SOFTALLOC: A Work Allocation Language with Soft Constraints

Christian Stefansen
University of Copenhagen

Sriram Rajamani
Microsoft Research, India

Parameswaran Seshan
SETLabs, Infosys Technologies Ltd.

Abstract

Today's business process orchestration languages such as WS-BPEL *and* BPML *have high-level constructs for specifying flow of control and data, but facilities for allocating tasks to humans are largely missing. This paper presents* SOFTALLOC*, a work allocation language with* soft constraints*, and explains the requirements and trade-offs that led to its design, in particular, what soft constraints are, and how they enable business process definitions to capture allocation rules, best practices, and organizational goals without rendering the business processes too strict.*

SOFTALLOC *combines with virtually any business process language and any conceivable legacy system, while guaranteeing polynomial performance. We present the design, the formal definition, and an evaluation of* SOFTALLOC.

1 Introduction

Computer-orchestrated business processes are increasingly playing a direct role in how companies organize work and now commonly involve both human resources and computer resources. A widely accepted architecture is to have a business process orchestration engine that orchestrates 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 human tasks. Therefore the process orchestration engine must decide in negotiation with the human resources who of the eligible resources ultimately carries out the task. Assigning a task to a resource is referred to as *allocation*.

Allocating tasks to humans is inherently more complex than allocating to computer resources: in addition to having multiple, changing attributes that decide what they can, may or should do, humans have personal preferences and they may choose to override the allocation rules at runtime, e.g. because they possess domain knowledge that is not captured in the system or because they make conscious, reflected violations to speed up processing in cases where the process description focuses too narrowly on compliance.

Suppose in a standard sales process we wish to say that the task *Receive payment* should be carried out by someone with the role *Finance*. We can imagine specifying this by attaching a rule to the task saying:

```
role = "Finance"
```

We can then compose small building blocks of rules into larger rules. Now imagine that we want *Receive payment* to be carried out by the user who did *Invoice* to retain familiarity. We would then add

```
    role = "Finance"
and user = whoDid ("Invoice")
```

as a rule to *Receive payment*.

But something is awry here. While we would certainly *prefer* the task *Receive payment* to be done by the same person who did the invoicing, this is only a *preference*—certainly not a strict rule that should be allowed to stand in the way of timely workflow completion if the designated person happens to be busy or temporarily absent. We have just committed one of the most common mistakes in workflow specification: we have promoted a *soft goal* to a strict rule and thereby created an inflexible system!

Alternatively, we might have removed the rule and only have said `role = "Finance"`, but that would have left out useful intentional information about our best practice. So just specifying the minimal number of constraints is not attractive either.

This example illustrates that allocation constraints can represent a wide spectrum of specifications: some rules are best kept strict (e.g. *Expense approval* must be done by a *Manager*) while other rules are simply guidelines (e.g. *Replenish printer cartridges* should be allocated on a rotation basis (*round robin*)). The latter allocation strategy represents an organizational *soft goal*, which might have been "rotate tedious tasks between qualified workers to achieve a sense of fairness and variation and keep workers happy". This is undeniably a laudable goal, but if the company is experiencing peak load, this goal must temporarily yield to more mission-critical business goals (e.g. response time *vis-à-vis* our customers). Therefore, it cannot be written as a hard constraint, but leaving it out entirely renders the system unable to suggest the preferred person.

Going back to our example what we probably mean could be written as

```
  role = "Finance"
prefer [10] user = whoDid ("Invoice")
```

which states that we require a finance person to handle the activity under all circumstances, but we prefer the person who did the invoicing in that process. The number 10 represents a *score* to indicate how strong a preference this is. This becomes more interesting, when more preferences are in play. Consider the following rule for allocating the *Credit approval* step in the workflow:

```
  role = "Manager" or role = "Finance"
prefer [10] role = "Manager"
       [-queueSize()]
```

The rule states that either *Manager* or *Finance* should handle the *Credit approval* task. A manager is preferred, but the number of items in the manager's queue is deducted from the preference level; i.e. someone with a short queue is preferred. Indeed, if all managers have more than 10 items in their work queues, someone from *Finance* will be preferred in the interest of time. This shows how soft constraints in conjunction with hard constraints can be used to express soft goals and performance heuristics in allocation.

