Infosys®
POWERED BY INTELLECT
DRIVEN BY VALUES

Win in the flat world

Technical Report

A Work Allocation Language with Soft Constraints

Christian Stefansen, Sriram Rajamani
Microsoft Research Lab India

Parameswaran Seshan
Infosys Technologies Limited

Abstract

This report defines a workflow allocation language with hard and soft constraints and explains the requirements and constraints that led to its current design. The language can be used with most business process and workflow languages. A high-degree of parameterization allows the language to be used with virtually any process languages.

March 2008

Contents

1 Introduction

This report describes a work allocation language that can be used with most business process and workflow languages. The language is parameterized over most functions and takes a small user database query sublanguage as a parameter. This high-degree of parameterization allows the language to be used with virtually any process languages and any conceivable legacy system, while guaranteeing polynomial performance, expressiveness, and uniform syntax. This report describes the example and patterns catalogues that lead to the language design, explains the semantics of the language, gives on overview of the implementation, and evaluates the language in terms of the resource allocation patterns proposed by van der Aalst et al. [1].

A salient feature of the language is its ability to express soft constraints. Soft constraints enable process designers to specify a wide range of organizatorial soft goals and preferences that the system takes into account when allocating tasks to resources.

1.1 Contributions

This report motivates and defines a declarative, generic allocation language. The distinctive features of the language are:

- it supports both hard constraints and soft constraints,
- it is pluggable with any (reasonable) business process language,
- it is pluggable with any organization context and legacy platform, and
- it runs in P-time (provided that the user-defined functions do so)

Our main contributions are the following:

- A work allocation language definition that is orthogonal to the host language. The language takes on an aspect-like flavor, in the sense that it is possible to disable the allocation aspect and still run the business process (which then would simply allow anybody to carry out any task). Similarly, the allocation can be run without being hosted in a process language. We argue that a natural way to contain the complexity of business processes without the shortcomings of using aspect-oriented programming with a general-purpose language.
- The introduction and uses of soft constraints in a BPMS (Business Process Management System).
- A language that demonstrates an attractive middle ground between allocation using full optimization and static or no scheduling. The allocation can be done in polynomial time, but yet is able to express a wide range of heuristics to improve workflow performance and resource utilization.
- Because allocation is expressed in a a simple, well-defined DSL, it is now possible to carry out performance analysis and what-if analyses on workflows. The language also enables infeasible allocation rules to be detected statically using a simple model checker.

2 Background and motivation

Computer-orchestrated business processes are increasingly playing a direct role in companies' organization of work. Thus, computer-orchestrated processes now commonly involve both human resources and computer resources. A widely accepted architecture is to have a business process orchestration engine that runs the process by calling upon human resources and computing resources to perform the actual tasks. Tasks can often be handled by many different resources; this is particularly often the case

for tasks handled by human resources. Therefore the process orchestration engine must decide in negotiation with the human resources who of the eligible resources ultimately carries out the task. The assigning of a task to a resource is referred to as *allocation*.

The implied architecture is essentially SOA (*service-oriented architecture*) in that all resources whether they be human or computers or unknown are simply regarded as services that can be called upon do complete process in a networked system. This architecture is an important change from early workflow systems that would often merely *guide* the human workers in carrying out the process themselves rather than actually *orchestrate* the process.

While dedicated systems and professional workflow systems have some support for allocation [1], popular process language used for orchestration, including WS-BPEL [2] and BPMN, do not. Hence since such languages are now taking over the role of dedicated workflow products to achieve a broad SOA integration, there is an increasing need for of allocation facilities. Some initiatives address this (e.g. BPEL4People [3]), but they remain in an embryonic stage.

Allocating tasks to humans is inherently more complex than allocating to computer resources. Allocation does have some similarities with the scheduling done in a multiprocessor system, but in the case of allocation the processors are rarely homogenous. In this way the problem area seems closer related to Grid scheduling, but there is yet more complexity, because in addition to having multiple, changing attributes (e.g. roles, location, skills, availability, time constraints, etc.), humans have personal preferences, they possess runtime knowledge, and they may—for entirely exogenous reasons—choose to override the allocation rules at runtime.

Workflows typically specify the dependencies and constraints governing a set of tasks that need to be carried out to reach some goal with control flow and dataflow being the primary focus. We say ”typically” because workflows have also been expressed without an explicit control flow in terms of logical rules and goals in the style of Prolog [4, 5] – more on this shortly. In addition, a workflow specification can govern allocation, resource access, security, distribution, transactions, etc.

But human-intensive work very often deviates from the prescribed process. Sometimes because the human workers have information that is not captured in the model, sometimes because the model is insufficient or does not adequately cover all failure types, and commonly because humans make conscious, reflected violations e.g. to speed up processing in cases where where the process description focuses too narrowly on perfect compliance.

Today, where a process engine orchestrates the process rather than guides it, deviations cannot be ignored. Instead the process engine must be built with flexibility in mind. This is particularly pertinent when we consider allocation. Put differently workflows are not the same as programs, and one defining difference that has been suggested recently [6] is that workflows are *flexible*. Flexible can mean that they are modified at runtime as suggested by Avinash et al. or that they are described in a model that leaves a high degree of freedom to the runtime system as suggested by Borch and Stefansen [7]. By analogy a Java program might contain the statements `x = 5; y = 4;`. This program overspecifies slightly because it enforces a particular sequential order on two statements that are entirely independent. For some programs this is not a problem, although substantial amounts of research has been devoted to program rewriting, optimizing compilers, out-of-order execution pipelines in processor design, out-of-order instruction emission (see e.g. Tomasulo et al. [8]), and programming languages that only specify a minimal sequential bias (e.g. Haskell, dataflow languages, Datalog/Prolog). Sequential constraints is only one of many examples of overspecification. For workflows, however, it is prudent to assume that *any* overspecification will be a problem more often that not, when human resources are involved. This issue is treated in more detail in a series of papers by Borch and Stefansen [7]; in this report we focus on avoiding overspecification in allocation rules.

However, just specifying the mimimal number of constraints also leads to problems: allocation constraints represent a wide spectrum of specifications ranging from strict constraints to mere guidelines that can be followed at the worker's leisure. E.g. if a process has an activity called “Expense approval” could be restricted to users with the role “Manager”. Such a constraint clearly should not be violated, as this would undermine the purpose of the approval step. On the other an activity such as “Replenish printer cartridges” might be considered tedious, and thus allocated to qualify personnel on a rotation

basis (*round robin*). The latter allocation strategy represents an organizatorial *soft goal*, which could in the case be "rotate tedious tasks between qualified workers to achieve a sense of fairly and variation and keep workers happy". This is a laudable goal to but if the company is experiencing peak load this goal must temporarily yield to the more important concern of reaching business goals (e.g. response time vis-a-vis out customers). This finer granularity of rules is another source of complexity in human-task allocation. Some rules cannot be violated, some can. Some of those that can take priority over others.

This project was done in colloboration with Infosys Technologies Ltd., India. Infosys focuses WS-BPEL and BPML, where the former is gradually gaining importance over the latter. To enable its clients to support their business processes consistently, Infosys has built its own BPM platform named *PEAS*. Typical applications include sales support, banking and *business process outsourcing* (BPO). The architecture of the PEAS platform is introduced in more detail in Section 5.3.

In short there is a need to augment existing business process languages with allocation support and there is a need that allocation rules can be flexible to accomodate soft goals while avoiding overspecification – or simply because humans are involved.

3 Requirements and design space

Given that allocation is clearly needed and not yet present in current standards and systems, a language for specifying allocation rules would seem appropriate.

Allocation rules can depend on the organization model, activity properties, runtime info, process state, activity state, observables, history, and legacy systems or other external calls. Some important requirements can be derived immediately from the business context and the architectural context:

- Because the environment (user database, log formats, legacy interfaces) is different in every organization, the functions that read these must be externalizable. To be able to write something like `user = whoLastDid(task)`, we need to be able to define an external function `whoLastDid` along with its argument and return types.
- It is common to navigate through the user database to express constraints, e.g. `user.manager.location = "Europe"`. The syntax of the sub-expression `user.manager.location` is defined in a sublanguage depending on the type of user database being used (e.g. LDAP, XQuery, SQL). The language should allow such pluggable, independent sublanguages in some form.
- The work allocation language needs to interface both with systems that are case sensitive and case insensitive. Hence a case-sensitivity switch is the best option, but case sensitivity should be the default.
- The work allocation language must be able to interface with any (reasonable) BPMS because different organizations will have different (even proprietary) systems.
- The language must work in tandem with any (reasonable) process model (e.g. BPEL, BPMN, EPC, Petri net-based model) that the BPMS uses. This means that it cannot be tied intimately to one language or paradigm.
- Flexibility is crucial in workflows. Therefore the language must make it possible to specify very wide constraints within which a good solution can be found using runtime information. The prototype should help optimize allocation decisions by identifying and proposing candidate allocations that satisfy as many (of the violable) rules as possible in the given situation.
- The language should simplify static formal analysis, including typing and model checking, e.g. by inferring types and not having general recursion. Keeping the language very simple also enables performance analysis, capacity planning, and *what-if* analyses on workflows with resources and allocation rules. This is a substantial improvement over previous systems, where the lack of integration made performance analysis a cumbersome job requiring integer programming skills to recast the workflow as an optimization problem.

