., a * # which can be replaced by a #) causing the current implementation to loop infinitely. (This is a well-known problem with simple regular expressions, too [34].)

In order to make complex expressions more comprehensible, especially for “computer science laymans,” a first draft of a graphical representation of interaction expressions is under preparation. Furthermore, an *abstraction* mechanism is being developed, which allows arbitrary subexpressions to be replaced either by (parametric) macros or by user-defined (unary, binary, or n-ary) operators. By that means, it is possible for an “expert” to predefine a set of application-specific macros and operators which can afterwards be used by a “layman” without understanding their possibly complicated internal structure. This work will be continued in order to make interaction expressions really suitable for workflow designers.

Currently, interaction expressions solve only one “half” of the inter-workflow coordination problem, viz *synchronization*. The other half, inter-workflow *communication*, has not been addressed so far. It has to be explored whether it is more reasonable to extend interaction expressions to incorporate that aspect or to develop an independent formalism. In any case, the remarks about explicit vs. implict appraoches apply analogously: Typical “explicit” approaches, such as message passing, will not be adequate to deal with dynamically evolving workflow ensembles. More “implicit” and flexible approaches like, e. g., generative communication [8, 13], seem to be more promising here.

References

[1] I. J. Aalbersberg, G. Rozenberg: “Theory of Traces.” *Theoretical Computer Science* 60, 1988, 1–82.

[2] A. V. Aho, R. Sethi, J. D. Ullman: *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[3] P. C. Attie, M. P. Singh, A. Sheth, M. Rusinkiewicz: “Specifying and Enforcing Intertask Dependencies.” In: R. Agrawal, S. Baker, D. Bell (eds.): *Proc. 19th Int. Conf. on Very Large Data Bases (VLDB)* (Dublin, Ireland, August 1993). 1993, 134–145.

[4] J. C. M. Baeten, W. P. Weijland: *Process Algebra*. Cambridge University Press, Cambridge, 1990.

[5] C. Bußler, S. Jablonski: “Die Architektur des modularen Workflow-Management-Systems MOBILE.” In: G. Vossen, J. Becker (eds.): *Geschäftsprozeßmodellierung und Workflow-Management. Modelle, Methoden, Werkzeuge*. International Thomson Publishing, Bonn, 1996, 369–388.

[6] R. H. Campbell, A. N. Habermann: “The Specification of Process Synchronization by Path Expressions.” In: E. Gelenbe, C. Kaiser (eds.): *Operating Systems* (International Symposium; Rocquencourt, April 1974; Proceedings). Lecture Notes in Computer Science 16, Springer-Verlag, Berlin, 1974, 89–102.

[7] R. H. Campbell, R. B. Kolstad: “An Overview of Path PASCAL’s Design and Path PASCAL User Manual.” *ACM SIGPLAN Notices* 15 (9) September 1980, 13–24.

[8] N. Carriero, D. Gelernter: “Linda in Context.” *Communications of the ACM* 32 (4) April 1989, 444–458.

[9] F. Casati, S. Ceri, B. Pernici, G. Pozzi: “Semantic WorkFlow Interoperability.” In: P. Apers, M. Bouzeghoub, G. Gardarin (eds.): *Advances in Database Technology – EDBT’96* (5th Int. Conf. on Extending Database Technology; Avignon, France, March 1996; Proceedings). Lecture Notes in Computer Science 1057, Springer-Verlag, Berlin, 1996, 443–462.

[10] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes: *Object-Oriented Development. The Fusion Method*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[11] A. K. Elmagarmid (ed.): *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[12] F. J. Faase, S. J. Even, R. A. de By: *TransCoop Deliverable IV.3 (CoCoA)*. University of Twente, The Netherlands, February 1996.

[13] D. Gelernter: “Generative Communication in Linda.” *ACM Transactions on Programming Languages and Systems* 7 (1) January 1985, 80–112.

[14] R. Govindarajan, L. Guo, S. Yu, P. Wang: “ParC Project: Practical Constructs for Parallel Programming Languages.” In: *IEEE Proc. of the 15th Ann. Int. Computer Software and Applications Conference*. 1991, 183–189.

[15] L. Guo, K. Salomaa, S. Yu: “On Synchronization Languages.” *Fundamenta Informaticae* 25 (3+4) March 1996, 423–436.

