Notes About: Concepts for Variability in Processes

Frank Puhlmann
frank.puhlmann@hpi.uni-potsdam.de

December 29, 2005

Contents

1 Classifications

Process modeling can be classified using different approaches. This section currently contains classification schemes based on control flow and on classical programming approaches.

1.1 Control Flow oriented approaches

A control flow oriented classification can be divided into four types. Sometimes, combinations of those types can be useful.

Procedural Approach. The procedural or block oriented modeling approach resembles high–level programming languages. The process is modeled using process statements like `action`, `branch`, `sequence`, etc. The Business Process Execution Language for Web–Services is some kind of procedural modeling language [BEA03]. The procedural approach is intuitive applicable for programmers, but difficult to understand by business people.

Rule based Approach. The rule based control flow modeling is oriented on rule based programming languages like Prolog. The model uses logical statements to determine which actions can be executed at any given time. The rule based approach is very powerful, but difficult to use, because the sequence of activities is not explicitly represented.

Graph based Approach. The graph based modeling approach uses graphs to represent the relationships among activities and other resources. Examples for graph based approaches are Petri nets [Pet62], UML activity diagrams [Gro03] or BPMN [BPM04]. Graph based models are easy to create and understand. Furthermore they are close to procedural models so that in most cases a mapping is possible. However, the theoretical foundations are rather hard to handle.

Agent based Approach. The agent based process modeling uses agents (programs) that are designed to make an intelligent decision of what to do next based on their current state. Agents can be distributed and act independently of each other. Agents are given a goal that matches to a particular process or a part of it. The internal processes of an agent can be ever changing. The agent should use the current process state and environment to intelligently determine the next state. Intelligent agents are a part of the artificial intelligence (AI) research.

Combinations Different approaches can be combined. The procedural and the graph based approach are very

close, often a a procedural model can be visualized using graphs. Other combinations are possible, like combining procedural and agent based approaches. Thereby a raw high level process schema is defined using a procedural language and the parts of the process are than executed by autonomous agents.

1.2 Classical programming approaches

Because every computer program describes a process, we should take a look at the existing literature and classification of programming languages. Naturally, intersections with other classification schemes can be found. Nevertheless, maybe we could reuse the concepts and ideas for modeling our kinds of processes.

Imperative programming. The imperative programming approach is action oriented. Every computation is viewed as a sequence of actions (statements). Actions are also used to affect the sequence flow (conditions, jumps, loops). Classical imperative programming languages are Fortran, Pascal and C.

Object–Oriented programming. The object–oriented programming approach uses classes and objects as key–concepts. The approach introduced inheritance and messaging between classes and objects. Languages supporting only objects and no classes are sometimes called object languages. Instead of inheritance, those languages use delegation and cloning. Common examples of object–oriented languages are C++, Java and Smalltalk.

Functional programming. The functional programming approach is based on an expression interpreter. Expressions consist of function applied to sub–expression. Functional programming is based on recursion and uses implicit storage allocation. Examples for functional programming languages are ML, Haskell and Opal.

Logical programming. Logic programming uses relations rather than functions. Programs consist of facts and rules. The language uses the facts and rules to deduce responses to queries. The classical language is Prolog.

Concurrent programming. Concurrent programming centers on processes or tasks. It deals with the communication and synchronization between them. A language for concurrent programming is Ada.

1.3 Other approaches

This subsection covers other approaches which do not fit into earlier classifications or consist of combinations of them.

Process Algebra oriented Approach. One can use process algebra like ACP [Bas98], CCS (Milner) or CSP (Hoare) to formally describe and analyze processes. A more recent approach centering on communication and mobile systems is the π-calculus [Mil99]. The basic techniques are easy to handle, however advanced topics are rather hard if one is not a mathematician or theoretical computer scientist.

Further considerations

Further Readings

- An approach combining procedural and agent techniques to model control flow can be found in [NB02].
- The concepts of programming languages can be found in [Set96, Seb99].

2 Procedural languages

This section covers the procedural modeling approach. Procedural modeling is based on imperative programming, but can be combined with object–oriented and concurrent programming.

Processes are modeled using statements like `action`, `branch`, `sequence` or (parallel) `flow`. Those example statements are taken from BPEL4WS [BEA03].

The definition and execution of computer languages based on statements is a well studied field: compiler writing. We could use the first part of it, the generation of a syntax–tree and symbol–table. But therefor, the syntax of the process definition must be explicit and formal to allow a computerized processing.

2.1 Formal languages

The syntax of process definitions can be constructed using grammars. The syntax of a language describes its written representation, including lexical details such as keywords and punctuation marks.

Grammar. A grammar γ over a labeling alphabet L is defined as follows:

$$\gamma = (T, N, S, R) \tag{1}$$

where $T \in L$ is a set of terminal symbols (tokens), $N \in L$ is a set of non–terminal symbols with $N \cap T = \emptyset$, $S \in N$ is a start–symbol and R is a set of production rules of the form $\alpha \rightarrow \omega$, where $\alpha, \omega \in (N \cup T)^*$ with $\exists x \in \alpha : x \in N$.

A sample grammar for generating real numbers could be written as follows, where $L = \{0..9, ., realnumber, integerpart, fraction, digit\}$:

$$\begin{aligned}\gamma_1 =&(\{0..9,.\},\\ &\{realnumber, integerpart, fraction, digit\},\\ &\{realnumber\},\\ &\{realnumber \rightarrow integerpart.fraction,\\ &integerpart \rightarrow digit,\\ &integerpart \rightarrow integerpart\ \ digit,\\ &fraction \rightarrow digit,\\ &fraction \rightarrow digit\ \ fraction,\\ &digit \rightarrow 0,\\ &digit \rightarrow 1, ...\}\\ &)\end{aligned} \tag{2}$$

0..9 is a placeholder for every monadic number. Also, $digit \rightarrow 0..9$ has to be written ten times.

Language. A language $\lambda(\gamma)$ that is defined through $\gamma = (T, N, S, R)$ is a set of all sequences of terminal symbols that can be derived from S using R.

Because of the leading zeros that γ_1 can produce, we can write $\lambda(\gamma_1) \supset \mathbb{R}$. If we somehow omit leading zeros in γ_1, we could write $\lambda(\gamma_1') = \mathbb{R}$.

BNF. A grammar γ can be hardly read by humans, especially if the complexity increases. Therefore, J. Backus and P. Naur introduced a notation called BNF (Backus–Naur–Form). The notation consists of rules as follows:

$$\text{Symbol ::= right side} \tag{3}$$

Symbol is a non–terminal and right side can consist of terminal and non–terminal expressions. Multiple possible expressions at the right side can be separated using |.

The notation of γ_1 in BNF is as follows:

$$\begin{aligned} realnumber &::= integerpart.fraction\\ integerpart &::= digit \mid integerpart\ \ digit\\ fraction &::= digit \mid digit\ \ fraction\\ digit &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned} \tag{4}$$

Sometimes all non–terminals are enclosed between special symbols ($\langle nonterminal \rangle$) and sometimes all terminal symbols are quoted ("terminal").

EBNF. The extended BNF notation enhances BNF with some syntactical abbreviations. EBNF can always be transformed back to BNF, so no additional functionality is added.

The recursion $A ::= A\ \ B \mid \phi$ can now be written as $A ::= \{B\}$, where ϕ is the empty sequence. To limit the number of elements, we could write $\{...\}_{min}^{max}$, where min is the lower boundary and max is the upper boundary of the recurrences. Instead of writing $\{...\}_0^1$ we can use brackets $[...]$.

Chomsky Hierarchy. Formal grammars can be classified using the Chomsky hierarchy. The hierarchy defines four types of grammars with four types of languages. The grammar types differ in the allowed production rules.

- *Type-0* grammars define a *recursively enumerable language* with no restrictions on the production rules.
- *Type-1* grammars define a *context–sensitive language* with production rules of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ with A a non–terminal and α, β, γ strings of terminals and non–terminals. γ must be non empty.
- *Type-2* grammars define a *context–free language* with production rules of the form $A \rightarrow \gamma$ with A a non–terminal and γ as a string of terminals and non–terminals.
- *Type-3* grammars define a *regular language* with production rules restricted to $A \rightarrow aB$ or $A \rightarrow a$ with A, B as non–terminal and a as a terminal symbol.

Interesting in this context is the computational complexity of the grammars. Type-1 grammars are classified under `PSPACE`, therewith they can be solved with polynomial memory and unlimited time. Type-2 grammars are classified under `NC`, therewith they can be solved efficiently (in polylogarithmic time) on parallel computers.

Considerations on Process Definitions. Because of computational complexity, the grammars used for existing programming languages are Type-2 with additional rules. The additional rules are used for context–sensitive features like variable declarations and typing. In addition to be useful, the rules have to be univocal.

Attribute Grammars. Attribute grammars are grammars, where the non–terminal symbols are enhanced with attributes. Also, every production rule can have additional attribute rules, which are used to describe context conditions. For example, every non-terminal expression can get a type attribute. The attribute rules can then define the type compatibility like $x \in \mathbb{R} + y \in \mathbb{N} = z \in \mathbb{R}$.

2.2 Language design

The following criteria for programming language design are taken from [TS85]. This book is rather old (1985), but gives some ideas that are still useful.

It brings up the starting questions: Is it necessary to design a new language? And if yes, what is the purpose of the new language? Already twenty year ago, there existed a bunch of programming languages, which could be easier modified and adapted than recreating the wheel. However, if the adapting effort is very high or the language extensions are profoundly, than a new language should be designed. The purpose of the new language should be exactly defined.

The remainder of this subsection lists some criteria for programming language design.

Human Communication. Is the language easily write- and readable? If the language design incorporates re–use, than the readability should be rated over writeability. Furthermore, the syntax must accurately and completely reflect the semantics of the language.

Prevention and Detection of Errors. The language should permit error detection at design time. This can be achieved using extensive typing. However, sometimes untyped languages are necessary, requiring other methods for error prevention and detection.

Usability. The language design should be simple and straightforward. There should be as less ways as possible to do the same thing. The language should consist of only as few elements as needed.

Programming Effectiveness. Programming Effectiveness covers the software–engineering aspect of usability. One point is the the support for abstraction.

Compilability. Compilability requires reduced complexity and as less context–sensitive statements as possible.

Efficiency. The efficiency has to be considered in the context of the total environment. For most cases, efficiency should not be overrated but kept in mind.

Simplicitiy. Simplicity covers the restriction to the objectives of the reasons, why the new language is designed. The language should be based on a few, well designed concepts.