3.1 Workflows as traces

Before considering the design of the language it is useful to frame the problem in slightly more general terms.

It is common to think of workflows in terms of control flow of parallel processes, and then secondarily about dataflow, security, resource allocation, etc. In fact, since workflows are used by many different people (planners, business analyst, process rengineering consultants, knowledge workers, etc.) workflows are often manipulated through different abstract views of the same complete workflow description.

One way of seeing a workflow is as a specification of allowable *traces*, that is sequences of activities or events. If we consider the stylized workflow $A; (B \mid\mid C); D$, it specifies that "$ABCD$" and "$ACBD$" are the only allowable traces. This is simplified, of course, because a real-world workflow would also contain data flow, policies, allocation rules, etc., but even when we add more detail to the specification, the fundamental idea remains that workflows specify allowable and disallowable traces.

In fact, when we compare a workflow with allocation rules, such as $A[intern]; (B[CEO] \mid\mid C[]); D[]$ to one that does not specify roles $A[]; (B[] \mid\mid C[]); D[]$, the former can be seen as a *refinement* of the latter. That is, we can write

$$A[intern]; (B[CEO] \mid\mid C[]); D[] \preceq A[]; (B[] \mid\mid C[]); D[]$$

to say that everything allowed by the refined workflow is also allowed by the abstract workflow (but not necessarily the other way around). This example suggests that allocation rules are somehow a further specification that can be removed if necessary, which further suggests that allocation rules could be made separate from the control flow specification. Suppose instead we had written

$$A; (B \mid\mid C); D \;\&\&\; A[intern], B[CEO]$$

to mean exactly the same as $A[intern]; (B[CEO] \mid\mid C[]); D[]$. This factors out the allocation rules so that they can be written separate from the control flow part of the workflow. The two still share names of activities and pieces of data, and this is what allows the separation. Here are some reasons why this separation might be useful:

- Each user can add his or her own allocation policies on top of the ones in the workflow
- More generally, allocation rules can be composed at several levels
- Policies can be applied to several workflows without having to rewrite/re-insert them each time
- The workflow code becomes easier to factor and thus could become easier to understand
- Allocation rules can be written separately from the workflow (provided that the names are agreed upon a priori)
- The control flow (and possibly data flow) part can be used as a public specification without revealing the internals of the process.

If we go further into technical detail, we could imagine a function Tr mapping the abstract syntax tree of a workflow to the set of its allowable traces. For out example this would give

$$\begin{aligned}
Tr[[A; (B \mid\mid C); D]] &= \{A[]B[]C[]D[], A[]C[]B[]D[]\} \\
Tr[[A; (B \mid\mid C); D \;\&\&\; A[intern], B[CEO]]] &= \{t \mid t \in Tr[[A; (B \mid\mid C); D]] \; and \\
&\quad (A[x] \in t \implies x = intern) \; and \\
&\quad (B[y] \in t \implies y = CEO)\}
\end{aligned}$$

or more generally

$$Tr[[spec1 \ \&\& \ spec2]] = Tr[[spec1]] \cap Tr[[spec2]].$$

The point is that this way of composing workflows allows us to start from a very general specification of the workflow and gradually restrict it more and more by applying policies, rules etc. We can also notice that as long as $\&\&$ is well-defined we can compose not only allocation rules, but other restrictions as well.

If we see workflows as specifications of allowable traces, the crux of the matter is really: what level of detail is in these traces? What are the elements of a trace? Put differently, what granularity of events is recorded?

On a very abstract level, traces simply contain the names of the tasks that were completed. If the process `A;(B || C);D` is completed we then have a trace containing either $< ABCD >$ or $< ACBD >$, provided that the system has an interleaving semantics. Had we included dataflow, the process might have been `b1,b2 : bool in b1 := A(5); b2 := B() || C(b1));D()`, and the traces would capture the data environment in some way. Naturally, if we would like to specify that `b2 = false` is unacceptable, then we would need the refined trace to determine if a violation had taken place or not.

For allocation we can imagine that re-allocation (taking an already allocated task and allocating it to a different resource) is possible. In that case the language can express constraint on re-allocation and the traces denoting the workflow would contain re-allocation events.

3.2 Re-allocation

If the workflow system allows re-allocation, the allocation can be changed at any point before an activity's completion even after the activity has been started. When an activity is completed, its completion along with its associated data and the resource, who completed it, are recorded in the process trace as a witness of what ultimately happened. Hence allocation can be seen as two different problems: (1) how are resources allocated, re-allocated and negotiated before and during the execution of the process, and (2) what resources ultimately completed what activities.

Addressing (1) requires an underlying state machine for activities. E.g. some activities can be first offered to a resources, then decline, offered to another, started, suspended, etc. Other activities only permit a simple invocation à la request/response. Yet other activities could be allocated by auction, in which case an auction process would determine the allocation based on received bids. This gives rise to a negotiation layer where the policies and constraints governing the runtime negotiation rely intimately on the state machine of each activity. Since each task could use different runtime negotiation protocol, the work allocation language needs to be able to express constraints on these negotiations (i.e. constraints on the traces containing negotiation events).

Addressing (2) on the other hand requires no knowledge of the activity state machine, but only of the process description, the process trace, and a *static* data model of resources and their properties. In this sense, allocations specified as (2) express properties that can be verified *a posteriori* by examining a trace containing *only* information of activity completions and what resources completed the activity.

Even if we adopt the simple model (1) and do not support re-allocation in the language, re-allocation can still be allowed in the system. The important proviso is that the state of a process *including* the state of all allocations can be persisted or recreated easily. Hence there can be an arbitrarily complex runtime negotiation layer; the work allocation language will only be able to specify constraints on who ultimately can complete the task.

Abstractly, allocation is a map from the processes, the resources and their properties, the workflow engine's runtime data, and possibly external data to allocations or negotiation events.

It is important to notice that in (2) both the allocation engine *and* the resources can produce negotiation events, whereas in (1) the allocation engine just issues an allocation (which may or may not be registered in the event trace), which must be reflected in the eventual completion event recorded when the process engine receives a completion notification from a valid resource.

In both cases a reasonable strategy is to rerun the allocation if anything changes. That could be that one of the arguments to the allocation function changes (say, a user gets a new role or logs off the system) and in the case of (1) also if a negotiation event is received. The allocation engine should also rerun when a process with no prior allocation state is loaded into the system. Notice that in (1) this is all independent from the work allocation language which will not concern itself with rules for re-allocation or if re-allocation is done at all. If we follow (2) on the other hand, the language would enable us to specify that some task should not be re-allocated if when changes in the context were observed. It would also enable us to specify that certain negotiation events were disallowed for some tasks (say, disallow "decline" events for the task "do laundry").

The system then has at least three available strategies, which will all work with the model (1) language:

1. Never re-allocate
2. Activities can be re-allocated locally (say, by humans), but the system does not respond by global re-allocation.
3. Re-allocate whenever there is a transition in the system whether it be a human reallocation, a transition or any other state change.

The trace perspective on workflows allows us to precisely define the scope: the language will exclusively deal with creation patterns, i.e. rules about who can and cannot do a task, and *not* how eligible resources negotiate and re-allocate at runtime. The adherence to the specified allocation rules can be verified by looking at an trace containing activities only and no negotiation events.

Although this decision would seem to make it impossible to express all pull patterns (i.e. allocation where the resource takes part in the decision), this is not entirely the case. Whereas we have though of a resource as only one human worker or one computer so far, resource could simply be queues, and there could then be multiple resources subscribing to multiple queues. This introduces another level of indirection that somewhat allows the architecture to accomodate pull patterns even if the language does cannot express that type of allocation rules.

3.3 Purist paradigms for runtime negotiation

It is enticing to try framing workflow and allocation in one comprehensive paradigm. This section contains a few ideas on how to do that. Since the language described here does not support rules about runtime negotiation, the reader is safe to skip this section.

1. Everything in the workflow and in the allocation rules is a constraint, and the runtime negotiation that goes on can be expressed as CRUD (Create, Read, Update, Delete) functionality on sets of constraints. Whenever the constraint set is changed, re-allocation is performed.
2. Everything is a process, and runtime negotiation is really just a subprocess of the activity. The allocation rules constrain these subprocesses and the runtime optimization is about resolving the remaining non-determinism. For the activities we can have:
   (a) The same type of state machine for all activities
   (b) No state machine – only one-step nonnegotiable allocation
   (c) Different state machines based on negotiation protocol
3. Runtime negotiation is orthogonal to allocation rules. The only concern of allocation should be the specification of who is allowed to do the task when all is said and done. How non-determinism is resolved at runtime is irrelevant. The rules can specify queue-based, history-based, round robin-based, etc. types of allocation, but whether any remaining non-determinism is solved through push, pull, re-allocation or escalation is not relevant, because these are the internals of the runtime

system.
In other words rules are about *what* allocations are acceptable and does not speak about *when* or *how* allocations are made.

