Workflow and Process Synchronization with Interaction Expressions and Graphs

Christian Heinlein

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

Abstract

Current workflow management technology does not provide adequate means for *inter-workflow coordination* as concurrently executing workflows are considered completely independent. While this simplified view might suffice for one application domain or the other, there are many real-world application scenarios where workflows − though independently modeled in order to remain comprehensible and manageable − are semantically interrelated. As pragmatical approaches, like merging interdependent workflows or inter-workflow message passing, do not satisfactorily solve the inter-workflow coordination problem, *interaction expressions and graphs* are proposed as a simple yet powerful formalism for the specification and implementation of synchronization conditions in general and inter-workflow dependencies in particular. In addition to a graph-based semi-formal interpretation of the formalism, a precise formal semantics, an equivalent operational semantics, an efficient implementation of the latter, and detailed complexity analyses have been developed allowing the formalism to be actually applied to solve real-world problems like inter-workflow coordination.

1. Introduction

Inter-Workflow Dependencies

Current workflow management systems (WfMS), whether commercial products or research prototypes, do not provide adequate means for *inter-workflow coordination* as concurrently executing workflows are considered completely independent. While this simplified view might suffice for one application domain or the other, there are many real-world application scenarios where workflows − though independently modeled in order to remain comprehensible and manageable − are semantically interrelated. As a simple example from the domain of medicine, consider the examination workflows depicted in Fig. 1 describing the performance of an ultrasonography (left) and an endoscopy (right) including necessary pre- and postprocessing steps like scheduling, report writing, etc. As long as these workflows refer to different patients, they might well be executed independently. If the same patient is involved, however, the activities *prepare*, *inform*, *call*, and *perform*[1] must be synchronized somehow as they access a “limited resource,” viz the patient under consideration. If for example, the activities *order*, *schedule*, *prepare*, and *inform* of both workflows have already been performed, the activities *call* are to be executed next, i. e., they will be inserted into the worklists of appropriate users − e. g., medical assistents of the ultrasonography and endoscopy departments, respectively − by the WfMS. As soon as one of these activities is actually executed, however, the other one should temporarily disappear from the worklists − or at least become marked as currently not executable − as a patient cannot be called to a second examination as long as he passes through the first one. Only after completion of the first examination (activity *perform* of the corresponding workflow), activity *call* of the second workflow should become executable again, i. e., reappear in appropriate worklists.

[1] For the sake of simplicity, only the verb of an activity (e. g., *prepare* instead of *prepare patient*) is used throughout the following text.

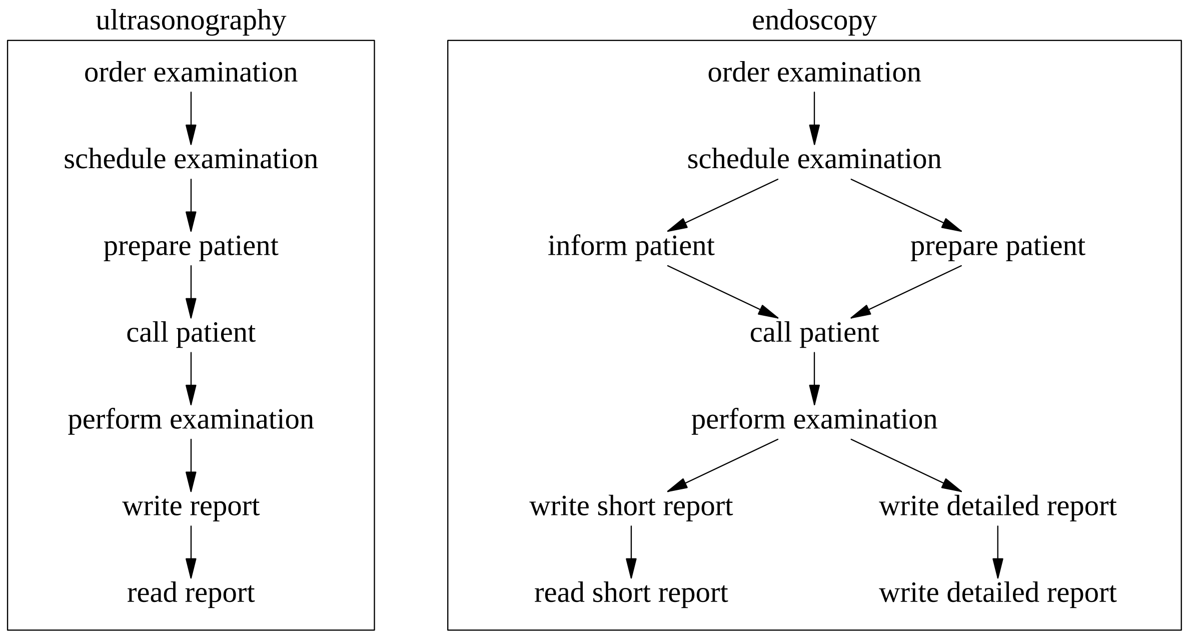

Figure 1: Medical examination workflows

Impractical Solutions

As current WfMSs neither provide adequate means to describe nor to implement such inter-workflow dependencies, one might resort to rather pragmatical approaches like merging interdependent workflows into a single workflow to transform *inter*-workflow dependencies to ordinary *intra*-workflow control flows. As soon as not only two, but maybe five, ten or twenty interrelated workflows have to be merged, however, the resulting workflows will reach magnitudes which are no longer comprehensible nor manageable in practice. Furthermore, a host of 2^n merged workflows would be necessary to capture every possible combination of n original workflows. Finally, typical intra-workflow control structures, like sequence, conditional and parallel branching, and possibly loop, would force a workflow designer to prescribe a particular ordering of the examinations ultrasonography and endoscopy in the example above (more precisely, of the activities *call* and *perform* of the corresponding workflows) as they do not allow to describe a sequential execution in either order. For these reasons, the idea to simply "define away" inter-workflow dependencies by translating them to well-known intra-workflow control flows has to be abandoned.

Another apparently attractive idea uses inter-workflow messaging or event services provided by some WfMSs to explicitly synchronize concurrently executing workflows. While this approach avoids the creation of unmanageable "mega workflows" as it retains the structure of the original workflows, it does not solve the "combinational explosion" problem as, in principal, 2^n variations of each workflow are necessary describing which messages to exchange with concurrent workflows depending on the "cast" of the actually executing "workflow ensemble." Similarly, the "mutual exclusion" problem, i. e., describing that the examinations can be performed in either order, cannot be solved with this approach as inter-workflow messages cannot be used to temporarily disable activities which have already been enabled by the WfMS. Therefore, the idea of reducing inter-workflow dependencies to bare message passing has likewise to be abandoned.

A common problem of both approaches not mentioned so far is their principal unability to deal with *dynamic* workflow ensembles where the number and the actual participants of a set of concurrently executing workflows is not known in advance and might change with time. As currently executing workflows might always terminate and additional workflows can be initiated by a user at any time, however, dynamically evolving ensembles are actually the normal case and must thus be supported accordingly.

Extended Regular Expression Formalisms

In order to find a satisfactory solution to the inter-workflow coordination problem at all, it is absolutely necessary to strictly *separate* inter-workflow synchronization aspects from individual workflow descriptions and to use an extremely flexible and declarative formalism for their specification. In some sense, this means to reapply a basic principle of workflow management, viz the separation of the overall control and data flow specification of a workflow from the implementation of individual application modules, one level higher: Inter-workflow synchronization aspects are extracted from individual workflows and described on a separate level using a tailored and well-suited formalism.