Such soft constraints can be expressed and combined on many levels. We can imagine that the company has a general policy to prefer the shortest queue and seek to distribute activities by rotation (*round robin*):

```
prefer [-queueSize()]
       [rndRobin(1,10)]
```

This combines with a process-level policy to strongly prefer a user at the same location as the process:

```
prefer [15] user.location = proc.location
```

If we consider *Receive payment* with the same rule as before, there are now three combined rules in effect:

```
  role = "Finance"
prefer [10] user = whoDid ("Invoice")
```

```
prefer [-queueSize()]
       [rndRobin(1,10)]
```

```
prefer [15] user.location = proc.location
```

We can now try to run the allocation on *Receive payment* with the combination of the three levels of rules active. Suppose we have already compiled the necessary information about each user (where `rndRobin` is a number [1..10] indicating how long time it has been since that user did *Receive payment*):

User	`role`	`location`	`queue`	`rndRobin`
Ashok	Finance	India	3	9
Diego	Support	Germany	7	5
John	Finance	US	29	2
Julia	Finance	Germany	12	3
Uno	Finance	Germany	9	8

Assuming that `proc.location = "Germany"` and `whoDid("Invoice") = "Julia"` the work allocation language will produce the following suggestion sorted by score:

User	**Score**	**Reasons**
Julia	16	[15] proc.location [10] whoDid ("Invoice") [3] rndRobin [-12] queueSize
Uno	14	[15] proc.location [8] rndRobin [-9] queueSize()
Ashok	6	[9] rndRobin [-3] queueSize()
John	-27	[2] rndRobin [-29] queueSize()

Notice, that Diego does not appear in the table as he does not satisfy the hard constraint on `role`. Julia will be able to see the contents of the score table when she receives the activity in her queue. This enables her to make an informed choice if for some reason she decides to re-allocate the task. In other words, soft constraints allow users to see the intensional information that has a bearing on the allocation, while they can retain their possibility to re-allocate based on any extra runtime knowledge they may have. By showing its reasoning the language supports the users in making an informed decision.

1.1 Contributions

This paper defines a declarative work allocation language, SOFTALLOC, that (a) supports soft constraints, (b) plugs into any business process language in an aspect-like manner, (c) can be used with a wide range of platforms/legacy systems, (d) allows rules on many organizational levels to be combined, and (e) runs fast enough to allocate and re-allocate at runtime. Let us examine the reasons and implications:

Supports hard constraints *and* soft constraints The introduction and use of soft constraints in a work allocation language alleviates serious issues with workflows that are either too rigid or too lenient (see Stefansen and Borch [14] for a detailed discussion).

Plugs into any business process language A work allocation language definition that is orthogonal to the business process language, meaning that it can combine with any of the popular languages currently being used.

Plugs into any platform/legacy system Carries the same advantages.

Permits composition of rules in several scopes Rules can be attached to activities, to scopes, to entire workflows, to people and to the entire system as policies. The compositional design of the language ensures that it is meaningful and well-defined to compose rules on several different levels.

Runs in P-time (provided that the user-defined functions do so.) The allocation can always be solved in polynomial time, making it feasible to rerun the allocation as often as needed during execution, but it is yet able to express a wide range of heuristics to improve workflow performance and human resource utilization.

The language has been implemented, tested, and evaluated in Infosys' PEAS platform, and it is slated for inclusion in the PEAS platform with a GUI that is being developed. We would like to stress early that the paper does not contribute an interface to be used directly by business process analysts, designers or users. For that purpose the syntax is too low-level, and it therefore remains future work to design a user interface for the language.