4. Allocation is an auction. There is a meta-process governing the auctioning off of each task. The allocation rules specify the auction protocol, the resources place bids (to take or not take the task), and the allocation engine finds the most desirable (i.e. utility optimizing outcome) based on the bids and the auction model. Note that this can be seen an instance of the paradigmn where everything is a process.

3.4 Allocation time

When an activity can be started we say that it is enabled. E.g. in the process expression `a ; (b || c) ; d` (`a`, then `b` and `c` in parallel, then `d`) the activity `a` is the only enabled activity. After `a` has been completed, both `b` and `c` are enabled.

Allocation can happen at three conceptually different times:

1. Early allocation happens before the task is enabled (long horizon) and enables the system to predict/guess bottlenecks early and avoid them through different allocation. The system may even speculatively allocate beyond branches (even if outcome of the branch is not yet known), which of course works best in conjunction with the ability to re-allocate. Some systems make a fixed battle plan (that is, allocation) for the entire workflow when it starts, but not being able to modify the allocation along the way seems to rigid for most of today's changeable workflows.

2. Allocation on enablement allocates an activity exactly when it becomes enabled. This is simple to model, to implement, and to understand for humans interacting with the system.

3. Late allocation allocates as late as possible, i.e. if all suitable resources still have items in the queue, the allocator might as well wait until someone's queue is (almost) empty, and only then allocate the activity. This is particularly suited for systems where re-allocation is not used or desired (people may prefer stable queue rather than queue where activities fly in and out every time the system transitions). Late allocation can be problematic for human workers because they only see a partial view of the activities in their queue.

Combinations of the three strategies are also possible. E.g. one can allocate early to give human workers a rough idea of what needs to be done, but then re-allocate continuously to make sure that the optimal allocation is being chosen. To make the queues more stable the system could limit late re-allocation to cases where a resource is idle.

The language designed here can support all three modes or a combination thereof, because the allocation rules can simply be re-evaluated to obtain a new allocation when desired by the allocation engine. In other words this is orthogonal to the language design *per se*.

3.5 Soft constraints and rule scope

As indicated previously workflows involving humans need to be flexible, but a allowing everything does not capture that among many allowable courses of action, some are preferable. To alleviate this we introduce the idea of soft constraints in the language. Soft constraints are rules that the allocation engine should try to satisfy, but if necessary it can pick an allocation not satisfying the constraint. The goal of the allocator is then to fulfill as many soft constraints as possible. Although we could just specify for each constraint whether it is hard or soft, more flexibility can be achieved by attaching a weight n to each constraint. For each allocation a soft constraint gives n points if it is satisfied and 0 if it is not. The allocations can then be preference ranked by their sum of points given for satisfying soft constraints.

Example of soft constraints include *round robin*, *prefer least loaded resource*, *prefer shortest queue*, *prefer finance officers to manager*, etc. Some of these smack of scheduler strategies and indeed the

question arises: what will the scheduler do if a number of potential allocations all give the same amount of points? How will it choose?

Conceptually we can think of soft constraints being attached not just to a tasks, but also to a scope, a set of activities, a process, a resource itself, or they can be inserted on several levels to override generic rules and policies with more specific ones. Soft constraints can also be used to express overall policies in the workflow engine. In this way the scheduler itself can been understood as a set of soft constraints that will be used if all the other constraints leave several options with the same preference level. A typical scheduler preference would be a greedy heuristic to minimize the make-span (the workflow completion time), but a simple scheduler could just break ties by random choice.

A very sophisticated scheduler could leverage leverage runtime statistics from the current and other processes to improve the allocation optimization. Runtime statistics could include average completion time pr. task type pr. agent, inferred probability of failure or delay, etc. All these statistics will result in more soft constraints that the scheduler can use in conjunction with the user-specified ones. Indeed the compilation and use of such statistics is the subject of the next part of the project, in which this report is the first part.

Of course the complexity is rather daunting as the scheduler would potentially need to examine all running processes and all activities contained therein. In this report we narrow the focus somewhat to only consider rules that are attached to one activity, but there is not inherent limitation in the language design to prevent rules attached to scopes, entire processes, resources or the scheduler. This is a key advantage of the compositionality of the soft constraints, achieved by making them simple weighted sums.

3.6 Advanced features for full optimization

A number of other features can be included in the work allocation language to make it into a complete staff planning tool. Most of these facilities render the allocation function NP hard and therefore they are not integrated in the current language design. In the following list, the design choice is *emphasized*.

1. Allowing inter-activity dependencies.
   (a) Constraints cannot reference other activities
   (b) *Constraints can only refer to guaranteed previous activities*
   (c) Constraints can refer to each other freely within one instance (hence variables can be a value or unset).
   (d) Constraints can refer to each other between instances of same process
   (e) Constraints can refer to any activity in any process instance
2. Modeling arrival and departure of agents
   (a) Agents are statically present
   (b) *Agents rarely arrive/depart; this is handled manually*
   (c) Agents often go in and out of office, or sign off the system. Reallocation is made accordingly. (This opens up for gaming because agents can plan to be offline deliberately when an undesired task is being scheduled.)
3. Can activities be scheduled to be worked on at particular times?
   (a) *No, no time scheduling*
   (b) Yes, but only as a matter of checking feasibility
   (c) Yes, they strictly become calendar entries. This is important when several resources need to come together for an activity.
   (d) Yes, soft scheduled entries. Preferred time slots are given.

4. Exactly one agent pr. task or more?
   (a) One agent pr. task
   (b) *Several agents pr. task (work asynchronously on the task)*
   (c) Several agents pr. task (must work synchronously on the task)
5. What are resources?
   (a) *Only agents (humans and computers), i.e. entities capable of carrying out tasks.*
   (b) Any resource both human/computer or a constrained resource such as machine hours, production inputs, equipment, conference rooms, projectors, etc. (If this type of resources are allowed and time scheduling is used, full planning can be done.)

4 Analysis of patterns and examples

Having now established the general scope of the language we can examine the details of allocation constraints. The next step therefore is to identify the abstractions that the language must provide to express the allocation constraints that we need. To this end we (1) consider a large selection of examples that was compiled during the requirement analysis, and (2) review the existing material on resource allocation patterns [1, 9, 10], most notably by Aalst et al. [1].

4.1 Examples of hard constraints

A selection of hard constraints were deemed common and necessary; they can be found in appendix A.1.

4.2 Resource allocation patterns

The paper [1] describes the following types of resource allocation patterns (our adapted grouping):

Creation patterns (1-11) Specified with the workflow statically. Set constraints that leave some variation up to runtime.

Push patterns (12-20) Pertain to runtime negotiation and scheduling.

Pull patterns (21-26) Pertain to runtime negotation and scheduling.

Detour patterns (27-32)

Control flow patterns (33-35) Skip, Redo and Pre-Do (Should we have Undo too? If I want to step back and make a different choice in a branch then Redo is not good enough.)

Auto-start patterns (36-39) Autostart is just direct allocation to non-human resource? Piled Execution and Chained Execution seem to be sorting options in the user's client.

Security patterns (40-41) Security is about who can view what items and parts of the workflow in what state. This seems orthogonal to allocation.

Multiple resource patterns (42-43) Specification sets the constraints, decided at runtime what resources are actually involved.

Given our goal to support only rules about what resource ultimately can complete a task and no runtime negotiation, the patterns relevant to us are the creation patterns (patterns 1–11). The other groups of patterns pertain to runtime negotiation, architectural concerns, and sometimes concerns that are entirely orthogonal to allocation rules (e.g. security patterns, dynamic scheduling, soft goals, and client view options). Also some of the patterns are simply instances of more general patterns (as the paper does itself point out), but at the same time a more rigorous a general approach is missing leaving some interesting patterns undiscovered (see auction patterns below).

Another peculiar shortcoming of the patterns collection is the lack of combinations. The patterns collection is meant to be a first approximation to gauge the expressiveness of an allocation rule language, but it does not mention whether or how patterns can be combined and what limitation there might be in doing so. In general the paper seems like a first approximation in an area where more research should be done.

See appendix A.3 for examples of the creation patterns 1–11 listing the requirements on the language's integration with its environment that they give rise to.

In addition to the patterns from Russell*et al.* [1], the following patterns have been collected from the authors' research. The most commonly required patterns among those are: Separation of duties, Deferred allocation (based on data or runtime nomination), Case-based allocation (i.e allocating an entire case to one resource).

PEAS currently supports the following ways of allocating work to users

Direct allocation Direct, design-time allocation to a single resource based on the current value of a process variable[1] (e.g. `approver`).

Role-based Direct, design-time allocation to a single role.

Custom hook A Java method invocation that maps the activity state and process state to a single resource.

Location-based For user location one can specify *default*, *any*, or a list of locations that restricts the assignment of that work item to users playing a specified role in a particular locale.

Back-to-back execution Very similar to chained execution. If several activities in a proces are allocated to the same resource, the resource can tend to them in one go, without returning to the workflow activity list. Instead the screens/applications associated with each activity should be directly shown. In PEAS, back-to-back execution has a timeout and is configured globally pr. process. This pattern already exists in the PEAS platform and can be disregarded as it acts independently from the work allocation itself.