In the past, similar approaches have been proposed for the synchronization of parallel programs [2, 10, 25]. Instead of directly encoding synchronization conditions using semaphor operations or the like in individual procedure implementations, an abstract formalism based on extended regular expressions is used to describe them separately in a compact, legible and easily adaptable manner. The basic idea with these formalisms is to interpret the *language* of an expression, i. e., the set of words it accepts, as set of *permissible execution sequences* of actions where actions correspond to the start or termination of individual procedures. By that means, it is indeed possible to specify synchronization conditions in a very flexible and declarative way.

Despite the fact that many similar formalisms have been proposed over the years (cf. Fig. 2), each of them lacks one important operator or the other. By carefully analysing the overall spectrum of operators provided, one can identify three pairs of “complementary” or dual operators: There are two basic composition operators, *sequential* and *parallel composition*, two corresponding closure operators, *sequential* and *parallel iteration*, and two Boolean operators, *disjunction* and *conjunction*.[2] Furthermore, the concept of *parametric expressions* and *quantifiers* can be found in a restricted form in some approaches. Apart from the fact, that none of the formalisms proposed so far is conceptually comprehensive or complete with respect to the others, most of them do not allow operators to be arbitrarily combined, but impose considerable restrictions on their nesting. In path expressions, for example, the parallel iteration operator must not contain other parallel iterations, while operands of a parallel composition in synchronization expressions must have disjoint alphabets.

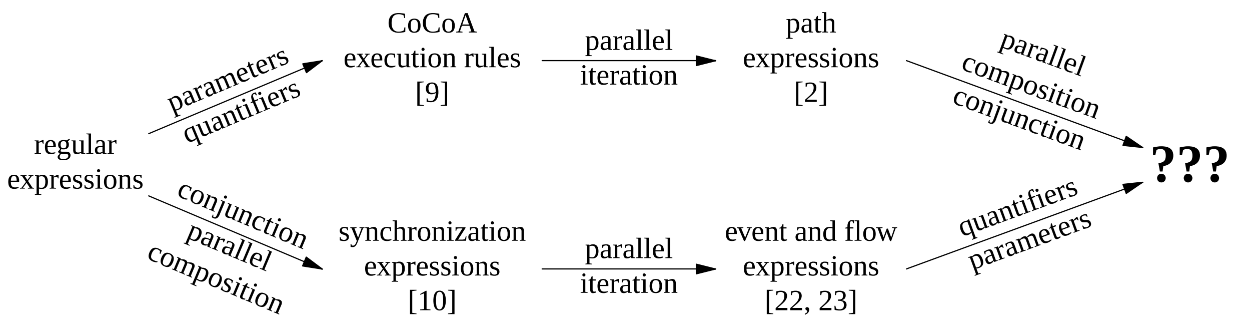

Figure 2: Formalisms based on extended regular expressions

Interaction Expressions and Graphs

This lack of orthogonality on the one hand and the conceptual incompleteness of the formalisms on the other hand calls for the development of a new formalism to describe synchronization requirements

[2] Note that regular expressions provide just the first operator of each pair: sequential composition (sequence), sequential iteration (Kleene closure), and disjunction (choice).

which is at least conceptually complete and fully orthogonal and thus can fill the hole depicted by the question marks in Fig. 2. Despite its conceptual completeness, such a formalism should also be flexibly extensible with user-defined operators in order to be optimally useful in different application domains. Furthermore, it should be readily comprehensible even for mathematically ignorant persons, which suggests the use of a graphical representation instead of or in addition to a formal notation. Last but not least, the proposed formalism must be efficiently implementable in order to be practically useful, and the implementation should be − in contrast to, e. g., Petri nets and process algebras − completely deterministic.

In order to meet the requirements just enumerated, *interaction expressions and graphs* have been developed in the author's Ph. D. thesis [11, 12] as a simple yet powerful formalism for the *expression*- or *graph*-based specification and implementation of *inter-action* dependencies, i. e., synchronization conditions. Interaction graphs, which constitute the graphical, user-oriented view of the formalism, are introduced in Sec. 2, while interaction expressions, their formal counterpart, are treated in Sec. 3. Sections 4, 5, and 6 dealing with the operational semantics, implementation, and complexity of interaction expressions, respectively, pave the way for their practical application which is illustrated in Sec. 7 by describing their integration with workflow management systems. Finally, Sec. 8 concludes the paper.

2. Interaction Graphs

Example

Figure 3 shows a typical example of an interaction graph specifying a generic *integrity constraint for patients* by describing necessary synchronization requirements for the activities *prepare*, *inform*, *call*, and *perform*. As all these activities refer to a particular patient p as well as a particular examination x, they possess corresponding parameters p and x containing, for example, a social security number identifying a patient and a symbolic value like *sono* or *endo* representing an examination, respectively.[3] The ellipses containing flash symbols, which − in contrast to the predefined circular operators − constitute a user-defined operator, represent a *mutual exclusion* describing that a patient p might either pass through exactly one examination x (middle branch) or be prepared for or informed about several examinations x simultaneously (upper and lower branch, respectively). The "for some x" quantifiers $\underset{x}{\bigcirc}\cdots\underset{x}{\bigcirc}$ specify that their body, i. e., the subgraph in between, must be traversed for exactly one arbitrarily chosen value of the parameter x, while the body of the "for all p" quantifier

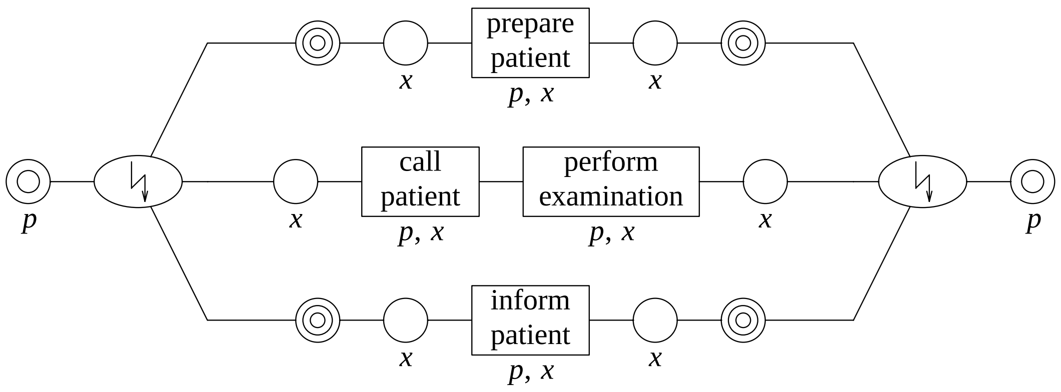

Figure 3: Integrity constraint for patients

[3] In the workflows of Fig. 1, these parameters have been omitted for the sake of simplicity. They might be considered global workflow variables which are implicitly passed to all activities of the workflow.