Further Considerations

GOTO. In the mid–80th, the GOTO statement was heavily debated. Today, in most modern programming languages it has been outdated or even omitted. Do we still need something similar like GOTO in process descriptions?

Design Domains. Different domains have to be considered during the design of a programming language. They include the micro- (tokens), expression- , data-, control- as well as the compile structures.

Further readings

- Compiler writing is a well establish area; there exist many lecture scripts about it, like [Kop, Jäg]. All cited scripts and books about compiler writing introduce formal languages in a more or less complete way.
- A good (and thin, but yet technical) introduction to compiler writing is a book from Niklaus Wirth [Wir96].
- An older but more in depth book, concerning compiler writing, is [TS85]. The language design subsection is based on it.
- An always cited reference about compiler writing is the dragon book [ASU00]; it centers on the practical aspects.
- An object–oriented language implementing activities and processes is P [P].

3 Rule based Processes

Rule based process descriptions are strongly based on logic. Wikipedia [Wik] defines logic as follows (March, 17, 2004):

Logic is the study of prescriptive systems of reasoning, that is, systems proposed as guides for how people (as well, perhaps, as other intelligent beings/machines/systems) ought to reason. Logic says which forms of inference are valid and which are not.

By using logic, we can make statements about the environment of a process and define logic rules which execute activities based on the current state of the environment.

3.1 Foundations

Different kinds of logic exist, like Aristotelian, formal mathematical, philosophical as well as propositional and predicate logic. Our point of interest is predicate logic.

Predicate logic consists of variables, objects constants, predicate constants, function constants and logical connectives. Predicate logic requires a world in which the propositions apply. We can formulate the predication that the sky is blue as:

$$is\_blue(sky) \tag{5}$$

A function in our world is:

$$eye\_color(Frank) \tag{6}$$

Equations can be formulated like this:

$$eye\_color(Frank) = eye\_color(Anke) \tag{7}$$

We can use variables to defines statements. The statement can be evaluated if we specify x:

$$Add(3, x) = 8 \tag{8}$$

Variables can also be *bound* to quantifiers. If a variable is bound, it denotes an element, which is not any further defined as that it exists, or that it can be every element of a set:

$$\exists x.(Add(3, x) = 8) \tag{9}$$

$$\forall x.(Add(3, x) = 8) \tag{10}$$

Σ_1		A	
sorts:	nat	A_{nat}	$=_{def} \mathbb{N}$
opns:	$zero :\rightarrow nat$	$zero_A$	$=_{def} 0$
	$succ : nat \rightarrow nat$	$succ_A(n)$	$=_{def} n+1$
	$add : nat\ nat \rightarrow nat$	$add_A(n,m)$	$=_{def} m+n$
rels:	$Prim\ \langle nat \rangle$	$Prim_A$	$=_{def} \{n \mid n \text{ is a prime}\}$

Table 1: A structure A for a signature Σ_1

Also, logical connectors can be used:

$$is\_blue(eye\_color(Frank)) \rightarrow is\_blond(Frank) \tag{11}$$

To give a more funded background, we now formally define logical signatures, structures, variables and formulas for predicate logic. The definitions are taken in a shortened form from [EMC+01].

Logical Signature. A logical signature is defined as a triple:

$$\Sigma = (S, OP, R) \tag{12}$$

with S as a non empty set where the elements are called *sorts*, OP as a set where the elements are called *operation–symbols*, and R as a set, where the elements are called *relation–symbols*. Every operation–symbol $f \in OP$ is declared as $f : s_1 \ldots, s_n$ where $n \in \mathbb{N}$ and $s_1, \ldots, s_n \in S$. If $n = 0$, f is called a constant–symbol, else a function–symbol. Every relation–symbol $r \in R$ is declared as $r : \langle s_1 \ldots s_n \rangle$, where $n \in \mathbb{N}^+$ and $s_1, \ldots, s_n \in S$.

Structure. A structure for a logical signature Σ, called Σ–structure is a Σ–algebra defined as a triple:

$$A = ((A_s)_{s \in S}, (f_A)_{f \in OP}, (r_A)_{r \in R}) \tag{13}$$

For every $s \in S$ is A_s a non empty set, every constant–symbol c is an element from A_s, every function–symbol f is a mapping $f_A : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s$, and every relation–symbol is r a relation $r_A \subseteq A_{s_1} \times \ldots \times A_{s_n}$.

An example of a structure A for a logical signature Σ_1 is given in table 1.

Variables. Let $X = (X_s)_{s \in S}$ a family of sets, whose elements are called variables and $s \in S : X_s \cap OP = \emptyset$. Variables are assigned using $\beta : X \rightarrow A$.

Formulas. Terms are evaluated using $xeval(\beta)$ on the terms of $T_\Sigma(X)$. If Σ is a logical signature and X is a fitting family set of variables, we can define formulas as follows:

1. If $s \in S$ and $t_1, t_2 \in T_{\Sigma,s}(X)$, so is the expression
$$t_1 = t_2 \tag{14}$$
called an *equation* over Σ and X.

2. If $r : \langle s_1 \ldots s_n \rangle$ is a relation–symbol and $t_i \in T_{\Sigma,s_i}(X)$ for $i = 1, \ldots, n$, so is the expression
$$r(t_1, \ldots t_n) \tag{15}$$
called a *predication* over Σ and X.

3. The set $Form_\Sigma(X)$ of the formulas of the first–order predicate logic over Σ and X is inductive defined as follows:

   - Every equation and every predication over Σ and X is a predicate logical formula.
   - The logical symbols $\top$ (true) and $\bot$ (false) are predicate logical formulas.
   - If φ is a predicate logical formula, then $\neg\varphi$ is a predicate logical formula.
   - If φ and ψ are predicate logical formulas, then $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $(\varphi \rightarrow \psi)$ and $(\varphi \leftrightarrow \psi)$ are also predicate logical formulas.
   - If φ is a predicate logical formula and x is a variable, then $(\forall x.\varphi)$ and $(\exists x.\varphi)$ are predicate logical formulas, too.

A formula is called atomic, iff it is an equation, a predication or one of $\top$ and $\bot$. The set of all atomic formulas over Σ and X are called $Atom_\Sigma(X)$.

Instead of writing $\neg(\varphi = \psi)$ we could write $(\varphi \neq \psi)$ and $(\neg\forall x.\varphi)$ can be written as $(\nexists x.\varphi)$.

3.2 ECA approach

The ECA approach originates from active database systems and means *Event*, *Condition*, *Action* rule [DHL90]. The event component specifies when a rule must be executed. After invoking the rule, the condition component must be checked and if it matches, the action component is executed.

Beside from active databases, the ECA approach could also be used to specify a control flow between different activities [KEP00]. The next paragraphs summarize the cited papers.

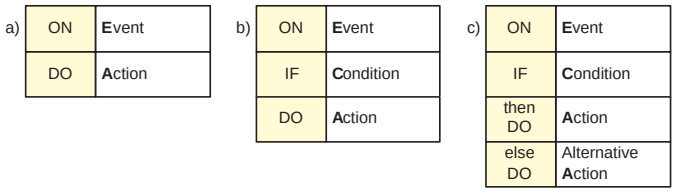

Figure 1: The EA (a), ECA (b) and ECAA (c) notation for business rules

Coupling Modes. The coupling modes originate from database transactions and specify the coupling between events and conditions (EC) or between conditions and actions (CA). Therefore, we suppose a transaction T that signals an event E for a rule R with condition C and action A. The following coupling modes can be applied:

- Immediate: C is evaluated within T immediately when E is detected (EC) or A is executed immediately after C is evaluated (CA), halting the remaining steps of T.
- Deferred: C is evaluated (EC) or A is executed (CA) after the last operation of T, but before T commits.
- Decoupled: C is evaluated (EC) or A is executed (CA) in a separate concurrently transaction T'. The dependencies between T and T' must be resolved first.

Triggers. A rule can have multiple triggers (events, E). The following triggers can be used (incomplete enumeration):

- OR–Trigger ($E_1 \vee E_2$), event E_1 or E_2 trigger the rule.
- AND–Trigger ($E_1 \wedge E_2$), event E_1 and E_2 together trigger the rule.
- Sequence–Trigger (E_1, E_2), event E_1 followed by E_2 triggers the rule.
- Counter–Trigger ($n * E$), n times the event E triggers the rule.
- m–out–of–n–Trigger, m events out of a set of events n trigger the rule.
- Periodical–Trigger, every n–th event triggers the rule.
- Interval–Trigger, every event E within an interval of events trigger the rule.

In addition, we need time triggers, that trigger an event at absolute, relative or repeated times. Finally, an event can trigger more than one rule, so that a rule priority is needed.

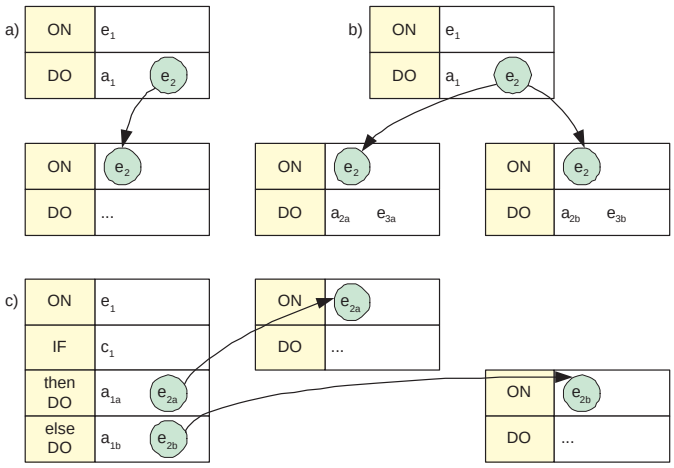

Figure 2: Rules for modeling sequential (a), parallel (b) or alternate (c) control flow

EC^{m}A^n. The EC^{m}A^n paradigm for workflows can be found in [KEP00]. It allows m conditions and n actions. Often used constructs in the workflow domain are ECAA, ECA and EA rules. They are shown in figure 1.

By using these constructs, different control flow structures like sequence, parallel or alternate execution can be defined (see figure 2). Other constructs like non–exclusive choice or repeated actions are contained in the cited paper. Actors and data can be added by defining an actor for every condition or action and data can be modeled using input and output fields for every condition or action.

Refinement. ECA business rules can refined by replacing a rule with a more complex one. This way, one can start modeling using abstract rules, which are then refined to elementary and precise rules that can be used to derive a workflow specification. The refined rules must take the same input event(s) and generate the same output event(s).