In addition to the currently supported patterns and the commonly required patterns mentioned above, the following allocation patterns must be supported in the new language:

Allocate to creator(= owner) The entire process is allocated to the resource who instantiated it.

Multiple roles An activity is allocated to several roles. A resources is needed having any of the roles mentioned.

Soft constraints Our research and previous research [7] strongly suggest a need to be able to specify that some rules are non-violable, while others can be ignored. The latter are referred to as soft constraints, and they carry an integer score. This score is used to break ties between the resources that satisfy the hard constraints.

Multiple resources An activity is allocated the several resources, who all of them need to collaborate on the work. The simplest acceptable model is just to specify for each activity how many resources n need to be allocated. These n resources can then be picked simply as the top-n from a resource list ordered by soft goal score.

[1] When processes run in the PEAS process engine, there are both process instance variables and application variables pertaining to the Java application.

Scalar properties Common properties such as location, role, or capability are binary in the sense that either a resource has the role or not. For skills, however, there may be several levels, e.g. (Java, 1–Novice), (C#, 2–Medium), (Python, 3–Expert). In such cases we conceivably want to be able to specify (a) a minimum skill level for some needed skill (e.g. Java) or (b) a soft preference for assigning the task to the resource with the highest skill level or (c) both.

PEAS allocates on enablement and does not re-allocate unless a restart occurs. It is assumed that the user database changes sufficiently seldom that manual administrator-forced re-allocation is an acceptable solution. Triggers can fairly simply be connected to the allocation engine, but other context information (e.g. external values) may not support triggers or subscriber/observer at all. Therefore using triggers only solves part of the problem.

Reallocation in PEAS deployments most often happens if (1) the nature of the case changes (e.g. minor fix turns out to be larger bug), (2) a resource goes offline, or (3) more resources are added to an activity to escalate it.

In addition to the allocation and reallocation requirements, it is interesting to identify and alleviate bottlenecks in processes. In our practical assessment, semantic deadlocks rarely occur in business processes. However, *approval bottlenecks* routinely occur. In these cases more resources can be added to an overflow group of resources who normally do not tend to approvals but who are able to take such tasks if the queues/expected completion times exceed a preset threshold.

4.3 Introducing preferences

So far we have only focused on hard constraints, but as implied previously soft constraints or preferences are highly desirable. See appendix A.2 for a catalog of soft constraints and how they could look in a language.

Preferences introduce order on breakable rules. This means any constraint C can be rendered as a preference specification by writing `[points] C` where the expression `points` should evaluate to an integer or a real. E.g. `[5] role = "Manager"`.

4.4 Other allocation schemes of interest

Overflowing When the queues of some people become too long, an overflow group can be used for allocation. This amounts to soft constraints based on queue length/expected average waiting time.

Timeout patterns If a task has not been started/completed before some relative or absolute deadline, branch off to a different part of the control flow or reallocate. This pattern will not be supported as we do not model time.

Minimize makespan A number of heuristics can be built into the scheduler. The soft constraints we have focused on here specifically cannot do full optimization because performance and simplicity have been prioritized.

Maximize skill refreshment [9]

Maximize skill match [9]

Timed auction

4.5 Summary

To express the patterns within our scope, we need user database lookups, access to process variables, process engine runtime information, process history, and external calls. We also need the ability to compose several patterns and to associate rules at different scopes. In particular rules must compose so that there is a straightforward semantics for two combined rules.

5 Architecture and integration

5.1 User database navigation sublanguage and external functions

Rather than marrying the work allocation language to one particular data model or one particular data query language, we want to allow a sublanguage to be plugged in to allow user database navigation and query. This is because resources will have different properties such as roles, groups, capabilities, managers, divisions, cost centers etc. An XQuery/XPath subset or XSDs would be reasonable candidate sublanguages, as would UML/OCL or SQL, but the tricky part is how to distinguish expression in the two languages when sublanguage expression occur as part of expressions in the work allocation language.

The simplest yet flexible solution which is what is used here is to specify lexical rewrites rules. The rules wrap all sublanguage constructs as unambiguous external calls.

The current implementation simply assumes that any expression that cannot be recognized in the main language belongs to the data language. E.g. assuming that all function calls start with # the parts of the expression `user.manager.role = "CEO" or user.location = #preferredLocation(task)` can easily be discerned, since anything not a integer, bool or string constant or starting with # can be assumed to belong to the data language. This does imply that spaces cannot be used in sublanguage expressions, but this has not been necessary thus far. In the given case the lexical rewrites rules would convert the expression to `#userExp("user.manager.role") = "CEO" or #userExp("user.location") = #preferredLocation(task)`

Other approaches such as enclosing those expression in in markers (e.g. `<@ ... @>`) have been considered and can easily be used in the current language design. However, the current design puts emphasis on keeping the syntax a simple as possible.

Expressing properties of users is central to allocation rules, yet – as we argued – best kept in an orthogonal sublanguage. But general function calls are also needed. We need to draw in information about queue sizes, workload, history etc. Again, we could do this either as part of the language, as a sublanguage or through external calls. External calls seem to be the best solution here for several reasons: first, many of the external calls are complicated functions operating on large sets of data; history is one example of this. Building in an entire history reporting language would tend to complicate the language too much and also would not provide much value because a preferable reporting language (e.g. SQL) is usually already available. Allowing the language to be plugged in as a sublanguage could work, but many of the functions involved are functions that are written once and used many times. Compare the expression `manager.role` to `whoLeastRecentlyDid(task)`. The first is something a process designer could write without much schooling and there is enormous variation in the user properties people want to use. The second represent a standard function of which there are not that many. Therefore is will be easier to simply write these functions when the allocation engine is being integrated with the legacy systems. Hence any other external data needed will be accessed through external functions.

5.2 Architectural challenges

Focusing only on creation patterns and not on runtime negotiation patterns as we have done here, can seem somewhat limiting. Specifically, we can only *push* a task to someone, never wait for the eligible resources to *pull* the task voluntarily. However, there are some architectural levels of indirection that can help alleviate this apparent shortcoming. We have assumed thus far that allocation is to a concrete resource, but we could also allocate to a queue. Then every resource could subscribe to many queue, and conversely queues could have many subscribers. The division of labor is then this: the engine *pushes* the task to a queue, but the resources subscribing to that queue can wait and *pull* the task from the queue at their leisure.

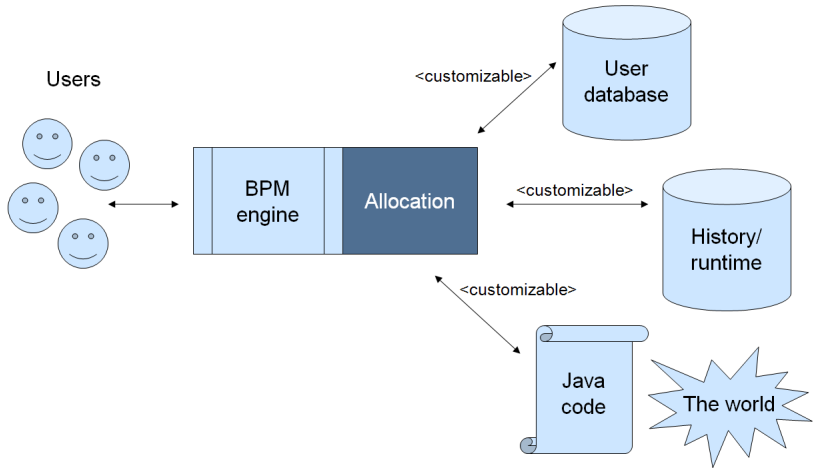
Also notice that the architecture – not the language – now decides how queue are handled, e.g. if resources can autonomously override task priorities, skip jobs in the queue, etc.

On a more general note, it is tempting to phrase allocation as part of a comprehensive SOA architecture. There are several ways of seeing it:

1. Allocation is a layer in the business stack. On top sit BPEL processes and below those sits the allocation layer arbitrating and negotiating with peers to make them take on the required activities. It is like the IP stack or the OSI model.
2. Allocation is just a service like everything else in a SOA world. A BPEL process sends info to the allocator, which negotiates with other peers on its behalf.
3. No special architecture needed, all is part of the same process specification. The business process is just an abstract view of what happens.
4. Allocation is an aspect, that is a concern in a program that can be toggled on and off.

Currently, the proposed allocation engine interfaces with the process engine by returning a list of allowable users after evaluating the constraints. In other words, what the process engine does with that list is out of the scope of this document. The allocation engine is tighly integrated with the process engine and not a stand-alone service although this approach would be interesting to follow.

Figure 1 The architectural context of resource allocation



5.3 Introduction to PEAS

PEAS is Infosys' proprietary platform for BPM that includes business process modeling, deployment, execution, and monitoring. Its architecture is multi-tiered, including (1) a SQL database tier (MySQL, Oracle, Microsoft SQL Server, etc.), (2) a process engine that can run as an independent server on its own or that can embed itself within a Java based web application, (3) a web application (Java-based) that acts a portal for the BPM platform with a browser based client tier, and (4) a tier enabling multi-mode remote communication with the process engine. Common interfaces to user data include LDAP, SQL/JDBC, XML/XSD, and SOA. The preferred development environment is Java/Eclipse.