$\underset{p}{\circledcirc}\cdots\underset{p}{\circledcirc}$ might be traversed concurrently and independently for all possible values of the parameter p. Thus, these operators constitute generalizations of the basic “either or” (disjunction) and “as well as” (parallel composition) branchings, respectively, depicted in Fig. 4. Finally, the “arbitrarily parallel” operators $\circledcirc\!\!\!\bigcirc\cdots\circledcirc\!\!\!\bigcirc$ allow an arbitrary number of concurrent and independent traversals of their body.[4]

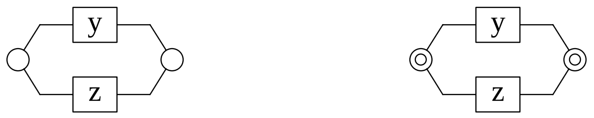

Figure 4: Basic branching operators: “either or” (left) and “as well as” (right)

User-Defined Operators

To complete the example above, Fig. 5 shows a possible definition of the mutual exclusion operator “flash” as a constant repetition (sequential iteration) of an “either or” branching containing the mutual exclusive branches x, y, and z.[5] Employing such kinds of templates does not only simplify the graphs containing them, but also raises their level of abstraction as a user of the “flash” operator does not need to know its precise definition but only its abstract meaning. Therefore, frequently occurring or fairly complicated application-specific operators might be predefined by an “interaction graph expert” and applied afterwards even by unexperienced users.

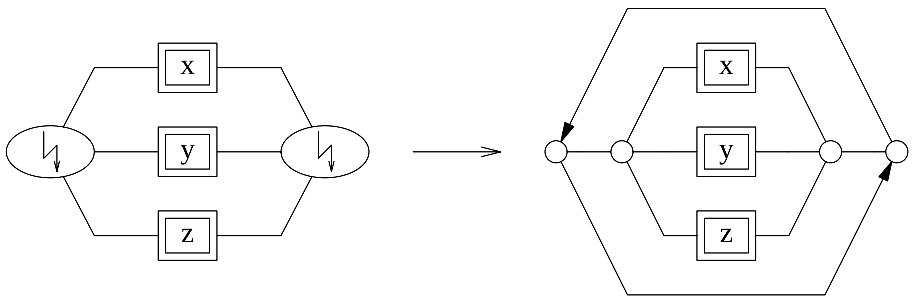

Figure 5: Definition of the mutual exclusion operator

Modular Combination of Graphs

Figure 6 shows another example of an interaction graph specifying a generic *capacity restriction for examination departments* by describing that for each kind of examination x (quantifier $\underset{x}{\circledcirc}\cdots\underset{x}{\circledcirc}$) three concurrent and independent instances (multiplier $\overset{3}{\circledcirc}\cdots\overset{3}{\circledcirc}$) of the sequence *call* – *perform* might be executed repeatedly (sequential iteration $\circ\cdots\circ$). Each of these sequences might be traversed with an arbitrary patient p (quantifier $\underset{p}{\bigcirc}\cdots\underset{p}{\bigcirc}$). This means effectively, that each examination department x can treat at most three patients p simultaneously.

[4] As a mnemonic aid, a single circle (whether small or large) expresses that *one* branch must be chosen, while a double circle requires *both* or *all* branches to be traversed. Finally, three circles represent an *arbitrary* number of parallel traversals.
[5] It is also possible to give a more general definition where the number of branches is variable.

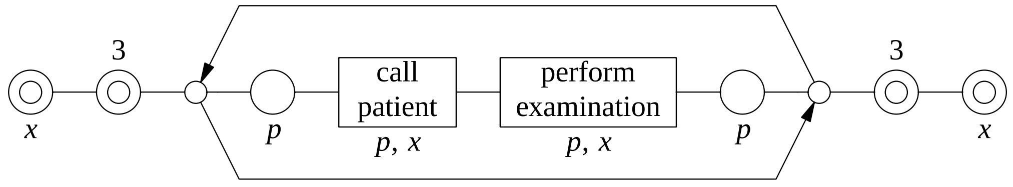

Figure 6: Capacity restriction for examination departments

Having specified separate synchronization conditions for patients (Fig. 3) and examination departments (Fig. 6), a coupling operator is employed to combine these independently developed subgraphs into a single interaction graph representing their semantic conjunction (cf. Fig. 7). More precisely, the combined graph permits the execution of a particular activity if and only if it is permitted by *all subgraphs containing this activity*. Applied to the graph of Fig. 7 this means that the execution of *call* and *perform* is permitted if and only if it is permitted by both branches of the coupling operator **o**···**o**, while *prepare* and *inform* are permitted as soon as they are permitted by the upper branch. In contrast to a strict conjunction operator (denoted ●···●) which permits execution of an activity if and only if it is permitted by *all* its branches, the more loosely coupling employed in Fig. 7 is usually much more intuitive and useful in practice as a subgraph should not prohibit the execution of activities which it does not explicitly mention. This kind of open-world assumption − there might be activities which are either unknown or irrelevant at the time a graph is developed − supports a modular development of small interaction graphs describing particular aspects or facets of a synchronization condition and their seamless integration into larger graphs afterwards. In contrast, formalisms providing strict conjunction only [10] or no explicit conjunction operator at all [22, 23] force graph developers to augment the individually developed subgraphs with auxiliary branches or special synchronization symbols before combining them to larger graphs [12].

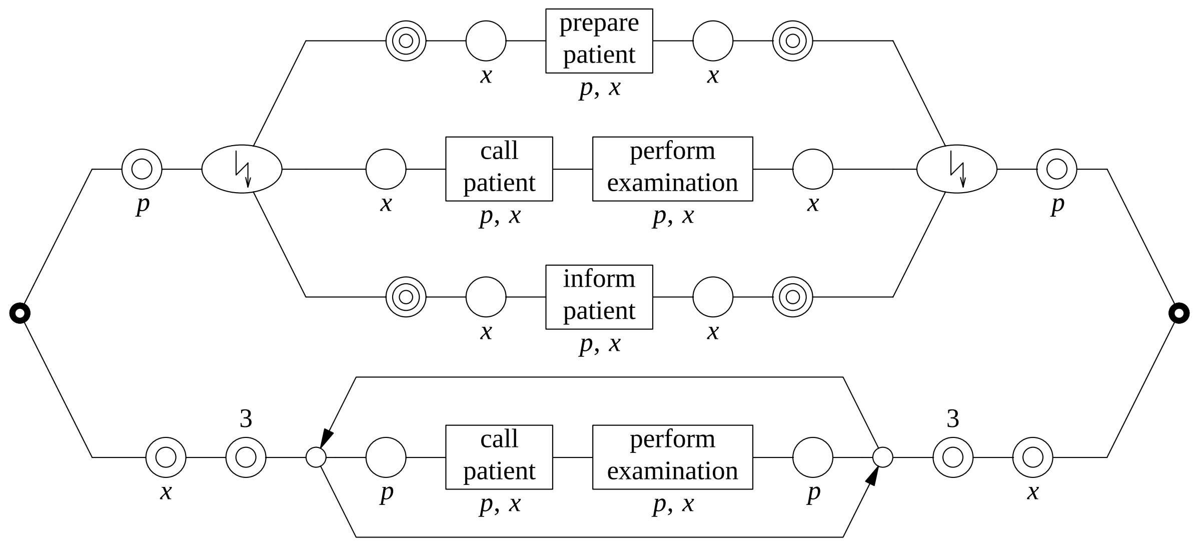

Figure 7: Coupling of independently developed subgraphs

3. Interaction Expressions

Formal Semantics

For normal applications, the meaning of an interaction graph, i. e., its set of permissible execution sequences, is intuitively derived by traversing the graph from left to right according to descriptive rules and recording the visited actions.[6] Formally, such a sequence of actions is called a *complete word* of the graph if the traversal is complete, i. e., reaches the rightmost end of the graph. Otherwise, if the traversal is terminated prematurely, the resulting sequence is called a *partial word*. In order to precisely determine the semantics of an interaction graph x, the sets $\Phi(x)$ and $\Psi(x)$ containing the complete and partial words of x, respectively, will be defined in the following.[7] As it is possible − typically by misusing the coupling operator − to construct graphs with "dead ends," i. e., graphs possessing partial but no complete words, partial words cannot be simply derived as prefixes of complete words but have to be defined separately. In order to simplify notations, interaction *expressions* are introduced in the following as an equivalent formal notation of interaction graphs. Expressed the other way round, interaction graphs are merely a graphical notation of interaction expressions just like syntax charts constitute a graphical representation of context-free grammars.