Rule Repository. A rule repository contains abstract and formalized rules. The designer must use the rule repository, the data model and the organizational model to create a process model.

3.3 Workflow Planner

This subsection is based on an internal paper by Harald Meyer and Hilmar Schuschel [MS04]. Their planner approach uses pre– and post–conditions (effects) for each activity. They described requirements for an automated planner.

A planner combines single activities to process definition by using the pre– and post–conditions. The planning algorithm is therewith a search within the rule space R^*.

An ideal planner should only find a partial order plan to allow parallel execution. They stated that the expressiveness of the modeling language is very important and defined the following requirements:

- The *logical representation* should be based on first–order predicate calculus.
- *Conditional effects* should be supported to allow easier modeling.
- *Uncertainty* must be supported. Uncertainty is something that is not known at the design–time of the process and must be decided at run–time (e.g. a data–based xor split).
- *Arithmetic functions* are needed to cover aspects like cost and durations.
- *Metrics* can be used to define optimization criteria.
- *Temporal planning* allows the optimization of the critical path length when creating parallel process flow.

Further considerations

Combinations. The ECA approach describes how a process actually works, whereas the planner approach focuses on properties for building processes based on metrics and other properties.

Further readings

- A framework for a business rule driven web service composition can be found in [OYP03].
- Concurrent Transition Logic as a foundation for workflow modeling and execution can be found in [DKRR98].

4 Set theoretic Graph Grammars

Graph grammars can be used to describe the structures of graphs. Every diagram of a visual process modeling notation like UML activity diagrams [Gro03], Petri nets [Pet62] or BPMN [BPM04] can be seen as some kind of graph.

A graph grammar defines rules for generating, transforming and testing a graph. Therewith a graph grammar can be used for generating graphs that belongs to a certain class. The other way around it can be used to test if a certain graph belongs to a class. It can also be used for transforming graph structures, e.g. for executing or optimizing a graph. By using thus possibilities, graph grammars can be used for formal describing the variability of processes.

Graph grammars can be categorized into different approaches. The one we introduce here is the set theoretic approach. The set theoretic approach is intuitive comprehensible and allows the representation of complex graphs with formal methods. Others approaches include the category theoretic and the logic oriented view. The first is based on the mathematical category theory, using total and associative catenations. The logic oriented approach uses formula sets to capture the graph as well as the graph rules and executes a rule by modifying the formula set.

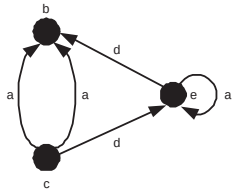

Figure 3: A directed, labeled graph

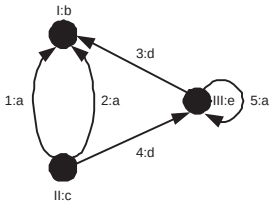

Figure 4: A directed, labeled graph with element numbering

The following considerations have been collected from different sources like lecture scripts [LMT95, SW92] because there exist a lack of textbooks on this area, as far as we know. However, there exists a handbook set [Roz97, Roz99a, Roz99b] which still must be examined. [1]

4.1 Foundations

We start with defining some essential key words for talking about graphs. The kind of graphs we use here are called *directed, labeled graphs*. A graph consists of *vertices* (V) and *edges* (E). Every edge has a source and a target vertex, making it directed. We can *label* vertices and edges by using a finite set of labels (L). An example is shown in figure 3. For better referencing parts of a graph, all elements can be *numbered*. To differ numbers from labels, we define a pair $x : y$ where x is the number and y is the label. The extended graph is shown in figure 4.

Graph. Different approaches exist to define a graph formally. We define a directed, labeled graph as a six-tuple over a labeling alphabet L:

$$G = (V, E, s, t, l, m) \tag{16}$$

where V is a set of vertices and E is a set of edges. The components s and t define a mapping $E \to V$, where each edge is assigned a source vertex (s) and a target vertex (t). The component l defines a mapping $l : V \to L$ and m defines a mapping $m : E \to L$, where each vertex and edge is assigned a label. The components of a graph G are named G_V, G_E, s^G, t^G, l^G and m^G.

[1] At a first glance, volume 1 covers basic techniques like the one described in this section. Many others like hyperedge replacement or node replacement graph grammars can also be found. Volume 3 center on interaction and distribution of graph systems. The graph

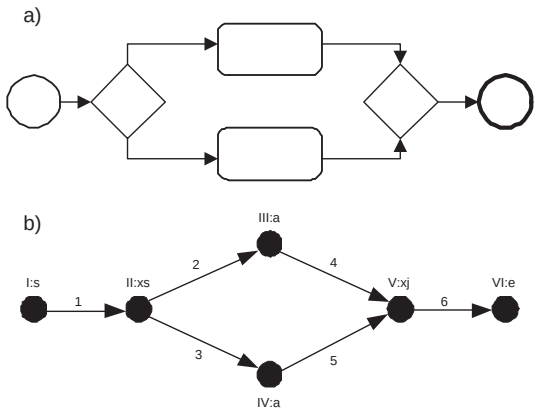

Figure 5: A BPMN diagram (a) seen as a directed, labeled graph (b)

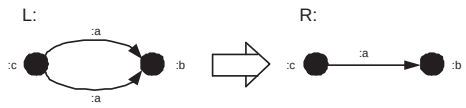

Figure 6: Graph substitution

The example from figure 5 can be written in algebraic notation as follows, where L={s,a,xs,xj,e} (s=Start Event, a=Activity, xs=XOR Split, xj=XOR Join, e=End Event):

$$
\begin{aligned}
G_1 = (&\{I, II, III, IV, V, VI\},\\
&\{1,2,3,4,5,6\},\\
&\{(1,I),(2,II),(3,II),(4,III),(5,IV),(6,V)\},\\
&\{(1,II),(2,III),(3,IV),(4,V),(5,V),(6,VI)\},\\
&\{(I,s),(II,xs),(III,a),(IV,a),(V,xj),(VI,e)\},\\
&\emptyset)
\end{aligned}
\tag{17}
$$

Graph Productions Rules. For a better understanding, we informally introduce graph production rules. We cover a special kind, the set theoretic graph substitution rule. Therewith we replace a part of a graph with another graph. This is shown in figure 6. We have an original graph L and a destination graph R. A graph substitution rule defines these two graphs and a common partitial graph K. K is contained in L as well as in R. K does not consist of anything else ($K = L \cap R$). If we have an original graph G that contains L as a partitial graph we can cut off anything in G that is equal to L, just leaving K as cutting/gluing points. We then can glue graph R to the remaining cutting/gluing points. An example rule is shown in figure 7.

A graph substitution rule is written as (L, K, R), containing three graphs. The following process steps apply to an original graph G and a graph substitution rule (L, K, R):

grammar section in this notes can of course only cover a very small subset.

Figure 7: Sample graph substitution rule (L, K, R)

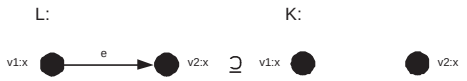

Figure 8: Context–free graph substitution rule $(L, K, ..)$

1. (L, K, R) can be applied on G, if G contains L as partitional graph.
2. Every vertices and edges from L that are not contained in K are deleted from G. This creates a remaining graph D.
3. Every vertices and edges from R that are not contained in K are added to D. The added elements should be glued to D as R is connected to K.

The formal definition of a graph substitution rule is as follows: A graph substitution rule p is a triple of graphs (L, K, R), so that there exists a graph K included in L and R as a partitional graph, $K \subseteq L$ and $K \subseteq R$ and $L \cap R = K$. L is the left side, R the right side of the rule and K consists of the common elements of the left and right side.

The formal definition of a graph substitution rule was easy, but problems occur if we try to formalize the execution steps of a rule p on a graph G (the informal process steps above). The definition of $L \subseteq G$ does not match alone. We must first rename either L or G to allow comparison. Further we must take care that no vertex of $R-L$ is contained in G (formally: $(L \cup R) \cap G = L$). The complete formalization can be found in [LMT95].

Graph Grammar. A graph grammar $\gamma = (T, N, S, R)$ consists of a set T of terminal labels for vertices and edges, a set N of non–terminal labels, an initial graph S over $T \cup N$ and a set R of graph substitution rules over $T \cup N$.

Graph Language. A graph language $\lambda(\gamma)$ that is defined through $\gamma = (T, N, S, R)$ consists of all terminal labeled graphs $GRAPH_T$ that can be derived from S using R.

4.2 Context–free Graph Grammars

Graph grammars can be classified using the Chomsky hierarchy. A Chomsky grammar has a context free language if a single variable can be replaced, regardless of the left and right context of the used word [EP02].

The computability of context–free grammars is classified under NC. Therewith it can be solved efficiently (in polylogarithmic time) on parallel computers.

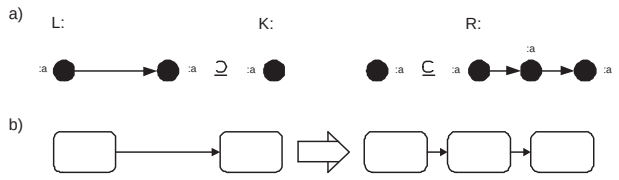

Figure 9: A context-free graph substitution rule (a) for adding an activity between two existing activities in a BPMN diagram (b)

Context–free Graph Grammar. A graph grammar $\gamma = (T, N, S, R)$ is called context free, if the left sides of the rules (R_L) are equal to $(\{v1, v2\}, \{e\}, s, t, l, m)$ with $s(e) = v1$ and $t(e) = v2$ and if the middle part of the rules (R_K) are equal to $(\{v1, v2\}, \emptyset, \emptyset, \emptyset, l, \emptyset)$ and all vertices in $GRAPH_T$ are marked with a special label (and are therewith not distinguishable). The left and middle part of a context–free rule is shown in figure 8.

An example for a context-free rule is shown in figure 9. Between two activities in a BPMN diagram another activity can be inserted. The problem is that this diagram can only consists of activity elements, because all vertices should be labeled the same (a).

4.3 Context–sensitive Graph Grammars

In the Chomsky hierarchy, a grammar has a context–sensitive language if a variable A is replaced through a string α with a length of at least 1. The replacement of A is only accomplished, if the left and right context that is required in the rule is found in the original word [EP02].

The computability of context–sensitive grammars is classified under `PSPACE`. Therewith it is solvable with polynomial memory and unlimited time.