2 Background and requirements

This project was done in collaboration with Infosys Technologies Ltd., India. Infosys uses *WS-BPEL* [8] and *BPML*, and typical applications include sales support, banking and *business process outsourcing* (BPO) projects.

While dedicated systems and professional workflow systems have support for allocation [11], popular process languages, including *WS-BPEL* [8] and *BPML*, do not. 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 allocation facilities. Some products and initiatives address this (e.g. *BPEL 4 People* [6]), but they remain in an embryonic stage.

In short, there is a need to augment existing business process languages with allocation support. There is also a need that those allocation rules support soft constraints so as to be able to guide the users whilst avoiding over-specification that leads to rigid systems.

2.1 Expressiveness requirements

To map out the allocation constraints that the language must be able to express we (1) compiled a large selection of actual business processes, and (2) cross-checked the requirements with the existing research on resource allocation patterns [11, 5, 13]. The most important scenarios were:

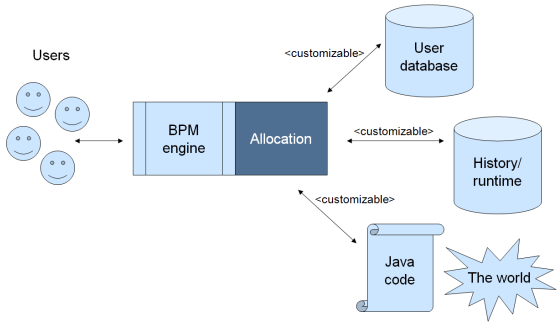

Figure 1. The architectural context of work allocation

Allocate to creator The entire process is allocated to the resource who instantiated it.

Multiple roles An activity is allocated to several roles. A resource having at least one of the roles mentioned is required.

Soft constraints The ability to specify that some rules are violable, while others are not. The former are referred to as soft constraints, and they carry an integer score. This score is used to give a preference between the resources that satisfy the hard constraints.

Multiple resources An activity is allocated to several resources, all of whom need to collaborate on the activity.

Scalar properties For some attributes there are several levels, e.g., 1 (Novice), 2 (Medium), 3 (Expert). In such cases it must be possible to specify (a) a minimum skill level for some needed skill (e.g. Java level 2) or (b) a soft preference for assigning the task to the resource with the highest skill level.

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.

Minimize makespan This is just one of many goals one can try to optimize. The soft constraints we have focused on here intentionally *cannot* perform full optimization because performance has been prioritized—instead we can express local greedy heuristics that in practice perform quite well.

One may justifiedly wonder how runtime negotiation is handled, i.e. the situation where activities are not given directly to a user, but allocated through some protocol of offer/accept to a group of users. Or activities are escalated

and re-allocated. Or activities are allocated to a queue associated with many users rather than to one particular user. These issues are certainly as important as the work allocation rules we have discussed, but since they are established protocols that rarely, if ever, change, they are better handled as an integral part of the BPM engine itself. The architecture is such that the BPM engine handles all runtime requests and then queries the allocation engine for a suggestion when allocation/re-allocation is needed due to some change in the state of the system (see Figure 1).

Also notice that the process engine, not the language, decides how queues are handled, e.g. if resources can autonomously override task priorities, skip jobs in the queue, etc. If we wish to assign activities to a queue shared by many users, this is handled by inserting that queue's name in the list of potential users that is given to the allocation engine. The allocation engine will then obliviously consider the queue as if it were a user.

Patterns In addition to the allocation scenarios above, a large number of "resource patterns" have been proposed [11]. Since we handle runtime negotiation in the process engine, only a few of these patterns apply: user data patterns (*Direct allocation*, *Role-based allocation*, *Capability-based allocation*, *Organizational allocation*), history-based patterns (*Separation of duties*, *Retain familiar*, *History-based allocation*), scope patterns (*Case handling*), and runtime information patterns (*Deferred allocation*).

Time of allocation 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 any order, 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 and helps the system 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.

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 process execution engine might as well postpone allocation until someone's queue is (almost) empty, and only then allocate the activity. Late allocation can be problematic for human workers in terms of planning because they only see a partial list of the tasks that potentially need their attention.