PEAS modeler A graphical tool for building executable processes. The processes have to be block-structured as the tool exports to WS-BPEL/BPML. The graphical notation is a subset of BPMN/UML Activity Diagrams where only block-structured constructs are allowed.

PEAS process engine The Java server-side components that handle business process deployment and execution.

PEAS portal A Java-based web application that allows process administrators and analysts to monitor executing processes and view data about completed processes. The portal also enables human participants (users) to participate in their respective processes.

Customer application The customer's application that plugs in with the engine as well as with the customer's existing systems.

6 Language definition

This section defines the work allocation language in detail. First we describe the grammar and the set of rewrites that the are performed during lexical analysis. We then consider the type system before turning to the denotational semantics.

The language is parametrized over the set of operators, functions, and types. This means that a large number of extensions to the language can be done without re-working the core semantics. Thus the properties described here will hold for any valid set of operators, functions, and types that one might wish to use.

To specialize the core language to a concrete language the following need to be specified:

1. Syntactic rewrites (syntactic sugar)
2. The set of additional types, β
3. The operators/functions and their types (for infix operators their precedence should also be specified)

We first describe the core semantics and then in section 6.3 we show an example of how to incorporate new functions/operators and types into the language based on the language implemented in our prototype.

Figure 2 Work allocation language core grammar

rule	::=	(`pick` *const*)? (`where` ***exp***)? (`prefer` ***score-exp****)?
exp	::=	***exp*** *op* ***exp*** | *unary-op* ***exp*** | *value* | #*function* (***exp***, ..., ***exp***) | `user` | `task`
score-exp	::=	`[` ***exp*** `]` ***exp***

First consider the general grammar in Figure 2. Since every rule denotes a partial map from user and task to a score, the only allowed variables are in the language are `user` and `task`.

Notice that both operators and functions are outside the core grammar. Specifically, the following domains must be defined in a concrete instance of the language:

U
: The domain of user ids (note that this is not the set of users in the system; it is the domain of all valid users ids.)

T
: The domain of task names

Value
: A set of values (union over all types) containing at least `true` and `0`. In addition we require that $U \subseteq \text{Value}$ and $T \subseteq \text{Value}$.

Func
: A set of function/operator symbols

Any of the top-level clauses `pick`, `where`, and `prefer` can be omitted. In such case the default values are respectively: `pick 1`, `where true`, `prefer` (empty list).

Additionally, if none of the keywords `pick`, `where` or `prefer` are present the given expression is assumed to be the `where` clause, i.e. ***exp*** without any top-level keywords means `pick 1 where` ***exp*** `prefer`.

The reserved words are `user`, `task`, `pick`, `where`, and `prefer`.

6.1 Type system

The following grammar generates the valid type values τ. Any additional types we may wish to plug in should be generated by the production β.

$$\tau \quad ::= \quad int \mid string \mid bool \mid \beta$$

The type int is used for scores, $bool$ is used to evaluate exp and $cexp$, and the type $string$ is the fixed type for the variables `user` and `task`[2].

The type derivation rules are shown in Figure 3. The typing rules are parameterized to permit specialization of the language without changes to the core type system. In particular the following sets and functions must be supplied:

Type
: A set of types generated by the type grammar, i.e. at least int, $string$, and $bool$.

type : $Value \rightarrow Type$
: A total map from value literals to types.

type : $Func \rightharpoonup Type\ list \rightharpoonup Type$
: A map from function/operator symbols to argument types to return type. Notice that overloading can be obtained by mapping a function/operator symbol to a map containing several distinct type mappings.

Definition 1 *An allocation rule*

$$\texttt{pick}\ \boldsymbol{n}\ \texttt{where}\ \boldsymbol{exp}\ \texttt{prefer}\ [\boldsymbol{pexp}_1]\ \boldsymbol{cexp}_1, \ldots, [\boldsymbol{pexp}_m]\ \boldsymbol{cexp}_m$$

is well-typed if and only if is has a derivation in the type system given in Figure 3.

Lemma 2 *The types of the subexpressions in an AST are unique, that is, any AST "pick..." generated by the grammar in Figure 2 either has no type derivation or has exactly one type derivation.*

Proof. Left as an exercise for the reader. ■

6.2 Denotational semantics

This section introduces a denotational semantics that maps pieces of syntax to maps from users to their associated scores. Everything describes in this section assumes to be operating only on well-typed allocation rules. The functions do not apply to rules that do not have a type derivation.

The semantics pre-supposes the following environment of domains and functions. These are the language parameters through which all extensions should be done:

Rule, Exp, ScExp
: The set of expressions in the syntactic categories of the grammar.

[2]This can be generalized further since scores need not be int, but could be any totally ordered domain. Also, `user` and `task` could in fact be any type that would make sense in conjunction with the operators and functions present in a concrete instance of the language.

Figure 3 Type system

$$\frac{\text{type}(\mathit{value}) = \tau}{\mathit{value} : \tau} \qquad \frac{}{\texttt{user} : \textbf{string}} \qquad \frac{}{\texttt{task} : \textbf{string}}$$

$$\frac{\mathit{exp}_1 : \tau_1 \qquad \mathit{exp}_2 : \tau_2 \qquad \text{type}(\mathit{op}, [\tau_1, \tau_2]) = \tau}{\mathit{exp}_1 \;\; \mathit{op} \;\; \mathit{exp}_2 : \tau} \qquad \frac{\mathit{exp}_1 : \tau_1 \qquad \text{type}(\mathit{op}, [\tau_1]) = \tau}{\mathit{op} \;\; \mathit{exp}_1 : \tau}$$

$$\frac{\mathit{exp}_1 : \tau_1, \ldots, \mathit{exp}_n : \tau_n \qquad \tau = \text{type}(\mathit{function}, [\tau_1, \ldots, \tau_n])}{\#\mathit{function}(\mathit{exp}_1, \ldots, \mathit{exp}_n) : \tau}$$

$$\frac{n : \textbf{int} \qquad \mathit{exp} : \textbf{bool} \qquad \mathit{pexp}_1 : \textbf{int}, \ldots, \mathit{pexp}_m : \textbf{int} \qquad \mathit{cexp}_1 : \textbf{bool}, \ldots, \mathit{cexp}_m : \textbf{bool}}{\texttt{pick}\ n\ \texttt{where}\ \mathit{exp}\ \texttt{prefer}\ \texttt{[}\mathit{pexp}_1\texttt{]}\ \mathit{cexp}_1 \cdots \texttt{[}\mathit{pexp}_m\texttt{]}\ \mathit{cexp}_m}$$

eval : $Func \rightharpoonup Value\ list \rightarrow Value$
: Maps a function/operator symbol, and a list of values to a return value. This is used for all function calls in the language. The type check will reject programs with invalid function symbols, i.e. functions symbols that do not map to anything. On other hand, any defined function is required to be a total map from a list of values to a value. (That is, it must terminate and return a value in the Value domain.)

cond : $(Value \times Value \times Value) \rightarrow Value$
: Equals its second argument if the first argument is `true` and its third argument otherwise.

$+$: $(\text{Value} \times \text{Value}) \rightharpoonup \text{Value}$ A binary operator to compose score values. Need only be defined for $\text{Value} : \mathit{int} \times \text{Value} : \mathit{int}$.

The denotational semantics has a map for each syntactic category in the grammar:

$$\begin{aligned}
\mathcal{R}[\![\cdot]\!] &: \text{Rule} \rightarrow (\mathcal{P}(U) \times T) \rightarrow U \rightarrow \text{Value}_{|\mathit{int}} \\
\mathcal{E}[\![\cdot]\!] &: \text{Exp} \rightarrow (U \times T) \rightharpoonup \text{Value} \\
\mathcal{S}[\![\cdot]\!] &: \text{ScExp} \rightarrow (U \times T) \rightharpoonup \text{Value}_{|\mathit{int}}
\end{aligned}$$

The exact definitions are shown in Figure 4. Given a well-typed rule r, a user database U and a current task t, the denotation $\mathcal{R}[\![r]\!](U, t)$ yields a partial map from users to their scores.