Table 8 summarizes the definition of interaction expressions x (first and second column) where y and z constitute recursively defined subexpressions and a represents an *abstract action*

$$a \in \Gamma = \{\, [a_0, a_1, \ldots, a_n] \mid n \in \mathbb{N}_0,\ a_0 \in \Lambda,\ a_1, \ldots, a_n \in \Omega \cup \Pi \,\}$$

consisting of an action *name* $a_0 \in \Lambda$ and zero or more *arguments* $a_1, \ldots, a_n \in \Omega \cup \Pi$ which are either concrete *values* $\omega \in \Omega$ or formal *parameters* $p \in \Pi$. Here, Λ, Ω, and Π denote corresponding basic sets for which the conditions $\Omega \cap \Pi = \varnothing$ and $|\Omega| = \infty$ shall hold. Furthermore, for each

Category	x	$\Phi(x)$	$\Psi(x)$	$\alpha(x)$
atomic expression option	a $\curvearrowright y$	$\{\langle a\rangle\} \cap \Sigma^*$ $\Phi(y) \cup \{\langle\rangle\}$	$\{\langle\rangle, \langle a\rangle\} \cap \Sigma^*$ $\Psi(y)$	$\{a\}$ $\alpha(y)$
sequential composition sequential iteration	$y - z$ $\ominus y$	$\Phi(y)\,\Phi(z)$ $\Phi(y)^*$	$\Psi(y) \cup \Phi(y)\,\Psi(z)$ $\Phi(y)^*\,\Psi(y)$	$\alpha(y) \cup \alpha(z)$ $\alpha(y)$
parallel composition parallel iteration	$y \circledcirc z$ $\circledcirc y$	$\Phi(y) \otimes \Phi(z)$ $\Phi(y)\#$	$\Psi(y) \otimes \Psi(z)$ $\Psi(y)\#$	$\alpha(y) \cup \alpha(z)$ $\alpha(y)$
disjunction conjunction	$y \circ z$ $y \bullet z$	$\Phi(y) \cup \Phi(z)$ $\Phi(y) \cap \Phi(z)$	$\Psi(y) \cup \Psi(z)$ $\Psi(y) \cap \Psi(z)$	$\alpha(y) \cup \alpha(z)$ $\alpha(y) \cup \alpha(z)$
synchronization	$y \boldsymbol{\circ} z$	$\Phi(y) \otimes \kappa_x(y)^* \cap$ $\Phi(z) \otimes \kappa_x(z)^*$	$\Psi(y) \otimes \kappa_x(y)^* \cap$ $\Psi(z) \otimes \kappa_x(z)^*$	$\alpha(y) \cup \alpha(z)$
disjunction quantifier	$\underset{p}{\bigcirc}\, y$	$\bigcup\limits_{\omega\in\Omega} \Phi\left(y_p^\omega\right)$	$\bigcup\limits_{\omega\in\Omega} \Psi\left(y_p^\omega\right)$	$\bigcup\limits_{\omega\in\Omega} \alpha\left(y_p^\omega\right)$
parallel quantifier	$\underset{p}{\circledcirc}\, y$	$\bigotimes\limits_{\omega\in\Omega} \Phi\left(y_p^\omega\right)$	$\bigotimes\limits_{\omega\in\Omega} \Psi\left(y_p^\omega\right)$	$\bigcup\limits_{\omega\in\Omega} \alpha\left(y_p^\omega\right)$
synchr. quantifier	$\underset{p}{\boldsymbol{\bigcirc}}\, y$	$\bigcap\limits_{\omega\in\Omega} \Phi\left(y_p^\omega\right) \otimes \kappa_x\left(y_p^\omega\right)^*$	$\bigcap\limits_{\omega\in\Omega} \Psi\left(y_p^\omega\right) \otimes \kappa_x\left(y_p^\omega\right)^*$	$\bigcup\limits_{\omega\in\Omega} \alpha\left(y_p^\omega\right)$
conjunction quantifier	$\underset{p}{\large\bullet}\, y$	$\bigcap\limits_{\omega\in\Omega} \Phi\left(y_p^\omega\right)$	$\bigcap\limits_{\omega\in\Omega} \Psi\left(y_p^\omega\right)$	$\bigcup\limits_{\omega\in\Omega} \alpha\left(y_p^\omega\right)$

Table 8: Formal semantics of interaction expressions

[6] The rectangular nodes of an interaction graph represent so called *activities* possessing a positive duration in time. In contrast, *actions* correspond to points in time without any duration. As the exact duration of an activity A is irrelevant, it is implicitly mapped to a sequence of two actions, A_S and A_T, representing the start and termination of A, respectively.
[7] As another mnemonic aid, Φ (pronounced **f**i) contains "**f**inal" or complete words, whereas Ψ (**p**si) contains **p**artial words.

expression x, Tab. 8 defines its set of complete and partial words (third and fourth column, respectively) where brackets $\langle \dots \rangle$ are used to denote *abstract words* $\in \Gamma^*$ and Σ represents the set of *concrete actions*,

$$\Sigma = \{\, [a_0, a_1, \dots, a_n] \mid n \in \mathbb{N}_0,\ a_0 \in \Lambda,\ a_1, \dots, a_n \in \Omega \,\},$$

whose arguments a_i are all concrete values. Consequently, a *concrete word* $w \in \Sigma^*$ corresponds to a sequence of concrete actions executed in the real world.