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.

2.2 Contextual requirements

Some important requirements can be derived immediately from the business context and the architectural context:

- The language must work in tandem with any process model (e.g. *WS-BPEL*, *BPMN*, *EPCs*, Petri net-based models) that the BPMS uses.

- Because the environment (user database, log formats, legacy interfaces) is different in every organization, the functions that read these must be externalizable.

- Oftentimes organizations have soft goals on many different levels. Therefore the language should allow several levels of rules to combine easily and seamlessly, so that e.g. general policies can be changed without mandating a change to all subordinate processes.

- It is essential that the allocation engine can recompute a new suggestion immediately when changes occur. This precludes *NP*-hard computation so the language must be able to express only polynomial-time performance heuristics. For this reason we do not allow freely interdependent allocation rules; an allocation rule for a task can only depend on the allocation of other tasks that are guaranteed to have completed (no cyclic allocation dependencies). For the same reason time scheduling (i.e. specifying that an activity must be done at a particular time) is not possible, though scheduling constraints can be introduced in a limited way as external functions.

3 Language definition

This section describes the syntax, type system, and semantics of SOFTALLOC. We assume that the workflow is specified using *WS-BPEL*, *BPML* or some comparable business process notation. The only assumption we make is that a workflow contains a finite set of tasks, each of which is uniquely named.

The syntax of the allocation language SOFTALLOC is given in Figure 2. A ***rule*** has a `pick` prefix that specifies the number of users needed to perform the task, and

rule	::=	(`pick` *const*)? ***clauses***
clauses	::=	(`where` ***exp***)? (`prefer` ***pair****)?
exp	::=	***exp op exp*** | ***unary-op exp*** | *value* | *function* (***exp***, ..., ***exp***) | `user` | `task` | `process`
pair	::=	`[` ***exp*** `]` ***exp***
op	::=	`+` | `-` | `*` | `/` | `<` | `>` | `=` | `<>` | `and` | `or`
unary-op	::=	`+` | `-` | `not`
function	::=	`procStr` | `userStr` | `queueSize` | `rndRobin` | `whoDid` | ...

`process.`*id*	⟶	`procStr(process,"`*id*`")`
`user.`*id*	⟶	`userStr(user,"`*id*`")`
`role`	⟶	`userStr(user,"role")`
exp <> ***exp***	⟶	`not` ***exp*** = ***exp***
`whoDid(`*id*`)`	⟶	`whoDid(process,`*id*`)`
`queueSize()`	⟶	`queueSize(user)`
`rndRobin(`*id*`,`*id'*`)`	⟶	`rndRobin(user,`*id*`,`*id'*`)`

Figure 2. Work allocation language core grammar (top), implementation-specific operators and functions (middle), and implementation-specific rewrites (bottom)

two ***clauses***. The `where` clause specifies a *hard constraint*, which is a set of users from which allocations can be made to this task, and the `prefer` clause specifies a list, where each element is a pair of a *score* and a *soft constraint*. Implicitly, they define a scoring function to order the set of users given by the `where` clause. The only allowed variables are in the language are `user` and `task` and `process`. The language is parameterized over the set of operators, functions, and types; in Section 3.1 we explain this in more detail. This means that a large number of extensions to the language can be made without re-working the core semantics. Thus, the properties described here are valid for any valid set of operators, functions, and types that one might wish to use, as we describe later.

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*** with no `prefer` clause.