[16] D. Harel: “Statecharts: A Visual Formalism for Complex Systems.” *Science of Computer Programming* 8, 1987, 231–274.

[17] M. Hennessy: *Algebraic Theory of Processes*. The MIT Press, Cambridge, MA, 1989.

[18] C. A. R. Hoare: *Communicating Sequential Processes*. Prentice-Hall, London, 1985.

[19] J. E. Hopcroft, J. D. Ullman: *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, 1979.

[20] M. Hsu (ed.): “Special Issue on Workflow and Extended Transaction Systems.” *IEEE Data Engineering Bulletin* 16 (2) June 1993.

[21] S. Jablonski: “MOBILE: A Modular Workflow Model and Architecture.” In: *Proc. 4th Int. Working Conference on Dynamic Modelling and Information Systems* (Noordwijkerhout, The Netherlands, 1994). 1994.

[22] B. Karbe, N. Ramsperger, P. Weiss: “Support of Cooperative Work by Electronic Circulation Folders.” In: F. H. Lochovsky, R. B. Allen (eds.): *Conf. on Office Information Systems* (Cambridge, MA, April 1990). *ACM SIGOIS Bulletin* 11 (2+3) 1990, 109–117.

[23] B. Karbe, N. Ramsperger: "Concepts and Implementation of Migrating Office Processes." In: W. Bianer, D. Hernandez (eds.): *Verteilte künstliche Intelligenz und kooperatives Arbeiten*. Springer-Verlag, Berlin, 1991, 136–147.

[24] J. Klein: "Advanced Rule Driven Transaction Management." In: *Proc. 36th IEEE Computer Society Int. Conf. (COMPCON)* (San Francisco, CA, March 1991). 1991.

[25] E. Kühn, F. Puntigam, A. K. Elmagarmid: "Multidatabase Transaction and Query Processing in Logic." In: A. K. Elmagarmid (ed.): *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, CA, 1992, 297–348.

[26] Y. Leu: "Composing Multidatabase Applications using Flexible Transactions." *IEEE Data Engineering Bulletin* 14 (1) March 1991, 29–33.

[27] A. Mazurkiewicz: "Trace Theory." In: W. Brauer, et al. (eds.): *Petri Nets: Applications and Relationships to other Models of Concurrency* (Proceedings of an advanced course; Bad Honnef, Germany, September 1986). Lecture Notes in Computer Science 255, Springer-Verlag, Berlin, 1987, 279–324.

[28] R. Milner: *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92, Springer-Verlag, Berlin, 1980.

[29] J. L. Peterson: "Petri Nets." *ACM Computing Surveys* 9 (3) September 1977, 223–252.

[30] R. Rastogi, S. Mehrotra, H. F. Korth, A. Silberschatz: "Transcending the Serializability Requirement." *IEEE Data Engineering Bulletin* 16 (2) June 1993, 8–11.

[31] W. Reisig: *Petri Nets. An Introduction*. EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin, 1985.

[32] Siemens Nixdorf Informationssysteme AG: *WorkParty Benutzerhandbuch* (Version 2.0), August 1995.

[33] J. Tang, J. Veijalainen: "Enforcing Inter-Task Dependencies in Transactional Workflows." In: S. Laufmann, S. Spaccapietra, T. Yokoi (eds.): *Proc. 3rd Int. Conf. on Cooperative Information Systems (CoopIS)* (Vienna, Austria, May 1995). 1995, 72–86.

[34] K. Thompson: "Regular Expression Search Algorithm." *Communications of the ACM* 11 (6) June 1968, 419–422.

[35] H. Wächter, A. Reuter: "The ConTract Model." In: A. K. Elmagarmid (ed.): *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, CA, 1992, 219–263.

[36] H. Wächter: "Flexible Business Processing with SAP Business Workflow 3.0." (Invited Presentation; Extended Abstract) In: *Int. Workshop on High Performance Transaction Processing Systems* (Asilomar, CA, September 1995). 1995.

[37] P. Zave: "A Distributed Alternative to Finite-State-Machine Specifications." *ACM Transactions on Programming Languages and Systems* 7 (1) January 1985, 10–36.