Context–sensitive Graph Grammar. A graph grammar $\gamma = (T, N, S, R)$ is context–sensitive, if the right sides of the rules (R_R) contain at least as much vertices as the left side of the rules (R_L). Formally: $|R_{L_V}| \leq |R_{R_V}|$. The vertices can be labeled freely. *[Incomplete definition]*

An example for a context–sensitive rule is shown in figure 10. The rule creates an edge (sequence flow) between a vertex labeled xs (XOR split) and a vertex labeled e (end event). The rule p is formally written as (L, K, R):

$$\begin{aligned} L =& (\{v1, v2\}, \emptyset, \emptyset, \emptyset, \{(v1, xs), (v2, e)\}, \emptyset) \\ K =& (\{v1, v2\}, \emptyset, \emptyset, \emptyset, \{(v1, xs), (v2, e)\}, \emptyset) \\ R =& (\{v1, v2\}, \{e1\}, \{(e1, v1)\}, \{(e1, v2)\}, \\ & \{(v1, xs), (v2, e)\}, \emptyset) \end{aligned} \tag{18}$$

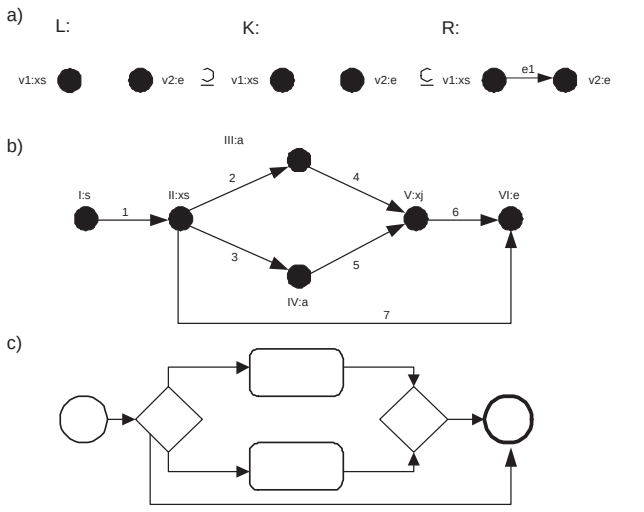

Figure 10: A context-sensitive graph substitution rule (a) applied to a graph (b) and visualized as BPMN diagram (c)

Further considerations

Vertex Attributes. A vertex label only indicate a terminal or non–terminal marker for a vertex. In the previous section examples we used it to distinguish between different diagram elements. Therewith, a label indicates the type of a a vertex. Further information can be added by using *vertex attributes*.

Terminal vs. Non–Terminal Labels. A label indicates the type of a vertex. Sometimes, we need both, a terminal as well as non–terminal label for the same type. E.g. as long as a diagram is under construction, all activities are labeled a (non–terminal label). An activity can therewith be replaced through other diagram constructs. If we like to make an activity final, we need a rule to transform a to A (terminal label).

It is possible to add non–terminal labeled vertexes, that do not map to a specific part of the target diagram, while constructing diagrams. Those could be used as variation points.

Node Replacement Graphs. Another way to generate graphs is by using node replacement graphs. Thereby, a production is of the form $X \rightarrow (D, C)$ where X is a non–terminal node label, D is a graph and C is a set of connection instructions.

Graph Tests. Certain conditions can be defined on a graph G that must match after applying a graph production rule. This is called a *graph test*. One rule could be, that there is no outgoing sequence flow from an end event in BPMN. This could be written formally as follows:

$$\forall x \in G_V : l^G(x) = e \rightarrow \nexists k \in G_E : s(k) = x \tag{19}$$

Open Issues

- Is it possible to define a graph grammar for visual process modeling languages like UML activity diagrams or BPMN?
- What kind of graph grammar would it be (context–free vs. context–sensitive)?
- Assuming, such a grammar is context–sensitive, is it possible to define a context–free subset of the rules while keeping as much potentials of the language as possible? To define the potentials of a language, e.g. workflow patterns could be used [vAtHKB00].
- Is it possible to use the labels instead of renaming the vertices and edges of K to match against a graph G?

Further readings

- All required definitions like partitional graph, intersection or union matching to this notes can be found in [LMT95].
- Considerations on context–free graph grammars can also be found in [RS97].
- Processes can be executed or simulated using graph grammars. In [LMT95] a transformation rule for Petri nets is given.
- In [KM97] it is said that Petri nets can be mapped on graph grammars, but not the other way around. They cited an article [Cor95].
- In [KM97] a formal method for business process engineering based on graph grammars is introduced. The authors used the set theoretic approach enhanced with hierarchical structured graph grammars.
- A graph grammar programming environment was developed at the RWTH Aachen [PRO, Sch00].

5 Intelligent Agents

Michael Wooldridge stated in [Woo02] that there is, for an increasingly large number of applications, a strong need for systems that can decide for themselves what they need to do in order to satisfy their design objectives.

Those systems are known as agents. Intelligent or autonomous agents must operate robustly in rapidly changing, unpredictable, or open environments, where there is a significant possibility that actions can fail in an unpredictable way.

Those environments can be classified by accessibility, determinism, episodic vs. non–episodic, static vs. dynamic and discrete vs. continuous. If an environment is sufficiently complex, than the fact that it is actually deterministic is not much help; i.e. it might as well be non–deterministic. The most complex class is an inaccessible, non–deterministic, non–episodic, dynamic and continous environment.

Using the above statement, think a bit about the internet. Now think about handling interorganizational workflow in such an environment. The advocates of agent based technologies claim to have the solutions covering the problems. A lots of publications concerning *agents* and *workflow* have been written. Some are cited in this section.

5.1 Foundations

There exists no common definition for agents yet. In [MS99], an intelligent agent is defined as an entity that has some degree of autonomy, reasoning and learning capabilities and is able to communicate in an intelligent way with other agents. An agent expresses autonomy by showing reactive and/or proactive behavior regarding its environment and taking initiative independently of a human.

Agents can be seen from different views like computational or even philosophical. A computational view might see agents as the next step in programming, like a successor of the object-oriented programming paradigm. Agents enhance object with *subject*–like features (e.g. ontology, knowledge/believe about their environment, goals). Communication is enhanced by asking questions to other agents instead of just invoking methods.

Intelligent agents usually have a persistent state, like objects in true object languages like Self. As objects in Self, agents can be cloned and modified to create new versions keeping the knowledge and experience of its predecessor.

To successful develop agents, theories from different domains have to be combined. Those are the *theory about actions and change* (from artificial intelligence), the *theory of norms* (covering social attitudes), *database theory*, the *theory of concurrent computing* as well as *principles of software engineering*.

In the following paragraphs, formal definitions concerning agents are given; they are taken from [Woo02].

Agent. If we define the states of the environment of an agent by a set $S = \{s_1, s_2, ...\}$, and the effective capabilities of an agent by a set $A = \{a_1, a_2, ...\}$ of actions, then an agent can be defined as a function:

$$action : S^* \rightarrow A \tag{20}$$

This function maps sequences of environment states to actions. This agent is called a *standard agent*, he decides what action to perform on the basis of its history — the sequence of environment states that the agents has yet encountered.

Purely reactive Agent. An agent is purely reactive, if the decision making is based entirely on the present:

$$action : S \rightarrow A \tag{21}$$

The purely reactive agent simply responds directly to his environment, without referencing former states. A purely reactive agent is a subset of a standard agent.

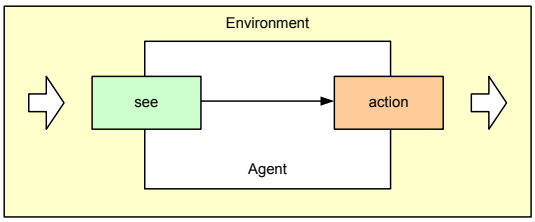

Figure 11: An agent in an environment

Environment Behavior. The (non–deterministic) behavior of an environment can be modeled as a function[2]:

$$env : S \times A \to 2^S \tag{22}$$

The function *env* takes the current state of the environment $s \in S$ and an action $a \in A$ as input and maps them to a set of environment states that could result from performing the action a in state s. If the result always contains one single member ($|env| = 1$) then the environment is deterministic.

Agent History. If we define s_0 as the initial state of the environment at the time the agent appears, then the sequence

$$h : s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots \xrightarrow{a_{u-1}} s_u \xrightarrow{a_u} \ldots \tag{23}$$

represents a possible history of the agent in the environment iff $\forall u \in \mathbb{N}, a_u = action((s_0, s_1, ..., s_u))$ and $\forall u \in \mathbb{N}^+, s_u \in env(s_{u-1}, a_{u-1})$ hold true. We denote by $hist(agent, environment)$ the history of an agent in an environment.

Behavior Equivalence. Two agents ag_1 and ag_2 are behaviorally equivalent with respect to an environment *env* iff $hist(ag_1, env) = hist(ag_2, env)$.

Perception. The concepts of an agent can be further refined. The agents decision function is divided into a function to observe its environment *see*, which maps environment states to percepts P and *action* is now overloaded as a function mapping sequences of percepts to actions:

$$see : S \to P \tag{24}$$
$$action : P^* \to A \tag{25}$$

This concept is visualized in figure 11. It allows interesting considerations about agents. Suppose, that we have two environment states $s_1 \in S$ and $s_2 \in S$, such that $s_1 \neq s_2$, but $see(s_1) = see(s_2)$. This means, that two different environment states are mapped to the same percept. The agent could therefore not distinguish between s_1 and s_2. Let x and y represent two statements, so that we have a set of environment states:

[2]Where 2^S is the power set of S.

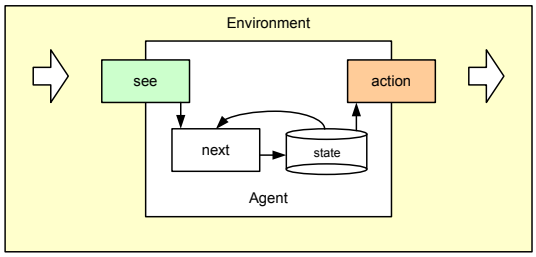

Figure 12: An agent with state

$$S_{xy} = \{\underbrace{\{x, y\}}_{s_1}, \underbrace{\{x, \neg y\}}_{s_2}, \underbrace{\{\neg x, y\}}_{s_3}, \underbrace{\{\neg x, \neg y\}}_{s_4}\} \tag{26}$$

If an agent only perceives x using his *see* function, than the state of y is unimportant. He therefore can not distinguish between the states s_1 and s_2 as well as between s_3 and s_4.