Figure 4 Denotational semantics

$$
\begin{aligned}
&\mathcal{R}[\![\texttt{pick}\ n\ \texttt{where}\ exp\ \texttt{prefer}\ \texttt{[}pexp_1\texttt{]}\ cexp_1 \cdots \texttt{[}pexp_m\texttt{]}\ cexp_m]\!](U,t) = \\
&\qquad \{(u,s) \mid u \in U \quad \wedge \quad \mathcal{E}[\![exp]\!](u,t) = \texttt{true} \\
&\qquad\qquad\qquad\quad \wedge \quad \mathcal{S}[\![\texttt{[}pexp_1\texttt{]}\ cexp_1 \cdots \texttt{[}pexp_m\texttt{]}\ cexp_m]\!](u,t) = s\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![value]\!] &= \lambda(u,t)\ .\ value \\
\mathcal{E}[\![\texttt{user}]\!] &= \lambda(u,t)\ .\ u \\
\mathcal{E}[\![\texttt{task}]\!] &= \lambda(u,t)\ .\ t \\
\mathcal{E}[\![exp_1\ op\ exp_2]\!] &= \lambda(u,t)\ .\ \mathrm{eval}(op, \mathcal{E}[\![exp_1]\!](u,t), \mathcal{E}[\![exp_2]\!](u,t)) \\
\mathcal{E}[\![op\ exp]\!] &= \lambda(u,t)\ .\ \mathrm{eval}(op, \mathcal{E}[\![exp]\!](u,t)) \\
\mathcal{E}[\![\texttt{\#}function(exp_1,\ldots,exp_n)]\!] &= \lambda(u,t)\ .\ \mathrm{eval}(function, \mathcal{E}[\![exp_1]\!](u,t),\ldots,\mathcal{E}[\![exp_n]\!](u,t))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}[\![\texttt{[}pexp_1\texttt{]}\ cexp_1 \cdots \texttt{[}pexp_m\texttt{]}\ cexp_m]\!] &= \lambda(u,t)\ .\ \mathrm{cond}(\mathcal{E}[\![cexp_1]\!](u,t), \mathcal{E}[\![pexp_1]\!](u,t), \texttt{0}) + \cdots \\
&\qquad + \mathrm{cond}(\mathcal{E}[\![cexp_m]\!](u,t), \mathcal{E}[\![pexp_m]\!](u,t), \texttt{0})
\end{aligned}
$$

Definition 3 (Well-typed evaluation) *We say that the functions* type *and* eval *are* consistent *if and only if for all function symbols f and values $[v_1 : \tau_1, \ldots, v_n : \tau_n]$*

$$type(f, [\tau_1, \ldots, \tau_n]) = \tau \Rightarrow eval(f, [v_1, \ldots, v_n]) = (v : \tau)$$

We also say that eval *is* well-typed *with respect to* type.

Theorem 4 (Ye olde wannabe progress and preservation theorem) *The denotational function is a defined on all well-typed rules. More formally, let U, T,* Value, Func, Type, type, *and* eval *be given such that* type *and* eval *are consistent, and a rule $r \in$* Rule *is well-typed. Then $\mathcal{R}[\![\cdot]\!]$ is defined for r and furthermore $\forall u \in U, t \in T : \mathcal{R}[\![R]\!](U,t)(u) = s \Rightarrow s : int$.*

Proof. Left as an exercise for the exceedingly interested reader. ■

Corollary 5 *The denotation $\mathcal{R}[\![r]\!](U,t)$ yields a (partial) function (as opposed to just a relation), i.e. a user has a unique score or no score.*

Proof. Follows trivially from the definition of $\mathcal{R}[\![\cdot]\!]$ and the fact that $\mathcal{E}[\![\cdot]\!]$ and $\mathcal{S}[\![\cdot]\!]$ are functions. ■

6.3 Concrete prototype language

We now consider an example of how to specialize the core language to a concrete one. Recall from Section 6 the necessary parts:

1. Syntactic rewrites (syntactic sugar)
2. Function/operator symbols (for prefix, postfix or infix operators their precedence should also be specified)
3. Additional types, β, and function/operator types
4. The semantics of operators/functions

Syntactic rewrites In additional to the global rewrites described in Section 6, the following specific rewrites take place:

$$exp_1 \texttt{ <> } exp_2 \quad \rightarrow \quad \texttt{not } exp_1 \texttt{ = } exp_2 \tag{1}$$

$$userexp \quad \rightarrow \quad \#user(\texttt{task}, \texttt{user}, \text{"}userexp\text{"}) \tag{2}$$

Rewrite 1 introduces a `<>` operator (not equal to) defined in terms of the operators `=` and `not`. Rewrite 2 takes any set of tokens without whitespaces that is not recognized as a reserved word, a function/operator symbol or a constant and assumes it to be a string in a sub-language used to navigate the user database. The string is passed verbatim to the function dispatch for external processing. E.g. the fragments `manager` and `location` would be taken to be $userexp$ and rewritten as `#manager(task,user)` and `#location(task,user)`, respectively.

To sugar the language further, the operators of the next section were permitted to be used as infix and prefix appropriately with the usual precedence levels.

Function/operator symbols For the language we introduce the infix operators `or`, `and`, `=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, `/` (integer division), `%` (modulo), and the unary prefix operators `+`, `-`, `not`. These all have the usual meanings (modulo is defined as in Java and C). The following functions are introduced:

`role` Returns the set of roles for the user.

`manager` Returns the manager of the user.

`location` Returns the location of the user.

`queue` Returns the number of items on the work queue of the user.

Thus we have $\text{Func} = \{\texttt{or}, \texttt{and}, \texttt{=}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{role}, \texttt{manager}, \texttt{location}, \texttt{queue}\}$.

Types Oftentimes we wish to write a query that checks for membership of a set. E.g. if a user has many different roles, we may wish to stipulate that the role `developer` is a member of that set of roles. To this end we introduce sets of the three primitive types. The complete type grammar becomes:

$$\begin{aligned} \tau &::= int \mid string \mid bool \mid \beta \\ \beta &::= set\,of\,int \mid set\,of\,string \mid set\,of\,bool \end{aligned}$$

The required domains and functions thus become:

Value
: This contains values of the type int, $string$, and $bool$ (written as one would do it in Java or C), as well as sets hereof. The prototype does not support writing set literals.

Type
: This set will contain all the types generated by the type grammer above.

type
: The mapping from values to types is completely standard.

type
: The functions/operators type map is defined as follows:

Figure 5 Concrete grammar with binary and unary operators added

rule	::=	(`pick` *value*)? (`where` ***exp***)? (`prefer` ***score-exp****)?
exp	::=	***exp op exp*** | ***unary-op exp*** | *value* | #*function* (***exp***, ..., ***exp***) | `user` | `task`
score-exp	::=	`[` ***exp*** `]` ***exp***
op	::=	`or` | `and` | `=` | `<>` | `<` | `>` | `<=` | `>=` | `+` | `-` | `*` | `/` | `%`
unary-op	::=	`+` | `-` | `not`

$$
\begin{array}{rcl}
\texttt{or} & \mapsto & \{[bool, bool] \mapsto bool\} \\
\texttt{and} & \mapsto & \{[bool, bool] \mapsto bool\} \\
\texttt{not} & \mapsto & \{[bool] \mapsto bool\} \\
\texttt{=} & \mapsto & \{[int, int] \mapsto bool, \\
 & & [set\ of\ int, int] \mapsto bool, \\
 & & [int, set\ of\ int] \mapsto bool, \\
 & & [string, string] \mapsto bool, \\
 & & [set\ of\ string, string] \mapsto bool, \\
 & & [string, set\ of\ string] \mapsto bool, \\
 & & [bool, bool] \mapsto bool, \\
 & & [set\ of\ bool, bool] \mapsto bool, \\
 & & [bool, set\ of\ bool] \mapsto bool\}
\end{array}
\qquad
\begin{array}{rcl}
\texttt{role} & \mapsto & \{[string, string] \mapsto set\ of\ string\} \\
\texttt{manager} & \mapsto & \{[string, string] \mapsto string\} \\
\texttt{location} & \mapsto & \{[string, string] \mapsto string\} \\
\texttt{queue} & \mapsto & \{[string, string] \mapsto int\} \\
\texttt{<} & \mapsto & \{[int, int] \mapsto bool\} \\
\texttt{>} & \mapsto & \{[int, int] \mapsto bool\} \\
\texttt{<=} & \mapsto & \{[int, int] \mapsto bool\} \\
\texttt{>=} & \mapsto & \{[int, int] \mapsto bool\} \\
\texttt{+} & \mapsto & \{[int, int] \mapsto int\} \\
\texttt{-} & \mapsto & \{[int, int] \mapsto int\} \\
\texttt{*} & \mapsto & \{[int, int] \mapsto int\} \\
\texttt{/} & \mapsto & \{[int, int] \mapsto int\} \\
\texttt{\%} & \mapsto & \{[int, int] \mapsto int\}
\end{array}
$$

Notice that = is overloaded for comparison between all three primitive types as well as for set membership test (but not for set to set comparison!)

Semantics of operators/functions In the prototype the function are handled by external dispatch to methods written outside the language (in Java), although the binary and unary operators have all been added to the language core because (1) they will generally be needed in any specialization and (2) the language performs better. The semantics of all operators is completely routine as they map to same operators set of operators in Java. One exception is =, which will mean either $\in$, $=$ or $\ni$ depending on the type of its arguments.

The functions symbols `role`, `manager`, `location` dispatch to Java function that perform lookup in the user database. They will always return a string since `null` values do not exist in the language as defined here.

The function `queue` returns the current number of items in the queue of the user provided as the second argument.