The concatenation ($U\,V$) and Kleene closure (U^*) of languages $U, V \subseteq \Sigma^*$ is defined as usual, whereas the *shuffle* of words $u, v \in \Sigma^*$ and languages $U, V \subseteq \Sigma^*$ as well as the corresponding closure is defined as follows [23, 10]:[8]

$$u \otimes v = \{\, u_1\, v_1 \dots u_n\, v_n \mid n \in \mathbb{N},\ u_1, v_1, \dots, u_n, v_n \in \Sigma^*,\ u_1 \dots u_n = u,\ v_1 \dots v_n = v \,\},$$

$$U \otimes V = \bigcup_{\substack{u \in U \\ v \in V}} u \otimes v = \{\, w \in \Sigma^* \mid \exists\, u \in U, v \in V \colon w \in u \otimes v \,\}.$$

$$\bigotimes_{i=1}^{n} U_i = \begin{cases} \{\langle\rangle\} & \text{for } n = 0, \\ \left(\bigotimes_{i=1}^{n-1} U_i\right) \otimes U_n & \text{for } n > 0, \end{cases} \qquad U\# = \bigcup_{n=0}^{\infty} \bigotimes_{i=1}^{n} U = \bigcup_{\substack{n \in \mathbb{N}_0 \\ u_1, \dots, u_n \in U}} u_1 \otimes \dots \otimes u_n.$$

For an expression y, a parameter $p \in \Pi$, and a value $\omega \in \Omega$, y_p^ω denotes the expression derived from y by replacing every occurrence of the parameter p with the value ω. Infinite unions and intersections are defined as usual, whereas the shuffle of infinitely many languages $U_\omega \in \Sigma^*$ ($\omega \in \Omega$) is either empty or can be reduced to a union of finite shuffles if all participants U_ω contain the empty word [12]:

$$\bigotimes_{\omega \in \Omega} U_\omega = \begin{cases} \displaystyle\bigcup_{\substack{n \in \mathbb{N} \\ \omega_1 \neq \dots \neq \omega_n \in \Omega}} \bigotimes_{i=1}^{n} U_{\omega_i}, & \text{if } \langle\rangle \in U_\omega \text{ for all } \omega \in \Omega, \\ \varnothing & \text{otherwise.} \end{cases}$$

Finally, Tab. 8 defines the *alphabet* $\alpha(x)$ of expressions x (last column) which is needed for the definition of the *alphabet complement* $\kappa_x(y) = \alpha(x) \setminus \alpha(y)$. Unfortunately, space does not permit a more detailed motivation and explanation of the definitions given in this section.

Properties of Interaction Expressions

Based on the definitions of Tab. 8, two interaction expressions x_1 and x_2 are considered equal or *equivalent*, if they possess the same alphabet and accept the same complete and partial words.[9] Given this equivalence relation, numerous useful properties of interaction expressions, like commutativity, associativity, or idempotence of operators, which are intuitively evident, can be formally proven [12].

Furthermore, interaction expressions can be compared with well-known formalisms like regular expressions and context-free grammars regarding their expressiveness. While it is obvious that interaction expressions are more expressive than regular expressions, their relation to context-free grammars is not yet exactly determined. On the one hand, there are expressions, e. g., $x = (\ominus\, a - \ominus\, b - \ominus\, c) \bullet \circledcirc\, (a - b - c)$, whose language, $\Phi(x) = \{\, \langle a^n, b^n, c^n \rangle \mid n \in \mathbb{N}_0 \,\}$, is not context-free. On the other hand, there are context-free grammars specifying, e. g., palindromes, whose language is *presumably* not expressible with interaction expressions as they − deliberately − do not allow recursive expressions. As these questions are of little relevance for practical applications of interaction expressions, they have not been investigated in more detail.

[8] Note that, in the first definition, u_i and v_i do not represent actions $\in \Sigma$, but *subwords* $\in \Sigma^*$ consisting of zero or more actions.

[9] More precisely, as x_1 and x_2 might contain unbound parameters, every pair of *concretions* $(x_1)_{p_1, \dots, p_k}^{\omega_1, \dots, \omega_k}$ and $(x_2)_{p_1, \dots, p_k}^{\omega_1, \dots, \omega_k}$ (for arbitrary parameters $p_1, \dots, p_k \in \Pi$ and values $\omega_1, \dots, \omega_k \in \Omega$) must accept the same complete and partial words.

4. Operational Semantics of Interaction Expressions

State Model

Given the formal semantics of interaction expressions, it is possible in principal to construct an algorithm solving the *word problem* − given an interaction expression x and a concrete word w, decide whether w is a partial or complete word of x − by more or less directly transforming the definitions of $\Psi(x)$ and $\Phi(x)$ into executable code. The problem with this algorithm is, however, that it is hopelessly inefficient as its complexity grows exponentially with respect to the length of the word w even for very simple expressions x [24, 12]. In order to obtain a more efficient and practically useful implementation of interaction expressions, it is thus necessary to introduce an operational *state model* comparable in some sense to finite state machines typically used for the implementation of regular expressions.

For that purpose, every interaction expression x is assigned an *initial state* $\sigma(x)$ where a state might be a complex, hierarchically structured mathematical object. Furthermore, a *state transition* function τ is defined which maps a state s and an action a to a successor state $s' = \tau_a(s)$. Finally, two *state predicates*, $\psi(s)$ and $\varphi(s)$, are introduced which correspond directly to the sets $\Psi(x)$ and $\Phi(x)$ of the formal semantics (cf. below). The nature of these definitions allows them to be transformed to executable program code quite directly. To further improve the efficiency of the so constructed implementation, an *equivalence relation* is introduced for states based on the predicates ψ and φ and an *optimization function* ρ is defined which maps some states s to equivalent, but less complex states $\hat{s} = \rho(s)$ which can be processed more efficiently.

Intuitively, the state model formalizes the descriptive idea of traversing an interaction graph. That means, the initial state $\sigma(x)$ of an expression x describes the *starting position* of a walker (or a group of walkers) who wants to walk through the corresponding interaction graph, while a state transition $\tau_a(s)$ represents the *traversal* of an action a. A successor state $\sigma_w(x)$, derived from the initial state $\sigma(x)$ by applying a sequence of state transitions corresponding to a word w, describes the set of *all possible positions* the walker(s) *might* have reached after traversing the sequence of actions w. Such a state is said to be *valid*, which is equivalent to its predicate ψ being true, if the sequence w is permissible, i. e., constitutes a partial word of x. It is called a *final* state, which is equivalent to its predicate φ being true, if the walker(s) might have reached the end of the graph after traversing the actions of w.

To actually guarantee the correctness of the state model with respect to the formal semantics of interaction expressions, the following equivalences must hold for every word $w \in \Sigma^*$:

$$w \in \Psi(x) \Leftrightarrow \psi(\sigma_w(x)) = \text{true} \quad \text{and} \quad w \in \Phi(x) \Leftrightarrow \varphi(\sigma_w(x)) = \text{true}.$$

The corresponding proof constitutes a very large structural induction using several smaller computational inductions (verifying properties of the states $\sigma_w(x)$ for the different categories of expressions x) as lemmas. Furthermore, an auxiliary theorem must be proven in parallel to make sure that quantifier expressions, though constituting conceptually infinite expressions, can nevertheless be implemented using finite states [12].

Example

As space does not permit to present the definitions of the state model in full detail, a single example should suffice to give the reader a “taste” of their nature. The states of a parallel composition $x = y \circledcirc z$ are tuples $[\circledcirc, A]$ consisting of the parallel composition operator $\circledcirc$ and a set A of *alternatives* describing possible positions of walkers in the graph corresponding to x. Each alternative constitutes a pair of substates $[l, r]$, where l and r represent states of the left and right operands y and z of the expression x, respectively, describing in turn possible positions of walkers in the corresponding subgraphs.