For completeness Figure 2 also shows an example of implementation-specific operators and functions and syntactic rewrites (for syntactic sugar). This example extension corresponds to the functions we have used in the examples throughout the paper. Interested readers are encouraged to consult the technical report [15] for more details on how the example extension interacts with the core language.

$$\frac{\Theta(value) = \tau}{value : \tau} \qquad \frac{}{\texttt{user} : \mathbf{string}}$$

$$\frac{}{\texttt{task} : \mathbf{string}} \qquad \frac{}{\texttt{process} : \mathbf{string}}$$

$$\frac{exp_1 : \tau_1 \qquad exp_2 : \tau_2 \qquad \Omega(op, [\tau_1, \tau_2]) = \tau}{exp_1 \;\; op \;\; exp_2 : \tau}$$

$$\frac{exp_1 : \tau_1 \qquad \Omega(op, [\tau_1]) = \tau}{op \;\; exp_1 : \tau}$$

$$\frac{exp_1 : \tau_1, \ldots, exp_n : \tau_n \qquad \tau = \Omega(function, [\tau_1, \ldots, \tau_n])}{function(exp_1, \ldots, exp_n) : \tau}$$

$$\frac{\begin{array}{c} exp : \mathbf{bool} \qquad pexp_1 : \mathbf{int}, \ldots, pexp_m : \mathbf{int} \\ n : \mathbf{int} \qquad cexp_1 : \mathbf{bool}, \ldots, cexp_m : \mathbf{bool} \end{array}}{\begin{array}{l} \texttt{pick}\ n\ \texttt{where}\ exp \\ \quad \texttt{prefer}\ [pexp_1]\ cexp_1 \cdots [pexp_m]\ cexp_m \end{array}}$$

Figure 3. Type system

An allocation rule is well-typed if and only if after syntactic rewrites it has a derivation in the type system given in Figure 3. The language has only static types, types are inferred (no explicit type declarations).

Types are given by $\tau \;\; ::= \mathbf{int} \mid \mathbf{string} \mid \mathbf{bool} \mid \beta$, where β represents any additional types that are given given as parameters as part of an implementation-specific extension of the language.

In addition to the types β, any parameters to instantiate the language come with (1) a domain $\mathcal{V}$ of values, which contains the set $\mathcal{U}$ of users, the set $\mathcal{T}$ of tasks, and the set $\mathcal{P}$ of process instance identifiers, (2) a domain $\mathcal{F}$ of function and operator symbols, (3) a function $\Theta : \mathcal{V} \to \tau$ that maps values to types, and (4) a function $\Omega : \mathcal{F} \rightharpoonup \tau^* \rightharpoonup \tau$ that maps functions and input types to output types. Intuitively, the type system requires hard constraints and soft constraints to be of type $\mathbf{bool}$, and scores to be of type $\mathbf{int}$.

3.1 Denotational semantics

When the process execution engine calls upon the work allocation engine to allocate a user to a task, it sends the allocation rules that apply to that task along with a set of users, the name of the task to be allocated, and the current process instance id. In response the work allocation engine sends back a subset of those users as well as their scores

$$\Gamma[\![\texttt{where}\ exp\ \texttt{prefer}\ \texttt{[}pexp_1\texttt{]}\ cexp_1 \cdots \texttt{[}pexp_m\texttt{]}\ cexp_m]\!](U,t,p) =$$
$$\{(u,s) \mid u \in U \quad \wedge \quad \Delta[\![exp]\!](u,t,p) = \texttt{true}$$
$$\wedge \quad \Pi[\![\texttt{[}pexp_1\texttt{]}\ cexp_1 \cdots \texttt{[}pexp_m\texttt{]}\ cexp_m]\!](u,t,p) = s\}$$

$$\Pi[\![\texttt{[}pexp_1\texttt{]}\ cexp_1 \cdots \texttt{[}pexp_m\texttt{]}\ cexp_m]\!] =$$
$$\lambda(u,t,p)\,.\,\mathbf{cond}(\Delta[\![cexp_1]\!](u,t,p), \Delta[\![pexp_1]\!](u,t,p), \texttt{0}) + \cdots$$
$$+\,\mathbf{cond}(\Delta[\![cexp_m]\!](u,t,p), \Delta[\![pexp_m]\!](u,t,p), \texttt{0})$$