6.4 Allocation as an optimization problem

It is enlightening to consider work allocation as an optimization problem. In fact, work allocation (also knows as rostering or scheduling) is one of the most common problems addressed by integer programming techniques in practice.

Assume a task t has the following rule attached to it:

$$\texttt{pick}\ n\ \texttt{where}\ exp\ \texttt{prefer}\ \texttt{[}pexp_1\texttt{]}\ cexp_1 \cdots \texttt{[}pexp_m\texttt{]}\ cexp_m$$

Given a user database U with users $\{u_1, ..., u_{|U|}\}$ and the appropriate definitions of the functions called in the expressions, an allocation is a vector $a_1, ..., a_{|U|} \in \{0,1\}^{|U|}$ where $a_j = 1$ iff the task is allocated to user u_j. We then seek to

$$\begin{aligned}
\text{maximize} \quad & \sum_{j \in [1..|U|]} \left(a_j \cdot \sum_{i \in [1..m]} [cexp_i(u_j, t)] \cdot pexp_i(u_j, t) \right) \\
\text{subject to} \quad & \forall j \in [1..|U|] : a_j = 1 \implies exp(u_j, t) \\
\text{and} \quad & \sum_{j \in [1..|U|]} a_j = n
\end{aligned}$$

When we write $[cexp_i(u_j, t)]$ in the objective function $[\cdot]$ is the *indicator function* which maps to 1 when its argument evaluates to true and to 0 otherwise. The objective function can therefore be read as "for all users, if the task is allocated to them, add to the sum of total points, the points they receive for each preference that is true for them". Strictly speaking, we should have enclosed exp, $cexp$, and $pexp$ in the evaluation function $\mathcal{E}[\![\cdot]\!]$, but it has been omitted to keep the formulation free of clutter.

The problem can be solved in polynomial time, provided that evaluating exp, $cexp$ and $pexp$ can be done in polynomial time. The naive, brute force algorithm simple evaluates each of the users: first it checks if they satisfy the hard constraint, and if they do, how many points they get. It then picks the top n satisfying users. The correctness of this algorithm is due to the monotonicity: the computation of each user's score is independent of who else is considered for allocation. (Proof left as an exercise for the reader.)

7 Implementation

The work allocation language has been implemented and tested in the PEAS BPM platform with the specialization described previously. The language used was Java with ANTLR as the lexer and parser generator. The design of the system is shown in Figure 6.

Notice the class `Type` which corresponds to the type production β in the semantics. Also notice the method `allUsers` on the class `UserData`. It is used to evaluate domain-dependent expressions such as `user.role != "Manager"`, which would be computed by taking allusers and then removing those with role `Manager`. Performance concern may later mandata that such expression are disallowed so that every allocation rule must contain at least one domain-independent predicate in each part of a disjunction.

Notice the very important classes `userScoreMap` and `userValueMap`. These are used to return the map of user ids to scores that ultimately decided what user is picked for the task.

8 Evaluation

The prototype has been tested and preliminarily evaluated with Infosys' PEAS platform. Although the language is slated for inclusion in the PEAS platform, it is yet to be used in a production setting at the time of writing.

Following the trend of workflow pattern evaluations Table 1 shows a patterns-based evaluation of our work allocation language. It is essential to note that only the patterns that specify who is *ultimately* allowed to perform a task are included (cf. our discussion in Section 3.2); as mentioned previsouly, the language did not set out to specify runtime negotiation rules.

Whereas our language fares quite well in the comparison, the patterns-based analysis inadequately captures the expressive power of our language. The last three rows (*Multiple resources*, *Soft constraints*,

Figure 6 Class diagram including lexer/parser generator files

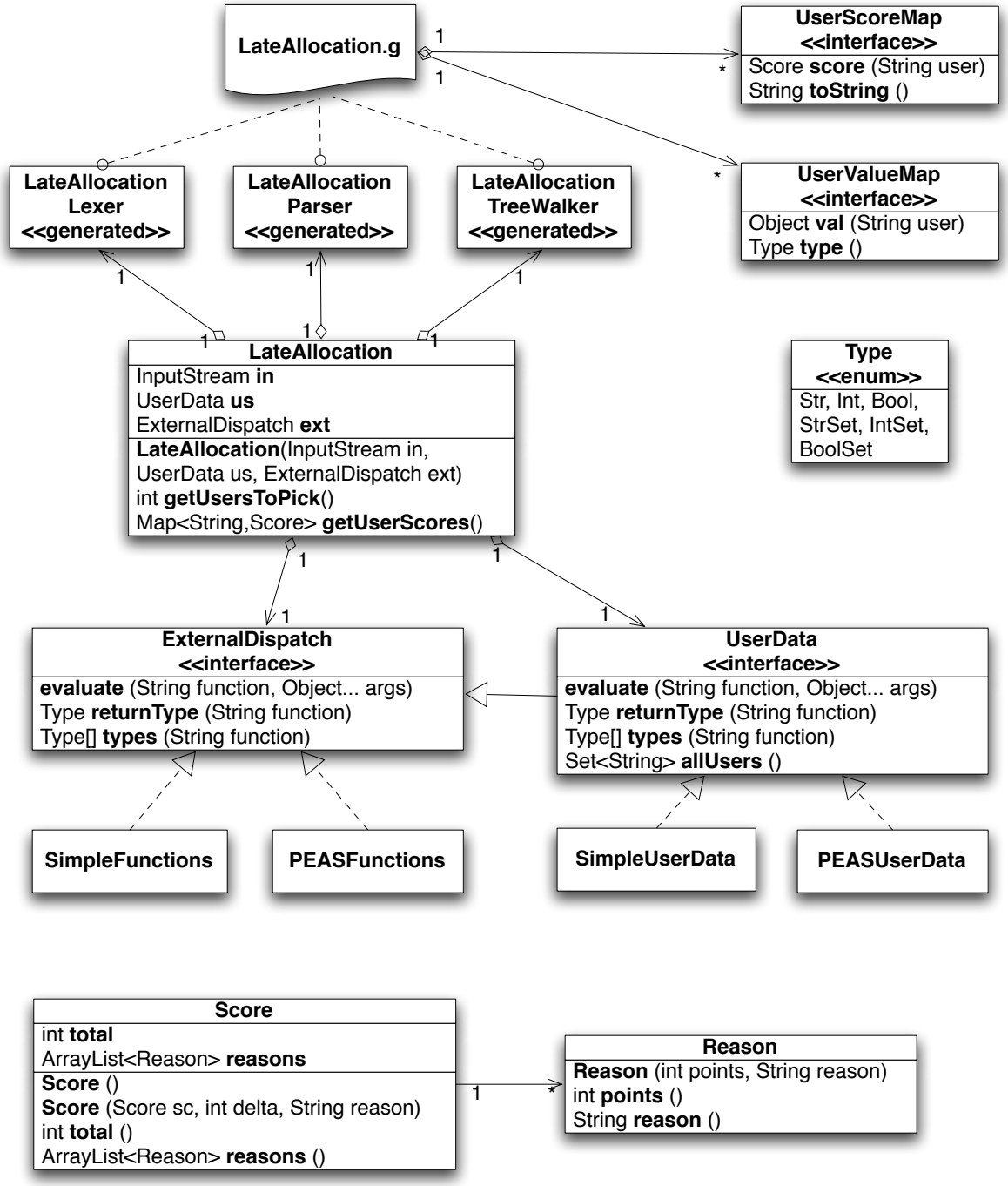


Table 1 A patterns-based product comparison. This table is an adaptation of a similar table from Russell et al. [1]. The columns and rows marked in *italics* have been added by us; everything else is from Russell et al. [1]. In the entries a + indicates full support for the pattern, x/- indicates partial support (e.g. through coding a bit) and - indicates no support.

Pattern	***Ours***	**Staffware**	**Websphere**	**FLOWer**	**COSA**	**iPlanet**
Direct allocation	+	+	+	+	+	+
Role-based	+	+	+	+	+	+
Separation of duties	+/-	-	+	+	+/-	+
Case handling	+/-	-	-	+	-	-
Retain familar	+/-	-	+	+	+	+
Capability-based	+	-	-	+	+	+
History-based	+/-	-	-	-	+/-	+
Organizational	+	+/-	-	+	+	+
Round robin	+/-	-	-	-	+/-	+/-
Random	+	-	-	-	+	+/-
Shortest queue	+/-	-	-	-	+	+/-
Multiple resources	+	?	?	?	?	?
Soft constraints	+	?	?	?	?	?
Combine patterns	+	?	?	?	?	?

and *Combine patterns*) illustrate this. E.g. soft constraints are not a single pattern, but an idea that could easily be expanded to comprise an entire suite of patterns in its own right. The patterns-based analysis falls short here because the inventors of the analysis did not anticipate soft constraints. Similarly, the patterns-based analysis does not mention if patterns can be combined and if so with what constraints. If two systems both allow, say, role-based allocation and organizational allocation, but only one of the systems permits the designer to combine these two patterns in one allocation rule, a patterns-based analysis will not reveal this critical difference of expressiveness. This rather clearly demonstrates that the patterns should be used as a checklist to avoid forgetting important functionality, but they should never be applied blindly or be trusted as a measure of expressiveness.