Agents with State. The history function of an agent can be further refined. Agents with state maintain internal data structures which are used to record information about the environment state and history. Let I be the set of all those internal states. The agent's decision making function is overloaded to generate actions based on the internal state information:

$$action : I \to A \tag{27}$$

Additionally, a function *next* is needed, which maps a percept and an internal state to another internal state:

$$next : I \times P \to I \tag{28}$$

This concept is visualized in figure 12. The agent starts in an initial state s_0. It observes its environment and generates a percept using *see*. The internal state of the agent is then updated using the *next* function, which takes the percept and s_0 as input. The agents than generates an action using the updated set I. After that, another cycle begins.

Concrete Architectures. In the last paragraphs, we defined only abstract agents. We used an abstract function *action* which somehow manages to select an action. The following is an enumeration of possibilities to implement this function:

- **Logic based Agents:** Decision making is realized through logical deduction.
- **Reactive Agents:** Decision making is implemented as some kind of direct mapping situations to actions.
- **Belief–Desire–Intention Agents (BDI)**: Decision making is realized through manipulating data structures that represent the beliefs, desires and intentions of the agent.

- **Layered Architecture:** Decision making is partitioned into different layers, each dealing with the agent's environment at different levels of abstraction.

5.2 Workflow Agents

If we imagine a workflow based on the internet, the number of possible exceptions is very large; this leads to the consequence that most workflows are incomplete by design. If one tries to cover all possible exception into a workflow, it will soon render it unusable. In [HS98] it is stated, that a dynamic approach is desirable, so that the workflow is enhancing itself by including often appearing exceptions.

The metamodel of most workflow systems is based on a variant of an activity network which uses graph structures. This workflow is defined by some central authority prior to use. The static structure make it hard to use for humans, which like to have some kind of control about their work. Furthermore, there can be dynamic requirements derived from a highly complex environment, that could not be foreseen and included at the design time of the process. That is the point, where workflow agents enter the game.

Agents for Workflow. Workflow agents perceive, reason about and affect their environment. Furthermore they can be adaptive and communicative. For a given environment, different types of agents could be developed. Each of them is assigned one or more roles. The roles of greatest interest for workflows are user-, resource- and broker agents. Those agents are aware of their local environment and can adapt to a workflow. The agents communicate with each other to ensure global constraints and efficiency. Most important, agents can learn from repeated instances and therewith adapt to changing environments.

Agents and ACID. ACID means Atomicity, Consistency, Isolation and Durability for transactions in traditional database engineering. In [HS98] it is stated that the ACID criteria can be hardly met in an open environment. The authors defined some kind of *relaxed transaction processing*, where criteria can only be guaranteed for a special time interval.

Interoperation. Another area for workflow agents is the interaction between different workflows. A workflow could be defined by web–services, which could be coordinated by workflow agents.

Dynamic Workflow Agents. A concept for dynamic workflow agents has been developed at the HP–Labs in Palo Alto [CDHG00]. The authors stated, that E-Commerce is a dynamic plug and play environment. Services as well as business partnerships need to be created dynamically and must be maintained only a short time. They developed a *dynamic behavior modification* for agents. Those dynamic agents do not have a fixed set of predefined functions, instead they carry application specific actions, which can be loaded and modified on the fly. The agents can expose those abilities as well as their knowledge, intentions and goals through messaging. Furthermore they developed an infrastructure supporting dynamic service construction, modification and movement in which dynamic workflow agents can interact.

The dynamic workflow agents are designed as web objects. They can be configured via an URL, on which the state of the agent and its current tasks are listed. The agents communicate using XML, and can dynamically load XML interpreters for different domains. The interpreters can be generated automatically using semantic web features. Several dynamic agents can cooperate to reach a goal, like a purchase task.

A dynamic agent is a carrier of tasks that represents steps of a business process. It is a continuous running object with a persistent communication channel and knowledge across tasks and processes. They are designed to integrate system components on the fly.

Further readings

- An excellent paper covering the key concepts of intelligent agents is [Woo02]. Much of the foundation section is based on this paper.
- In [CDHG00] the dynamic workflow agents are described.
- Mobile agents for interorganizational workflow are considered in [MLL97].
- In [Bog99] mobile agents with Java are considered.
- An open source framework from IBM for Java and agents, called Aglets, can be found under [AGL]. The webpage also contains links to further information like virtual enterprises with agents.

6 Process Algebra

Process algebra can be used for the description of and reasoning about processes. Different algebras covering sequential, parallel, communication and mobile processes have been developed during the last decades.

Almost all process algebra are based on a very small set of axioms, making them easy to handle. By combining the axioms or adding additional components like constants or rules, one has a very powerful framework.

6.1 Foundations

We start the foundations by introducing a basic process algebra (BPA), on which almost all other algebras are based. We then extend or investigate the BPA by different things like deadlocks, empty processes, recursion, bisimulation, concurrency or abstraction.

$x + y = y + x$	(A1)
$(x + y) + z = x + (y + z)$	(A2)
$x + x = x$	(A3)
$(x + y)z = xz + yz$	(A4)
$(xy)z = x(yz)$	(A5)

Table 2: The axioms E_{BPA} of the basic process algebra BPA

Basic Process Algebra (BPA). The basic process algebra is a tuple $BPA = (\Sigma_{BPA}, E_{BPA})$ where Σ_{BPA} is a signature, e.g. the set of constants and function symbols, and E_{BPA} is a set of axioms, e.g. a set of equations of the form $t_1 = t_2$ where t_1 and t_2 are terms.

Σ_{BPA} consists of two binary operators, $+$ and $\cdot$, and a number of constants, named $a, b, c, ...$ The set of constants is denoted by A, so that we have $\Sigma_{BPA} = \{+, \cdot\} \cup A$. The operator $\cdot$ is often omitted, we write ab instead of $a \cdot b$. Also, $\cdot$ binds stronger than all other operators and $+$ binds weaker. The axioms E_{BPA} of the basic process algebra are given in table 2.

We still need some semantics for the specification of the BPA. The constants $a, b, c, ..$ are called atomic actions which are indivisible. The $\cdot$ is the product or *sequential composition*; $x \cdot y$ is the process that first executes x and after completition of x starts y. The $+$ is the sum or *alternate composition*; $x + y$ is the process that executes either x or y but not both.

Basic Terms. The BPA has a set of terms which are called basic terms: (1) every atomic action a is a basic term, (2) if t is a basic term and a is an atomic action, then $a \cdot t$ is a basic term and (3) if t, s are basic terms, then $t + s$ is a basic term.

Action Relations. The action relations describe which actions a process can perform. We define two relations on the set of BPA terms, $\xrightarrow{a}$ and $\xrightarrow{a} \surd$ for each $a \in A$.

- $t \xrightarrow{a} s$ denotes that t can execute a and then turn into s; i.e. the execution of step a in state t moves us to state s.
- $t \xrightarrow{a} \surd$ denotes that t can terminate by executing a.

An inductive definition of the possible action relations is given in table 3. The table also contains generalized action relations where σ is defined as $\sigma \in A^+$. Therewith, σ is a sequence of actions and $t \xrightarrow{\sigma\!\!\!\!\rightarrow} s$ is called a trace of t by σ.

For example, the action relations of the term $(a + bb)c$ are (1) $(a + bb)c \xrightarrow{ac} \surd$ and (2) $(a + bb)c \xrightarrow{bbc} \surd$. A graph containing all possible action relations is shown in figure 13.

$a \xrightarrow{a} \surd$	
$x \xrightarrow{a} x'$	$\Rightarrow x + y \xrightarrow{a} x'$ and $y + x \xrightarrow{a} x'$
$x \xrightarrow{a} \surd$	$\Rightarrow x + y \xrightarrow{a} \surd$ and $y + x \xrightarrow{a} \surd$
$x \xrightarrow{a} x'$	$\Rightarrow xy \xrightarrow{a} x'y$
$x \xrightarrow{a} \surd$	$\Rightarrow xy \xrightarrow{a} y$
$t \xrightarrow{a} s$	$\Rightarrow t \xrightarrow{a\!\!\!\!\rightarrow} s$
$t \xrightarrow{\sigma\!\!\!\!\rightarrow}$ and $s \xrightarrow{\rho\!\!\!\!\rightarrow} r$	$\Rightarrow t \xrightarrow{\sigma\rho\!\!\!\!\rightarrow} r$
$t \xrightarrow{a} \surd$	$\Rightarrow t \xrightarrow{a\!\!\!\!\rightarrow} \surd$
$t \xrightarrow{\sigma\!\!\!\!\rightarrow} s$ and $s \xrightarrow{\rho} \surd$	$\Rightarrow t \xrightarrow{\sigma\rho} \surd$

Table 3: The action relations for the BPA

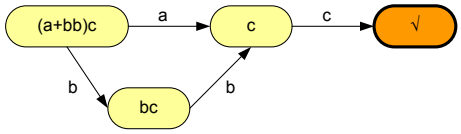

Figure 13: A graph based on the action relations for the term $(a + bb)c$

Reasoning. We could do some reasoning based on the axioms. For example we could prove that $a(b+b) = ab+ab$:

$$a(b + b) \overset{A3}{\Rightarrow} a(b) \Rightarrow ab \overset{A3}{\Rightarrow} ab + ab \quad \blacksquare \tag{29}$$

We had to prove this term, because the axiom $x(y+z) = xy + xz$ that gives full distributivity is missing in table 2. That is correct, because the left side describes a free choice process whereas the right side describes a pre–selective choice.

We could also prove that $BPA \vdash a(b + b) = ab + ab$, where $X \vdash y$ stands for derivability, meaning we can derive y from X:

$$BPA \vdash x = x + x \tag{30a}$$
$$BPA \vdash ab = ab + ab \tag{30b}$$
$$BPA \vdash a(b + b) = ab + ab \quad \blacksquare \tag{30c}$$

Line 30a was derived using axiom A3. Line 30b was derived by substituting x by ab. Line 30c was derived by expanding the left side b using rule A3.

Deadlock. A process is called a deadlock if it has stopped executing actions and, for whatever reason, can not continue although there are actions left. A deadlock is denoted by the special constant symbol δ, where $\delta \notin A$.

The extended algebra BPA_δ consists of additional axioms shown in table 4 ($\Sigma_{BPA_\delta} = \Sigma_{BPA} \cup \{\delta\}$, $E_{BPA_\delta} = E_{BPA} \cup \{A6, A7\}$). The axiom A6 states that there is

$x + \delta = x$	(A6)
$\delta x = \delta$	(A7)

Table 4: The additional axioms for deadlocks

$x\epsilon = x$	(A8)
$\epsilon x = x$	(A9)

Table 5: The additional axioms for empty processes

no deadlock as long as there is an alternative action. A7 states that after a deadlock has occurred, no further actions are possible.