The initial state of x consists of a single alternative containing the initial states of the subexpressions y and z:

$$\sigma(x) = [\circledcirc, A] \quad \text{where} \quad A = \{\, [\sigma(y), \sigma(z)] \,\}.$$

As the branches of a parallel composition are executed concurrently and independently, the traversal of an action $a \in \Sigma$ in x might be performed in either branch, y or z. That means, that a state transition $\tau_a(s)$ of a state $s = [\circledcirc, A]$ should replace each alternative $[l, r] \in A$ with two transformed alternatives $[l', r]$ and $[l, r']$ where $l' = \tau_a(l)$ and $r' = \tau_a(r)$ represent the corresponding successor states of l and r, respectively:

$$\tau_a(s) = [\circledcirc, A'] \quad \text{where} \quad A' = \{\, [\tau_a(l), r] \mid [l, r] \in A \,\} \cup \{\, [l, \tau_a(r)] \mid [l, r] \in A \,\}.$$

A state $s = [\circledcirc, A]$ should be considered valid or final, if and only if it contains an alternative $[l, r] \in A$ where both substates l and r are valid or final, respectively:

$$\psi(s) = \bigvee_{[l, r] \in A} (\psi(l) \wedge \psi(r)), \qquad \varphi(s) = \bigvee_{[l, r] \in A} (\varphi(l) \wedge \varphi(r)).$$

Finally, a state $s = [\circledcirc, A]$ might be optimized by removing alternatives containing invalid substates as these do not represent reasonable positions of walkers in the graph:

$$\rho(s) = [\circledcirc, \hat{A}] \quad \text{where} \quad \hat{A} = \{\, [l, r] \in A \mid \psi(l) \wedge \psi(r) \,\}.$$

5. Implementation of Interaction Expressions

Implementation of the State Model

As already mentioned in Sec. 4, the nature of the definitions of the functions σ, τ, ψ, φ, and ρ allows to transform them to executable program code quite directly. It turns out, however, that the state predicate ψ is dispensible if invalid states are already recognized by the optimization function ρ and mapped to a special null state. Furthermore, as the state transition function τ and the optimization function ρ are always applied successively, it makes sense to combine them into a single optimized state transition function $\hat{\tau}_a(s) = \rho(\tau_a(s))$. The remaining functions, $\sigma(x)$, $\hat{\tau}_a(s)$, and $\varphi(s)$, can be readily implemented using any suitable programming language.

Solution of the Word and Action Problems

Assuming corresponding function implementations `init()`, `trans()`, and `final()` in C++, it is easily possible to implement top level functions `word()` and `action()` solving the following problems (cf. Fig. 9):

1. The function `word()` solves the *word problem*, i. e., it decides whether a sequence `w` of `n` actions is a complete, partial, or illegal word of an interaction expression `x` and returns a corresponding integer value. For that purpose, the initial state `s` of `x` is computed (function `init()`) and successively transformed using the actions `w[i]` of `w` (function `trans()`). If the resulting state `s` is a final state (function `final()`), `w` constitutes a complete word of `x`; otherwise, if `s` is valid (i. e., different from the null state), `w` is a partial word of `x`; otherwise, `w` is illegal.

2. The function `action()` solves the so called *action problem*. After computing the initial state `s` of the expression `x`, it successively reads actions `a` (function `ReadNextAction()`) and decides whether each such action is *currently permissible*. For that purpose, a “tentative” state transition is performed to check whether the successive state `t` is valid. If it is, `a` is accepted and the state transition is actually performed by replacing the current state `s` with the successor state `t`. Otherwise, `a` is rejected and the current state `s` remains unchanged.

As will be explained in Sec. 7, solving the action problem is highly relevant for practical applications of interaction expressions, while the solution of the word problem is more or less a by-product of primarily theoretical interest.

```
// Functions implementing the state model.
State init(Expr x);            // Return the initial state of expression x.
State trans(State s, Action a); // Perform an optimized state transition
                                // of state s with action a.
bool final(State s);           // Determine whether s is a final state.

// Function to solve the word problem.
int word(Expr x, Action* w, int n) {
    State s = init(x);
    for (int i = 0; i < n; i++) s = trans(s, w[i]);

    if (final(s)) return 2;     // Complete word.
    else if (s) return 1;       // Partial word.
    else return 0;              // Illegal word.
}

// Function to solve the action problem.
void action(Expr x) {
    State s = init(x);
    while (true) {
        Action a = ReadNextAction();
        if (State t = trans(s, a)) { printf("Accept.\n"); s = t; }
        else printf("Reject.\n");
    }
}
```

Figure 9: Solution of the word and action problems

6. Complexity of Interaction Expressions

Despite the fact, that statements about the computational complexity of interaction expressions are highly relevant for practical applications, space does not permit to treat this topic in much detail. Nevertheless, the main results providing the basis for a successful practical employment shall be briefly presented.

Generally, there is a good news and a bad news about the complexity of interaction expressions. The bad news is, it is possible to construct "malignant" expressions, i. e., expressions for which the complexity of a state transition (in the current implementation) grows exponentially with respect to the length of the action sequence processed so far. The good news is, those expressions do not seem to occur in practical applications. In order to substantiate this admittedly vague statement, extensive and detailed analyses about the growth and evolution of states of an expression have been carried out. For example, the size of a parallel composition state $s = [\circledcirc, A]$, i. e., the cardinality of the set of alternatives A, *potentially* grows by a factor of two for each state transition (cf. Sec. 4). In practice, however, the transformed alternatives $[l', r]$ and $[l, r']$ often contain invalid substates l' or r' causing them to get immediately removed by the subsequently applied optimization function ρ. Therefore, the cardinality of A and thus the complexity of subsequent state transitions remains nearly constant for many practical examples.

To obtain more precise propositions about the actual behaviour of expressions, several useful subclasses of interaction expressions have been identified, e. g., quasi-regular expressions, completely and uniformly quantified expressions, etc., for which detailed criterions for their "benignity" have been elaborated. For example, it can be shown that quasi-regular expressions (i. e., expressions not containing parallel iterations or quantifiers) are "harmless" (the complexity of a state transition remains constant) and that completely and uniformly quantified expressions (which constitute the normal case of quantified expressions in practice) are "benign" (the complexity of a state transition grows polynomially with respect to the length of the action sequence processed so far). Furthermore, these propositions can be used in combination to evaluate step by step that a given expression is benign.

To put it in a nutshell, all practical examples considered so far − including those presented in this paper − have been formally proven benign using the complexity propositions developed in [12]. Furthermore, the actual degree of the polynomial growth is rarely greater than 1 or 2. On the other hand, malignant expressions − including a suitable word for which they actually behave malignant − have to be selectively constructed and do not seem to have any practical relevance.

7. Integration with Workflow Management Systems

Coordination and Subscription Protocols

Having designed interaction expressions and graphs (Sec. 2), defined their formal semantics (Sec. 3), developed, verified, and implemented an equivalent operational semantics (Secs. 4 and 5), and finally proved its efficiency for practically relevant expressions (Sec. 6), the question arises how interaction expressions and graphs can actually be employed to synchronize the execution of real-world activities. Assuming that these activities will be executed by some sort of *interaction clients* (typically workflow management systems), a central scheduler or *interaction manager* and a suitable *coordination protocol* is needed to monitor and control the execution of actions (cf. Fig. 10, left side).