Developing a more precise measure of expressiveness is the subject of future work. Also, constructing a collection of soft constraint patterns is future work.

9 Related work

Please refer to the annotated references.

10 Conclusion and future work

The language can express all patterns for which it was designed and all examples that were deemed necessary. In addition, through the external function calls, te function can interface with any legacy systems using a simply type system. The allocation patterns by van der Aalst et al. [1] while useful, proved insufficient to really capture the expressiveness of the language which is arguably much higher than its competitors when it comes to our focus, creation patterns. Also the patterns lack soft constraints and preferences and we feel that this is a particularly important and useful feature of the current design.

The prototope has been tested and and has potential for adoption and integration into production systems for Infosys customers. Future integration with Microsoft BizTalk Server is being considered.

Certain issues were left out and constitute interesting areas of future research:

- Transactions, rollback and compensation.

- Security, including the ability to grant/remove access privileges.
- Distribution, i.e. the allocation algorithm is distributed over several peers that each govern allocation to their own resources.
- Gaming, i.e. users playing the system by logging in/out, placing artificial constraints to avoid certain allocations, etc.

A GUI for the language is begin developed in the production setting where the language is to be used. For this a standard form (e.g. disjunctive normal form) may be useful for making the language more amenable to adoptation in a GUI. Also previews of how a rule would work on the live data would be very useful.

The benefits of the most important contribution are yet to be reaped: the language is a small DSL and this means that allocation rules are not only expressions that can be evaluated, but also pieces of data that can be used to perform static analysis, feasibility check and – our most important ongoing research area – performance simulation. Performance simulation can be used to identify bottlenecks, estimate capacity required, and suggest what resources to add. This ability to do what-if performance analysis on process before and during deployment is very promising.

A Examples

A.1 Examples of hard constraints

```
"Direct logical allocation based on properties using logical
 connectives to combine"

    role = "Reviewer" and location != "Africa" or role = "Manager"


"Direct allocation with negation"

    user = "Bob" or not location = "Europe"


"User database navigation.  Manager's manager should be located
in India"

    manager.manager.location = "India"


"Advanced user database navigation.  Allocate to a manager who has a
subordinate programmer any number of levels down."

    subordinate*.role = "Programmer" and role = "Manager"


"Variable binding.  Same as previous, but the programmer must be in
Africa."

    user = subordinate*  s.t.
    user.role = "Programmer" and user.location = "Africa"
    and role = "Manager"
```

```
"Integers.  Allocate to a resource with at least level 5
Java skills"

    skill.Java >= 5


"Reading process parameters."

    role = var("InputParam")


"Using context/functions without user parameter.  Allocate to person
who least recently did the job (round robin)."

    this.user = whoLeastRecentDid(this)


"Using context information parametrized over the user. Allocate to
persons with less than 10 queue items."

    runtime(user,queue) < 10


"Alternatively, produce a set of users with short queues and test
for membership."

    this.user memberOf usersWithShortQueues()


"Give current task as argument /and/ user as argument to external
function.  Invoke external allocation handler for this task."

    externalFunction(this,user) = 5


"Externalize predicate completely."

    externalAllocEngine(this,user) [= true]


"Refer to other tasks.  Assign to the same person who did "Approve
Invoice"."

    this.user = "Approve Invoice".user


"Alternatively, look it up using named function on log/history."

    this.user = history("whoDid", "Approve Invoice")
```

"Refer to attribute on the process, e.g. the creator."

```
this.user = process("creator")
```

"Rules can apply to scopes instead of just single tasks. All tasks in scope should be done by a programmer."

```
(A.B.C.D.E.F) & [role = "Programmer"]
```

"Nested scopes conjoin. All tasks should be done by a programmer task C in addition stipulates J2EE skills."

```
(A.B.(C & [skill = "J2EE"]).D.E.F) & [role = "Programmer"]
```

"Making exceptions. All tasks in scope to be done by a programmer, except C which needs a manager who is not necessarily a programmer."

```
(A.B.C.D.E.F) & [(role = "Programmer" or
                  (this = C and role = "Manager")]
```

"Set equality. Allocate to someone with exactly the same (no more, no less) skills as Bob"

```
every skill = some Bob.skill and
every Bob.skill = some role
```

"Subset test. Allocate to someone with at least the same roles as "Bill"."

```
every "Bill".role = some role
```

"Non-empty intersection test. Allocate to someone who shares at least one role with "Bill"."

```
some role = some "Bill".role
```

"Allocate all tasks in the scope to the same programmer. Could also use history lookup instead of direct reference."

```
(A.B.C.D.E.F)
& [role = "Programmer and
   A.user = B.user = C.user = D.user = E.user = F.user]
```

"Allocate all tasks in the scope to the same person except

```
task C which should be a manager who is not that person."

    (A.B.C.D.E.F)
    & [A.user = B.user = D.user = E.user = F.user and
       C.role = "Manager" and not A.user = B.user]


"Externalizing preferences.  See next set of examples for a better
approach.  Allocate to most competent Java programmer"

    user = bestJavaPrg()


"Needing more resources to have a go at one task"

    pick 2 where ...
```

A.2 Examples of soft constraints/preferences

```
"Prefer manager to submanager to anyone, but anyone can do it."

    [10] role = "Manager" or [5] role = "Submanager"


"Use submanager if manager's queue is too long"

    [10] (role = "Manager" and runtime(user,queue) < 10) or
    [5]  (role = "Submanager")


"Prefer most competent Java programmer"

    [10] user = bestJavaPrg()


"Prefer programmers with higher Java skills"

    [5*user.Java] role = "Programmer"


"Prefer shorter queue"

    [runtime(user,queue)]


"Prefer in order of least recently having done the task"

    [-leastRecentlyDidRanking(this,user)]


"Minimize expected task completion time"
```

```
    [1/(runtime(user,queue) + avgTime(this,user))]


"Minimize total makespan (expected completion time)"

    Not expressible in current language proposal, I believe.
    The following will not work because (1) it does not account for
    some of the users being the same and therefore spending more time
    and (2) more importantly the rule is not evaluated once for the scope,
    but once for /every/ task in the scope (I our current thinking).

    (A | B | C) & [1/(runtime(a.user,queue) + avg(a,a.user)
                    + runtime(b.user,queue) + avg(b,b.user)
                    + runtime(c.user,queue) + avg(c,c.user))]

    A simpler sequential example like A.B.C. will work, though,
    because the incremental allocation in three steps will
    find the minimum since no parallellization is possible.

    Basically we currently allocate one task at a time, meaning that
    any optimization whose objective function depends on several
    non-sequential tasks cannot be done.  If we were to have more
    than a one-task horizon, this would change.
```

A.3 Patterns-based examples

```
Pattern:                Syntax:                    Need:
-------------------------------------------------------------------------------
Direct                  user = "xxx"               list of users

Role-based              role = "xxx"               list of roles pr. user

Separation of duties    user != task.user          list of other tasks/scopes

Case handling           user = "xxx"               put rules on scopes

Retain familiar         user = task.user           absolute/backward only refs
                        user = next/previous.user  relative - not allowed

Capability-based        skill = "xxx"              list of skills pr. user

History-based           user|role|skill = $f(Tr)$  full language for $f$

Organizational          user = ("xxx"|task.user).(manager|subordinate)
                        title = "xxx"              list of org. relations

Round robin             user = whoLeastRecentlyDid(task)
                                                   cf. history-based

Random allocation       <nothing>                  break ties at random by
                                                   default
```

```
Shortest queue          user = shortestQueue()     engine's runtime info
```

References

[1] Russell, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Workflow resource patterns. Technical report, Eindhoven University of Technology (2005)

[2] OASIS Open, Inc.: WS-BPEL 2.0 Committee Specification. (May 2006)

[3] Kloppmann, M., Koenig, D., Leymann, F.: BPEL 4 People. Technical report, IBM and SAP (July 2005)

[4] Gregory, S., Paschali, M.: A prolog-based language for workflow programming. In Murphy, A.L., Vitek, J., eds.: Coordination Models and Languages, 9th International Conference, COORDINATION 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings. Volume 4467 of Lecture Notes in Computer Science., Springer (2007) 56–75

[5] Bonner, A.J.: Workflow, transactions and datalog. In: PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, New York, NY, USA, ACM Press (1999) 294–305

[6] Malik, N.: Why a workflow model is not code. web page, last accessed Aug. 10, 2007 (March 2006)

[7] Stefansen, C., Borch, S.E.: Balancing flexibility and support in business process management systems (bpms). International Journal of Business Process Integration and Management (2007) To appear.

[8] R.M.Tomasulo: An efficient algorithm for exploiting multiple arithmetic units. IBM Journal of Research and Development **11**(1) (January 1967) 25–33

[9] Hamadi, Y., Quimper, C.G.: The smart workflow foundation. Technical Report MSR-TR-2006-114, Microsoft Research (November 2006)

[10] Senkul, P., Toroslu, I.H.: An architecture for workflow scheduling under resource allocation constraints. Information Systems **30**(5) (July 2005) 399–422