Empty Process. As a counterpart to δ, we have a process ϵ, the empty process. The only action ϵ contains is successful termination. We assume that $\epsilon \notin A$ and define some additional axioms shown in table 5. The complete definition for BPA_ϵ is $(\Sigma_{BPA_\epsilon} = \Sigma_{BPA} \cup \{\epsilon\}, E_{BPA_\epsilon} = E_{BPA} \cup \{A8, A9\})$.

A process algebra containing the deadlock as well as the empty process is written as $BPA_{\delta\epsilon}$.

Recursion. Recursion allows the possibility for the object to be defined to occur again in the right-hand side of the definition. We have to kinds of recursive definitions, the *recursive equation* and the *recursive specification*:

- **Recursive equation**: $X = s(X)$, where $s(X)$ is a term over BPA containing the variable X but no other variables.
- **Recursive specification**: $X = s_X(V)$, for each $X \in V$ where s_X is a term over BPA containing variables from the set V.

Guarded Recursion. (1) If s is a term over BPA containing a variable X, we call an occurrence of X in s guarded if s has a subterm of the form $a \cdot t$ where a is an atomic action and t is a term containing X. (2) A term s is *completely guarded* if all occurrences of all variables in s are guarded. (3) A recursive specification E is completely guarded if all right hand sides of all equations of E are completely guarded terms. (4) A term s is also guarded if we can rewrite s to a completely guarded term by using the axioms.

Bisimulation. A bisimulation is a binary relation R on two processes p and q: (1) if $R(p,q)$ and $p \xrightarrow{a} p'$ then there is a q' such that $q \xrightarrow{a} q'$ and $R(p',q')$, (2) if $R(p,q)$ and $q \xrightarrow{a} q'$ then there is a p' such that $p \xrightarrow{a} p'$ and $R(p',q')$, (3) if $R(p,q)$ then $p \xrightarrow{a} \surd$ iff $q \xrightarrow{a} \surd$. Two processes p and q are bisimilar, denoted by $p \leftrightarrow q$, if there exists a bisimulation between p and q.

Informally, bisimilarity means that the branching structures of two processes are equal; e.g. if one process is capable of performing a step a to a new state then any equivalent process should be able to do an a step to a corresponding state.

$x \parallel y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x$	(M1)
$a \lfloor\!\lfloor x = ax$	(M2)
$ax \lfloor\!\lfloor y = a(x \parallel y)$	(M3)
$(x+y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$	(M4)

Table 6: The additional axioms for concurrency

$x\tau = x$	(B1)
$x(\tau(y+z)+y) = x(y+z)$	(B2)

Table 7: The additional axioms for abstraction

Concurrency. Processes that can occur in parallel are denoted by the merge operator $\parallel$, e.g. $x \parallel y$. Merged processes can be seen as interleaved serial processes. Every time we see either the next action of x or the next action of y. Intuitively, we can state $a \parallel b = ab + ba$.

To formally define the extended BPA signature, called Process Algebra (PA), we need the left merge operator $\lfloor\!\lfloor$. Here, $x \lfloor\!\lfloor y$ has the same meaning as $x \parallel y$ with the restriction that the first step must come from x. We define the components of PA as follows: $\Sigma_{PA} = \Sigma_{BPA} \cup \{\parallel, \lfloor\!\lfloor\}$, and $E_{PA} = E_{BPA} \cup \{M1, M2, M3, M4\}$. The new axioms $M1 - M4$ are shown in table 6.

Renaming. We define an unary renaming operator ρ_f which replaces every occurrence of a constant $a \in A$ by $f(a)$. Thereby f maps to $f : A \to A \cup C$, where C is a set of special constants like δ or ϵ.

Abstraction. We can abstract from certain process actions by hiding them. However, we can not easy remove a constant symbol, instead we substitute it by a special symbol. Therefore, we introduce a new constant τ, the *silent step*. The substituted term can be processed by new axioms, the τ laws, given in table 7.

For example, if we have the process $a+b\delta$, by abstracting from b we obtain $a + \tau\delta$ and can not remove τ. If we have a process abc and abstract from b, we obtain $a\tau c = ac$. We can now define BPA^τ or $BPA^\tau_{\delta\epsilon}$ as previously (omitted).

Communication. Communication is rather hard to handle using standard PA. Other process algebra like CSP or CCS introduce special syntax and semantics.

Instead of looking at this algebra, we consider a newer approach for the communication inside concurrent and mobile systems, the π-calculus.

6.2 π-Calculus

The π-calculus is a mathematical model of processes with changing structures, called mobile processes. It supports systems with arbitrary linked processes and allows for changing this linkage by the use of communication. Therefor, a technique called link passing is used. In this subsection we introduce the basic construction of the π-calculus;

Prefixes	$\alpha ::=$	$\overline{a}x$	Output
		a(x)	Input
		τ	Silent
Agents	$P ::=$	$\mathbf{0}$	Nil
		$\alpha.P$	Prefix
		$P + P$	Sum
		$P \mid P$	Parallel
		if $x = y$ then P	Match
		if $x \neq y$ then P	Mismatch
		$(\mathbf{v}x)P$	Restriction
		$A(y_1, ..., y_n)$	Identifier
Definition	$A(x_1, ..., x_n) \stackrel{def}{=} P \ (i \neq j \Rightarrow x_i \neq x_j)$		

Table 8: The syntax of the π-calculus

much more details and examples are referenced under further readings.

The π-calculus is based on names (contained in the set $\mathcal{N}$) and process, or agent identifiers from the set $\mathcal{K}$. We use $a, b, ... \in \mathcal{N}$ to range over names and $P, Q, ... \in \mathcal{K}$ to range over processes. The syntax of the π-calculus is shown in table 8. It is explained in the following paragraphs.

Empty Agent. The empty agent $\mathbf{0}$ cannot perform any action.

Output Prefix $\overline{\mathbf{a}}\mathbf{x}.\mathbf{P}$. The output prefix is used to send the name x along the name $\overline{a}$.

Input Prefix $\mathbf{a}(\mathbf{x}).\mathbf{P}$. The input prefix is used to receive a name along the name a and then x is replaced by the received name.

Silent Prefix $\tau.\mathbf{P}$. The silent prefix τ represents an agent that does not interact with the environment.

Sum $\mathbf{P + Q}$. The sum represents an agent that can either execute P or Q.

Parallel Composition $\mathbf{P \mid Q}$. The parallel composition represents an agent that executes P and Q in parallel. P and Q can communicate if one performs an output and the other a corresponding input along the same name.

Comparison. The agents if $x = y$ then P or if $x \neq y$ then P will behave as P if x and y are the same or different names.

Restriction. The agent $(\mathbf{v}x)P$ behaves as P but the name x is locally bound to P, so it can not be used as a free name anymore. We define the free names of the process P as $fn(P) \rightarrow \mathcal{N}$ and the bound names as $bn(P) \rightarrow \mathcal{N}$ where $fn(P) \cap bn(P) = \emptyset$.

Identifiers. Every identifier has a definition so that $A(y_1, ..., y_n)$ behaves as P with y_i replacing x_i for each i. A definition can be thought of as a process declaration with $x_1 ... x_n$ as formal parameters. The identifier is an invocation which replaces every x_i with y_i.

In later versions of the calculus, a bang-operator (written as $!P$) is introduced instead of identifiers. This operator replicates a given process recursively and simplifies the π-calculus.

Substitution. A substitution is a function that maps names to names: $f_{sub} : \mathcal{N} \rightarrow \mathcal{N}$. We write $\{x/y\}$ for the substitution that maps y to x.

Examples. We consider some basic examples for the π-calculus in the following lines.

$$a(x).\overline{c}x \mid \overline{a}b \xrightarrow{\tau} \overline{c}b \mid \mathbf{0} \tag{31}$$

The example in formula 31 shows communication between parallel processes. The right hand process sends the name b over the name $\overline{a}$. The left hand process receives the name b over the name a and thereafter replaces every local occurrence of x with b.

$$S \mid C \mid P \equiv \overline{b}a.S \mid b(c).\overline{c}d.C \mid a(e).P \tag{32}$$

A popular example for the π-calculus is a server S that holds a link a to a printer. The server sends the link to a client C which in turn can send data d to the printer P. The processes are shown in formula 32.

$$(\mathbf{v}e)(S \mid e.R) \tag{33}$$

Formula 33 shows communication using a private channel e. Currently, only process S can invoke R by using $\overline{e}$. It could also give the link name e to another process, thereby expanding the scope of the restriction. Please note, that e and $\overline{e}$ are an abbreviation for $e(x)$ and $\overline{e}x$ in the case that x is unimportant; $\overline{e}$ is thereby used as a trigger.

$$(\mathbf{v}p)\overline{c}p.\overline{p}d.\overline{p}e.S \tag{34}$$

An extended example for using private channels is shown in formula 34. Imagine, S wants to send two names, d and e, to a client. By first establishing a private channel p, it ensures that the same client receives both names in the case that there exist multiple clients.

$$c(p).p(x).p(y).Q \tag{35}$$

The matching client for formula 34 is shown in formula 35. It first receives the bound name p and thereby expands the scope to the client. It then receives c and d over p.

The scope extrusion of p for the process S (formula 34) to a particular process C (formula 35) is shown in figure 14. In a) the original situation is given; a process S holding two names d and e as well as a bound name p. In b) the

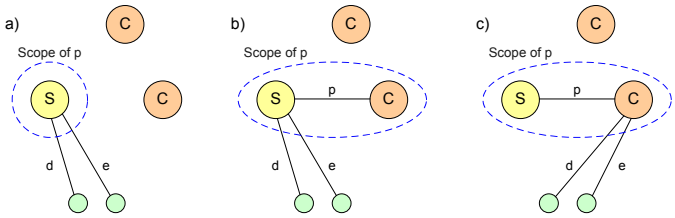

Figure 14: π-calculus scope extrusion

bound name p is send to a particular process C. Finally, in c), the names d and e are transmitted from S to C over p.

Further considerations

Open Issues

- As van der Aalst asked in an unpublished discussion paper [vA], how efficient can the workflow patterns be modeled using process algebra?