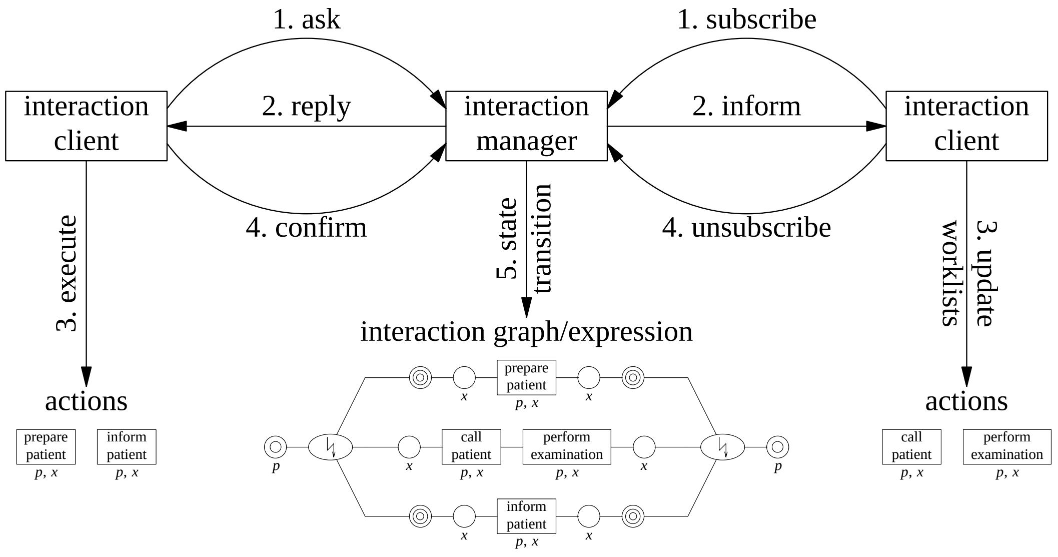

Figure 10: Coordination and subscription protocols

To make sure that a client does not execute an action which is currently not permitted by the given interaction graph, the client has to *ask* the interaction manager for permission first (step 1). Depending on the current state of the graph, the interaction manager *replies* either yes or no (step 2). If a positive answer is received, the client actually *executes* the respective action (step 3) and *confirms* its execution (step 4) causing the interaction manager to perform a corresponding *state transition* of the graph (step 5). Otherwise, the client must refrain from executing the action now and try again later.

In order to avoid busy waiting in that case causing unnecessary communication and interaction manager workload, a client can *subscribe* to a particular action (step 1, right side of Fig. 10) causing the interaction manager to *inform* him about every *status change* of the respective action (step 2), i. e., the client receives informational messages whenever the status of a subscribed action changes from permissible to non-permissible or vice versa. These messages can be used on the one hand to keep users’ worklists up to date (step 3) and on the other hand to wait passively for the right moment to ask again for permission to execute an action. Finally, if a client is no longer interested in the status of an action, a corresponding *unsubscribe* message (step 4) tells the interaction manager to stop sending informations about this action.

As space does not permit to discuss these protocols in more detail, the interested reader is referred to [12], where several alternative coordination protocols, possessing different complexity and particular advantages and disadvantages, are presented and − to avoid the interaction manager to become a bottleneck − generalized to application scenarios involving multiple interaction managers. Furthermore, the employment of persistent message queues [1] for the communication between interaction manager and clients as well as recovery strategies of the interaction manager are described.

Adaptation of Worklist Handlers versus Workflow Engines

In order to force a WfMS, which is per se not designed to ask anybody else for permission before executing an activity, to participate in a coordination protocol, two alternative strategies can be pursued possessing different advantages and disadvantages. As the runtime component of a WfMS basically consists of a workflow engine communicating with several worklist handlers via the WfMS’s API, either the workflow engine or the individual worklist handlers can be adapted to become interaction clients participating in a coordination protocol with an interaction manager (cf. Fig. 11).

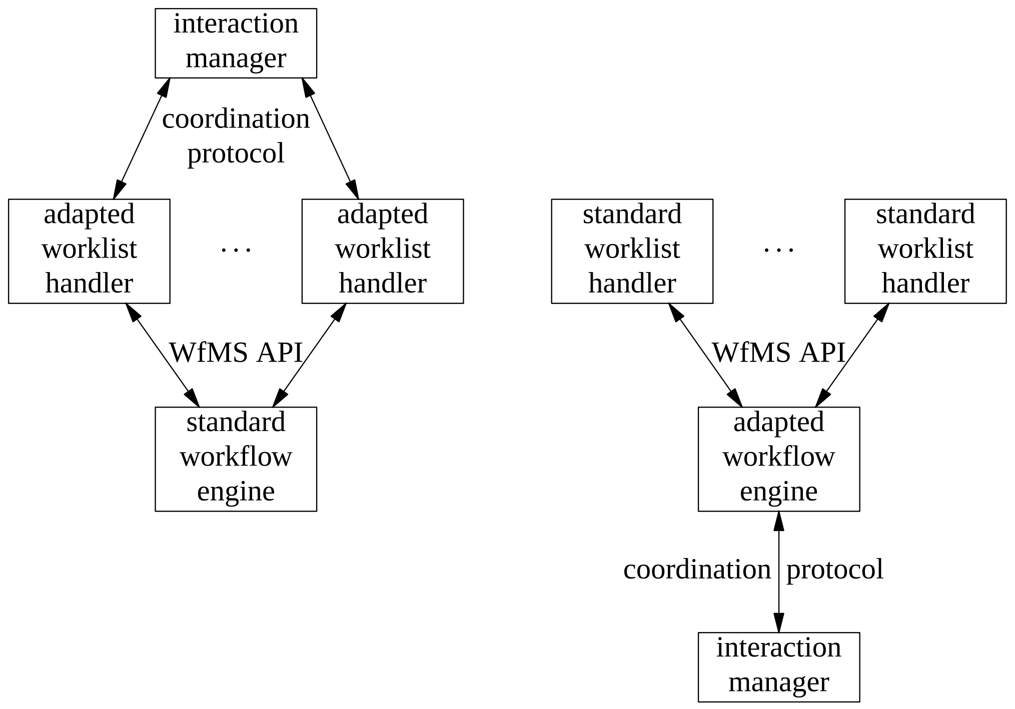

Figure 11: Adaptation of worklist handlers (left) versus workflow engines (right)

As the WfMS’s API is either standardized by the Workflow Management Coalition (WfMC) or at least documented by the vendor, it is common practice to replace the standard worklist handlers of a WfMS with customized implementations fitting better into the overall appearance of users’ desktop environments or the like. Under these circumstances, it takes little extra effort to incorporate a coordination protocol into such a customized worklist handler implementation causing it to become a central mediator between workflow engine and interaction manager (left side of Fig. 11). In this scenario, an adapted worklist handler offers and executes only those activities which are regularly scheduled by the workflow engine *and* currently permitted by the interaction manager, whereas the workflow engine remains completely unchanged and does not even know of the interaction manager's existence.

Although this solution is rather easy to realize, it has several drawbacks in practice. First of all, as every worklist handler has to communicate with the interaction manager, it introduces substantial communication overhead. Secondly, as the workflow engine is not involved in the coordination protocol, it might happen that activities will be executed accidentally through a standard worklist handler of the WfMS, i. e., the approach is not completely "waterproof." A third problem arises from the fact that worklist handlers usually run on users' desktop computers which are rather unreliable. If, for instance, a user switches off his PC while the worklist handler performs step 3 of the coordination protocol (cf. Fig. 10), the interaction manager waits in vain for the confirmation in step 4 causing him to remain stuck in a critical region comprising steps 2 to 5. The only way to alleviate this problem is to use a more complicated coordination protocol inducing even higher communication overhead [12].