Further readings

- The foundations of process algebra can be found in [BW90].
- Hoare's CSP (Communicating Sequential Processes) can be found in [Hoa78].
- Milner's CCS (Calculus of Communicating Systems) is described in [Mil89].
- In his Turing Award lecture, Robin Milner [Mil93a] draws the path from sequential over communication up to mobile process algebra.
- The π–Calculus was early described in [MPW92]. The first part contains easy to follow examples, whereas the second part contains more formal descriptions.
- Robin Milner wrote a tutorial [Mil93b] and a single book [Mil99] about the π–Calculus. The book contains a nice introduction to automata theory and labeled transition systems as well as good examples. Beside the introduction, the key concepts are difficult to read and comprehend. The earlier papers mentioned above are much easier to understand.
- Another introduction to the π-Calculus can be found in [Par01].

7 Business Process Model Transformations

A rule based approach for reasoning about business process model transformations is introduced in [SO00].

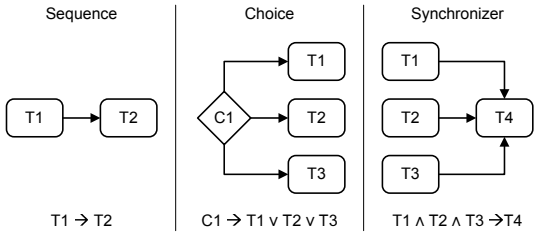

Figure 15: Mapping modelling structures to triggering constraints

The general idea is very simple. The authors start with defining three types of relationships between workflow graphs. Then they state different transformation rules that obtain a given kind of relationship between the old and the new graph. The paper is concluded by giving some formal triggering constraints, which can be used to reason about the transformation rules. However, this is not done in the paper.

Nodes. The authors use the concept of *execution nodes* and *coordination nodes* to define relationships. An execution node is a node that directly performs action, usually it is an activity that does not act as a null (or placeholder) activity. Choice and merge nodes are coordination nodes.

Equivalent Relationship. A workflow graph G' is defined to be structurally equivalent to another workflow graph G, written as $G' \leftrightarrow G$, if the sets of execution nodes in both G and G' are equal and each of them preserves the structural/control flow constraints specified in the other.

Imply Relationship. A workflow graph G' implies another workflow graph G, written as $G' \rightarrow G$, if the sets of execution nodes in both G and G' are equal and G' preserves the structural/control flow constraints specified in G. G may not satisfy all of the structural/control flow constraints specified in G'.

Subsume Relationship. A workflow graph G' subsumes another workflow graph G, written as $G' \supset G$, if the set of execution nodes in G is a subset of the execution nodes in G' and G' preserves the structural/control flow constraints specified in G. G may not satisfy all of the structural/control flow constraints specified in G'.

Transformation Rules. Transformation rules that preserve a given kind of relationship between an original graph G and a transformed graph G' can be found in the original paper.

Formal Considerations. The transformation rules stated in the original paper are given informal. However the authors give a rule based approach of how to reason about them. They defined a triggering constraint of the form $X \rightarrow Y$, meaning that the completion of X triggers

the start of Y. X and Y are algebraic terms. The mapping of some modeling structures to triggering constraints is shown in figure 15. The transformation principles are of the kind $R_1 \rightarrow R_3, R_2 \rightarrow R_3 \Leftrightarrow R_1 \vee R_2 \rightarrow R_3$ where R_X is a triggering constraints term.

Further readings

- This section just covers some interesting key concepts of the paper; further information can be found in the paper itself [SO00].

8 Pockets of Flexibility

In [SSO01] a runtime approach for variability can be found. The authors of the paper categorize three types of change characteristics in workflow processes: dynamic, adaptive and flexible workflows. A dynamic workflow is the ability to change when the underlying business process evolves. The important question is, how to map the active workflows to the new model. An adaptive workflow can react to exceptional circumstances. A flexible workflow is based on a loosely or partially specified model. The missing pieces are constructed at runtime and may be unique to each instance.

The pockets of flexibility center at flexible workflow by bridging the workflow process and workflow execution aspect. The authors introduce a layer between the definition and execution data as well as a workflow model which only consists of the partial definitions. Those partial definitions are called *flexible workflow* and consist of:

- A defined *core process* containing
  - Pre-defined workflow activities and
  - *Pockets of flexibility* within the pre-defined process with
    - A set of *workflow fragments*, where a fragment can be a single or compound activity and
    - a special activity called *build activity* that provides rules for instantiating the pocket with a valid composition of workflow fragments.

An example workflow with a pocket of flexibility is shown in figure 16. The notation is BPMN [BPM04] with an enhancement, the pocket of flexibility.[3]

The pocket of flexibility can be seen as an activity within a process, therewith it does not depend on a particular workflow language. The example was taken from the original paper [SSO01] and converted to BPMN. The pocket of flexibility is defined as special kind of sub–process, marked with a build–marker. The parts inside the build activity

[3]The enhancement was proprietary defined by the author of this note to illustrate the examples in this section.

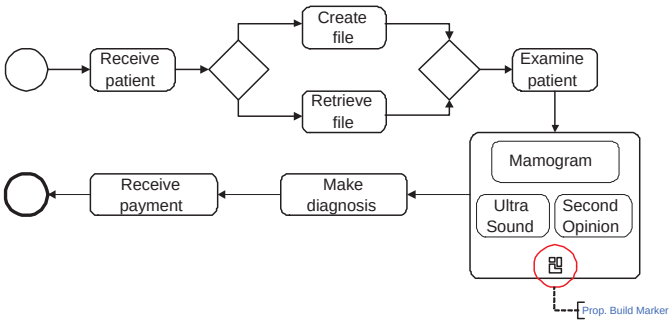

Figure 16: Specifying flexible workflow using pockets of flexibility

will be combined on a case–by–case basis. They can consists of different kinds of process fragments. When the process flow arrives at the build activity, the sub–process inside is constructed using these fragments. Prior to building, the instance specification of the whole process is called an *open instance*. After building all specific build activities, the instance specification is called *instance template*. A process may contain several build activities, in hierarchical or sequential order.

The build activity contains rules for combining the contained elements. Only instance templates following these rules are allowed. The build activity can be executed fully or partially automated or entirely manual. The fully automatic build uses instance data and given constraints. The partially automated building invokes an application program to build the instance template from a given set of process fragments, following the given constraints. Under manual controll, new fragments can be defined using a process/workflow client. The new and existing fragments are used to build an instance template.

There remain some questions, like what kind of process–fragments should be allowed? The authors of the paper state that sequence, fork and synchronize as well as iteration fragments can be used very well. It is suggested not to use choice or merge constructs, since those choices should be made while executing the build activity. The authors further identify three factors for customizing a build activity for a given application:

- Type: Fully or partially automated or manual controlled selection of process fragments
- Extent: How many process fragments can be selected from the available pool
- Structure: What modeling constructs can be used to compose the process fragments

Further work needs to be done by designing an appropriate language for specifying the build rules and constraints.

Open Issues

1. How to solve dependencies between different build activities?

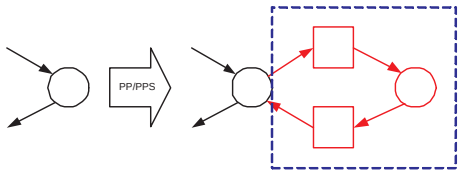

Figure 17: Protocol/projection inheritance-preserving rule PP/PPS

9 Inheritance of Workflows

W.M.P van der Aalst developed a workflow language based on Petri nets to describe a workflow [vAvH02]. Together with T. Basten, he developed rules for the inheritance of workflows [AB97, Bas98, AB99].

Van der Aalst and Basten identified four different notations of inheritance concerning workflow nets: *protocol inheritance*, *projection inheritance*, *protocol/projection inheritance* and *life-cycle inheritance*.

In [Aal99] projection inheritance is defined based on abstraction:

If it is not possible to distinguish between x and y when arbitrary tasks of x are executed, but when only the efforts of tasks that are also present in y are considered, then x is a subclass of y with respect to projection inheritance.

This means, if we hide the new parts of the sub–process x, and then the behavior of the remainder of x is exactly like y, we have projection inheritance.

Protocol inheritance is defined based on encapsulation [Aal99]:

If it is not possible to distinguish x and y when only tasks of x that are also present in y are executed, then x is a subclass of y.

This means, if we block all tasks present in x but not in y and the behavior of x is still the same as y, we have protocol inheritance.

Protocol/projection inheritance is the most restrictive form of inheritance. It combines protocol and projection inheritance. Life–cycle inheritance is the most liberal form of inheritance, protocol and/or projection inheritance implies life–cycle inheritance. The following inheritance–preserving transformation rules are formally proofed in [AB99].

Protocol/projection inheritance–preserving rule. There is one transformation rule confirming to protocol/projection inheritance, PP/PPS (see figure 17). New transitions and places are added to the workflow net in such a way, that tokens are only temporary removed from the place in the original net. The added subnet may consist of any structure, as long as it is guaranteed that any token taken from the original place will return finally and no tokens remain in the subnet. The PP/PPS rule adds additional behavior to the workflow.

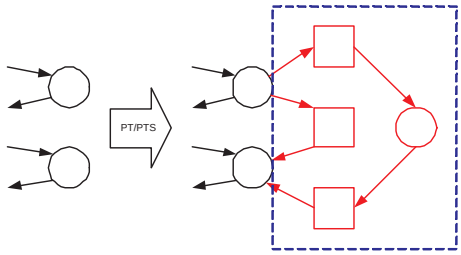

Figure 18: Protocol inheritance-preserving rule PT/PTS

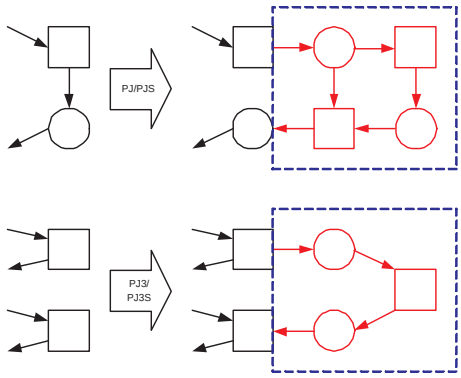

Figure 19: Projection inheritance-preserving rules PJ/PJS and PJ3/PJ3S

Protocol inheritance–preserving rule. The protocol inheritance-preserving transformation rule PT/PTS is shown in figure 18. The added subnet removes tokens from places of the original net and places them back at other places in the original net. The added subnet may consists of any structure, as long as it is guaranteed that any token taken from the original net will finally return and no tokens remain in the subnet. Other requirements are that all new tasks consuming tokens from the original net should not appear in the original net and that the route via the subnet does not create new states in the original net. The PT/PTS rule allows the addition of alternative behavior for the workflow.

Projection inheritance–preserving rules. There are two transformation rules confirming to projection inheritance, PJ/PJS and PJ3/PJ3S (see figure 19). The subnet inserted using rule PJ/PJS consumes tokens fired by the original task and places them finally back to the original target place. The added subnet may consists of any structure, as long as it is guaranteed that any token fired by the original task will finally be placed in the original target place and no tokens remain in the subnet. The PJ/PJS rule allows the addition of a new subnet in a workflow, replacing an existing connection.

The subnet producible by rule PJ3/PJ3S takes additional tokens generated by a task from the original net and finally places them back as new input places for a task in the original net. The added subnet may consists of any structure, as long as (1) the execution of the firing task of the original workflow is always follow by the execu-

tion of the receiving task of the original workflow, (2) the activation of the subnet via the firing task of the original workflow is always followed by a state which marks the input places of the receiving task of the original workflow in the subnet and (3) no tokens remain in the subnet after firing the receiving task of the original workflow. The PJ3/PJ3S rule allows the addition of parallel behavior in a workflow.

Further readings

- The inheritance–preserving rules have been applied in a paper covering a public to private approach [vAW01].
- Further considerations covering different problem domains can be found in [Aal99].

10 Further ideas

References

[Aal99] W.M.P. van der Aalst. Inheritance of Workflow Processes: Four problems - One Solution? In F. Cummins, editor, *Proceedings of the Second OOPSLA Workshop on the Implementation and Application of Object-Oriented Workflow Management Systems*, pages 1–22, Denver, Colorado, 1999.

[AB97] W.M.P. van der Aalst and T. Basten. Lifecycle Inheritance: A Petri-net based approach. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997, volume 1248 of LNCS*, pages 62–81, Berlin, 1997. Springer-Verlag.

[AB99] W.M.P. van der Aalst and T. Basten. Inheritance of Workflows: An approach to tackling problems related to change. Computing science reports 99/06, Eindhoven University of Technology, Eindhoven, 1999.

[AGL] IBM Aglet Homepage. `http://www.trl.ibm.com/aglets/` (March 25, 2004).

[ASU00] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers*. Addison–Wesley, Reading, Massachusetts, 2000.

[Bas98] Twan Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1998.

[BEA03] BEA Systems, IBM, Microsoft, SAP, Siebel Systems. *Business Process Execution Language for Web Services Version 1.1*, May 2003.

[Bog99] Marko Boger. *Java in verteilten Systemen*. dpunkt.verlag, Heidelberg, 1999.

[BPM04] BPMI.org. *Business Process Modeling Notation*, 1.0 edition, May 2004.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1990.

[CDHG00] Qiming Chen, Umesh Dayal, Meichun Hsu, and Martin Griss. Dynamic-Agents, Workflow and XML for E-Commerce Automation. In K. Bauknecht, S. Kumar Madria, and G. Pernul, editors, *Electronic Commerce and Web Technologies: First International Conference, EC-Web 2000, volume 1875 of LNCS*, pages 314–323, Berlin, 2000. Springer-Verlag.

[Cor95] A. Corradini. Concurrent Computing: From Petri nets to Graph Grammars. In *Proceedings of the Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, 1995.

[DHL90] Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. Organizing long-running activities with triggers and transactions. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 204–214, New York, 1990. ACM Press.

[DKRR98] Hasan Davulcu, Michael Kifer, C. R. Ramakrishnan, and I. V. Ramakrishnan. Logic Based Modeling and Analysis of Workflows. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 25–33. ACM Press, 1998.

[EMC+01] H. Ehrig, B. Mahr, F. Cornelius, M. Große-Rhode, and P. Zeitz. *Mathematisch–strukturelle Grundlagen der Informatik*. Springer–Verlag, Berlin, 2nd edition, 2001.

[EP02] Katrin Erk and Lutz Priese. *Theoretische Informatik*. Springer-Verlag, Berlin, 2nd edition, 2002.

[Gro03] Object Management Group. *UML 2.0 Superstructure Final Adopted specification*. Object Management Group, 2003.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[HS98] Michael N. Huhns and Munindar P. Singh. Workflow Agents. *IEEE Internet Computing*, pages 94–96, July 1998.

[Jäg] Michael Jäger. Script: Compilerbau – eine Einführung. http://hera.mni.fh-giessen.de/~hg52/lv/compiler/skript/compilerskript.pdf (March 25, 2004).

[KEP00] Gerhard Knolmayer, Rainer Endl, and Marcel Pfahrer. Modeling Processes and Workflows by Business Rules. In W. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies, volume 1806 of LNCS*, pages 16–29, Berlin, 2000. Springer-Verlag.

[KM97] Christoph Klauck and Heinz-Jürgen Müller. Formal Business Process Engineering based on Graph Grammars. *International Journal of Production Economics*, 50:129–140, 1997.

[Kop] Herbert Kopp. Script: Compilerbau. http://www.informatik.fh-wiesbaden.de/~weber/comp/kopp/co_buch.pdf (March 25, 2004).

[LMT95] Michael Löwe, Jürgen Müller, and Gabriele Taentzer. *Einführung in die Theoretische Informatik: Graphersetzung*. Universität Bremen, 1995.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

[Mil93a] Robin Milner. Elements of Interaction. Turing Award Lecture. *Communications of the ACM*, 36:78–89, January 1993.

[Mil93b] Robin Milner. The polyadic π–Calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246, Berlin, 1993. Springer-Verlag.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, Cambridge, 1999.

[MLL97] M. Merz, B. Liberman, and W. Lamersdorf. Using Mobile Agents to support Interorganizational Workflow. *Applied Artificial Intelligence*, 11:551–572, September 1997.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I/II. *Information and Computation*, 100:1–77, September 1992.

[MS99] John-Jules Ch. Meyer and Pierre-Yves Schobbens. Formal Models of Agents: An Introduction. In John-Jules Ch. Meyer and Pierre-Yves Schobbens, editors, *Formal Models of Agents: ESPRIT Project ModelAge Final Workshop, volume 1760 of LNCS*, pages 1–7, Berlin, 1999. Springer-Verlag.

[MS04] Harald Meyer and Hilmar Schuschel. Requirements on a Planner for Workflow Modelling Languages. In *IPI/BPT*, 2004.

[NB02] John Noll and Bryce Billinger. Modeling Coordination as Resource Flow: An Object-Based Approach. In *Proceedings SEA '02*, Cambridge, 2002.

[OYP03] Bart Orriëns, Jian Yang, and Mike P. Papazoglou. A Framework for Business Rule Driven Web Service Composition. In *Conceptual Modeling for Novel Application Domains, volume 2814 of LNCS*, pages 52–64, Berlin, 2003. Springer-Verlag.

[P] P language homepage. http://blrc.edu.cn/research/p/ (April 13, 2004).

[Par01] Joachim Parrow. An Introduction to the π–Calculus. In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Bonn, 1962.

[PRO] PROGRES homepage. http://www-i3.informatik.rwth-aachen.de/resea (March 25, 2004).

[Roz97] Grzegorz Rozenberg, editor. *Handbook of graph grammars and computing by graph transformations, Volume 1: Foundations*. World Scientific, 1997.

[Roz99a] Grzegorz Rozenberg, editor. *Handbook of graph grammars and computing by graph transformations, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[Roz99b] Grzegorz Rozenberg, editor. *Handbook of graph grammars and computing by graph transformations, Volume 3: Concurrency, Parallelism, Distribution*. World Scientific, 1999.

[RS97] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages, Volume 3: Beyond words*. Springer-Verlag, 1997.

[Sch00] Andy Schürr. *PROGRES for Beginners*. RWTH Aachen, Lehrstuhl für Informatik III, 2000.

[Seb99] Robert W. Sebesta. *Concepts of Programming Languages*. Addison–Wesley, Reading, Massachusetts, 4th edition, 1999.

[Set96] Ravi Sethi. *Programming Languages: Concepts & Constructs*. Addison–Wesley, Reading, Massachusetts, 2nd edition, 1996.

[SO00] Wasim Sadiq and Maria E. Orlowska. On business process model transformations. In A.H.F. Laender, S.W. Liddle, and V.C. Storey, editors, *Conceptual Modeling - ER 2000: 19th International Conference on Conceptual Modeling, volume 1920 of LNCS*, pages 267–280, Berlin, 2000. Springer-Verlag.

[SSO01] Shazia Sadiq, Wasim Sadiq, and Maria Orlowska. Pockets of Flexibility in Workflow Specification. In *Conceptual Modeling - ER 2001 : 20th International Conference on Conceptual Modeling, volume 2224 of LNCS*, pages 513–526, Berlin, 2001. Springer-Verlag.

[SW92] Andy Schürr and Bernhard Westfechtel. Graphgrammatiken und Graphersetzungssysteme. Technical Report Aachener Informatik-Berichte Nr. 92-15, RWTH Aachen, 1992.

[TS85] Jean-Paul Tremblay and Paul G. Sorenson. *Compiler Writing*. McGraw-Hill Book Company, New York, 1985.

[vA] W. M. P. van der Aalst. Pi calculus versus petri nets: Let us eat "humble pie" rather than further inflate the "pi hype". `http://is.tm.tue.nl/research/patterns/download/pi-hype.pdf` (May 31, 2005).

[vAtHKB00] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. Technical Report BETA Working Paper Series, WP 47, Eindhoven University of Technology, 2000.

[vAvH02] Wil van der Aalst and Kees van Hee. *Workflow Management*. MIT Press, 2002.

[vAW01] W. M. P. van der Aalst and M. Weske. The P2P approach to Interorganizational Workflow. In K.R. Dittrich, A. Geppert, and M.C. Norrie, editors, *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01), volume 2068 of LNCS*, pages 140–156, Berlin, 2001. Springer-Verlag.

[Wik] Wikipedia - The Free Encyclopedia. `http://www.wikipedia.org` (March 25, 2004).

[Wir96] Niklaus Wirth. *Grundlagen und Techniken des Compilerbaus*. Addison-Wesley, Bonn, 1996.

[Woo02] Michael Wooldridge. Intelligent Agents: The Key Concepts. In V. Maík, O. Tpánková, H. Krautwurmová, and M. Luck, editors, *Multi-Agent-Systems and Applications II : 9th ECCAI-ACAI/EASSS 2001, AEMAS 2001, HoloMAS 2001, volume 2322 of LNCS*, pages 3–43, Berlin, 2002. Springer-Verlag.