In order to remedy all these shortcomings, it is necessary to incorporate the coordination protocol directly into the workflow engine (cf. Fig. 11, right side). In that case, the adapted workflow engine offers and executes only those activities which are regularly scheduled according to the respective workflow definitions *and* currently permitted by the interaction manager, whereas the worklist handlers are completely unaffected. (Of course, it is possible to use customized implementations anyway.) Though absolutely preferable in principal, this solution requires substantially more design and implementation effort as additional functionality has to be incorporated into an already fairly complex software system, viz a workflow engine. Furthermore, this solution can only be realized by the WfMS vendor possessing the workflow engine's source code and documentation, while the adaptation of worklist handlers is realizable by customers, too.

8. Conclusion

Interaction expressions and graphs constitute a flexible and expressive formalism for the specification and implementation of synchronization conditions in general and inter-workflow dependencies in particular. In addition to a declarative semi-formal interpretation (traversing interaction graphs), a precise formal semantics, an equivalent operational semantics, an efficient implementation of the latter, and detailed complexity analyses have been developed allowing the formalism to be actually applied to solve real-world problems like inter-workflow coordination. In contrast to other formalisms based on extended regular expressions, interaction expressions are conceptually comprehensive and completely orthogonal. Compared to other well-known approaches for the specification of concurrent systems, especially Petri nets [21, 16] and various kinds of process algebras [13, 14, 20], their behaviour is fully deterministic, even though this heavily complicates their operational semantics and implementation [12].

Despite the fact that inter-workflow dependencies occur frequently in practical applications, they have not received much attention in the workflow community yet. Neither special issues of journals devoted to the overall topic of workflow management [5, 6, 7, 8, 17] nor books reflecting the state of the art in that field [26, 15] have really addressed the problem so far. The same holds for conference and workshop proceedings in general where the number of papers dealing with other workflow management problems, e. g., flexibility and scalability, is steadily increasing. Two noteable exceptions are [3] and [18]. Both approaches, however, are not able in principal to deal with dynamically evolving workflow ensembles whose participants are not known in advance and might change with time. Therefore, the thorough development of interaction expressions and graphs and their application to coordinate dynamically evolving workflow ensembles constitutes a pioneering approach towards a general solution of the inter-workflow coordination problem.

In addition to a very mature core implementation of interaction expressions based on the formally verified operational semantics (cf. Sec. 5), a syntax-driven editor for interaction graphs has been developed to facilitate their creation in practice. Furthermore, the coordination and subscription protocols described in Sec. 7 have been prototypically implemented and tested for the WfMS ProMInanD [19]. Their integration into the next generation WfMS ADEPT [4] is a topic of future work.

References

[1] P. A. Bernstein, M. Hsu, B. Mann: “Implementing Recoverable Requests Using Queues.” In: *Proc. ACM SIGMOD Int. Conf. on Management of Data*. 1990, 112−122.
[2] R. H. Campbell, A. N. Habermann: “The Specification of Process Synchronization by Path Expressions.” In: E. Gelenbe, C. Kaiser (eds.): *Operating Systems*. Lecture Notes in Computer Science 16, Springer-Verlag, Berlin, 1974, 89−102.
[3] 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*. Lecture Notes in Computer Science 1057, Springer-Verlag, Berlin, 1996, 443−462.
[4] P. Dadam, M. Reichert: “The ADEPT WfMS Project at the University of Ulm.” In: *1st European Workshop on Workflow and Process Management* (Zürich, Switzerland, 1998). 1998.
[5] Special Issue on Workflow and Extended Transaction Systems. *IEEE Data Engineering Bulletin* 16 (2) June 1993.
[6] Special Issue on Workflow Systems. *IEEE Data Engineering Bulletin* 18 (1) March 1995.
[7] Special Issue on Software Support for Work Flow Management. *Distributed and Parallel Databases* 3 (2) April 1995.
[8] Special Issue on Workflow Systems. *Distributed Systems Engineering Journal* 3 (4) December 1996.
[9] F. J. Faase, S. J. Even, R. A. de By: *Introduction to CoCoA* (TransCoop Deliverable IV.3). Technical Report TC/REP/UT/D4-3/033, University of Twente, The Netherlands, February 1996.
[10] L. Guo, K. Salomaa, S. Yu: “On Synchronization Languages.” *Fundamenta Informaticae* 25 (3+4) March 1996, 423−436.
[11] C. Heinlein, P. Dadam: *Interaction Expressions − A Powerful Formalism for Describing Inter-Workflow Dependencies*. UIB 97-04, Fakultät für Informatik, Universität Ulm, February 1997.
[12] C. Heinlein: *Workflow and Process Synchronization with Interaction Expressions and Graphs*. Ph. D. Thesis (in German), Fakultät für Informatik, Universität Ulm, 2000.
[13] M. Hennessy: *Algebraic Theory of Processes*. The MIT Press, Cambridge, MA, 1988.
[14] C. A. R. Hoare: *Communicating Sequential Processes*. Prentice-Hall, London, 1985.
[15] S. Jablonski, M. Böhm, W. Schulze (eds.): *Workflow-Management. Entwicklung von Anwendungen und Systemen*. dpunkt-Verlag, Heidelberg, 1997.
[16] K. Jensen, G. Rozenberg (eds.): *High-Level Petri Nets. Theory and Application*. Springer-Verlag, 1991.
[17] Special Issue on Workflow and Process Management. *Journal of Intelligent Information Systems. Integrating Artificial Intelligence and Database Technologies* 10 (2) March 1998.
[18] M. Kamath, K. Ramamritham: “Failure Handling and Coordinated Execution of Concurrent Workflows.” In: *Proc. 14th Int. Conf. on Data Engineering (ICDE)* (Orlando, FL, February 1998). IEEE Computer Society, 1998, 334−341.
[19] B. Karbe: “Flexible Vorgangssteuerung mit ProMInanD.” In: U. Hasenkamp, S. Kirn, M. Syring (eds.): *CSCW − Computer Supported Cooperative Work*. Addison-Wesley, Bonn, 1994, 117−133.
[20] R. Milner: *Communication and Concurrency*. Prentice-Hall, New York, 1989.
[21] J. L. Peterson: “Petri Nets.” *ACM Computing Surveys* 9 (3) September 1977, 223−252.
[22] W. E. Riddle: “A Method for the Description and Analysis of Complex Software Systems.” *ACM SIGPLAN Notices* 8 (9) September 1973, 133−136.
[23] A. C. Shaw: “Software Description with Flow Expressions.” *IEEE Transactions on Software Engineering* SE-4 (3) May 1978, 242−254.
[24] A. C. Shaw: “On the Specification of Graphics Command Languages and Their Processors.” In: R. A. Guedj, P. J. W. ten Hagen, F. R. A. Hopgood, H. A. Tucker, D. A. Duce (eds.): *Methodology of Interaction*. North-Holland Publishing Company, Amsterdam, 1980, 377−392.
[25] A. C. Shaw: “Software Specification Languages Based on Regular Expressions.” In: W. E. Riddle, R. E. Fairley (eds.): *Software Development Tools*. Springer-Verlag, Berlin, 1980, 148−175.
[26] G. Vossen, J. Becker (eds.): *Geschäftsprozeßmodellierung und Workflow-Management. Modelle, Methoden, Werkzeuge*. International Thomson Publishing, Bonn, 1996.