$$\Delta[\![value]\!] = \lambda(u,t,p)\,.\,value \qquad \Delta[\![exp_1\ op\ exp_2]\!] = \lambda(u,t,p)\,.\,\mathbf{eval}(op, \Delta[\![exp_1]\!](u,t,p), \Delta[\![exp_2]\!](u,t,p))$$
$$\Delta[\![\texttt{user}]\!] = \lambda(u,t,p)\,.\,u \qquad \Delta[\![op\ exp]\!] = \lambda(u,t,p)\,.\,\mathbf{eval}(op, \Delta[\![exp]\!](u,t,p))$$
$$\Delta[\![\texttt{task}]\!] = \lambda(u,t,p)\,.\,t \qquad \Delta[\![func(exp_1,\ldots,exp_n)]\!] = \lambda(u,t,p)\,.\,\mathbf{eval}(func, \Delta[\![exp_1]\!](u,t,p), \ldots, \Delta[\![exp_n]\!](u,t,p))$$
$$\Delta[\![\texttt{process}]\!] = \lambda(u,t,p)\,.\,p$$

Figure 4. Denotational semantics

(and leaves it up to the BPM engine to handle runtime negotiations).

This is reflected by the denotational semantics, which maps a clause and a triple of a set of users, a task, and a process instance id, to a function from users to integer values (denoted by $\mathcal{V}_{|int}$). The semantic functions have the following signatures:

$$\Gamma[\![\cdot]\!] \;:\; \textbf{\textit{clauses}} \to (2^{\mathcal{U}} \times \mathcal{T} \times \mathcal{P}) \to \mathcal{U} \rightharpoonup \mathcal{V}_{|int}$$
$$\Delta[\![\cdot]\!] \;:\; \textbf{\textit{exp}} \to (\mathcal{U} \times \mathcal{T} \times \mathcal{P}) \rightharpoonup \mathcal{V}$$
$$\Pi[\![\cdot]\!] \;:\; \textbf{\textit{pair*}} \to (\mathcal{U} \times \mathcal{T} \times \mathcal{P}) \rightharpoonup \mathcal{V}_{|int}$$

The definitions of these denotations are shown in Figure 4. The definitions use two auxiliary functions, **eval** and **cond**. The function **eval** is used to evaluate external functions that are defined in the parameters given to instantiate the language. Formally, $\mathbf{eval} : \mathcal{F} \rightharpoonup \mathcal{V}^* \to \mathcal{V}$ maps a function or operator symbol, and a list of values to a return value. In an instantiation of the language, every defined function is required to be a total map from a list of values of appropriate argument types to a value of the appropriate return type. The function $\mathbf{cond} : (\mathcal{V} \times \mathcal{V} \times \mathcal{V}) \to \mathcal{V}$ returns its second argument if the first argument is `true`, and its third argument otherwise.

Given a well-typed rule r `= pick` n c, with a user database U, a current task t, and a process instance id p, the denotation $\Gamma[\![c]\!](U,t,p)$ yields a partial map from users to their scores. Using this partial map, the allocation engine chooses n users for the task t in process instance p heuristically based on their scores (and other criteria, such as availability of the users). For more definitions and the progress and preservation theorem see the technical report [15].

Definition 1 (Composition of rules) *Given two rules* `where` *exp* `prefer` *pairs and* `where` *exp′* `prefer` *pairs′ their composition is defined as follows:*

$$\Gamma[\![\texttt{where}\ exp\ \texttt{prefer}\ pairs]\!](U,t,p)$$
$$\oplus\ \Gamma[\![\texttt{where}\ exp'\ \texttt{prefer}\ pairs']\!](U,t,p) =$$
$$\{(u,s) \mid u \in U \quad \wedge \quad \Delta[\![exp]\!](u,t,p) = \texttt{true}$$
$$\wedge \quad \Delta[\![exp']\!](u,t,p) = \texttt{true}$$
$$\wedge \quad \Pi[\![pairs\ pairs']\!](u,t,p) = s\}$$

For almost any concrete instantiation of the language with a sane `and` operator, this definition will be equivalent to

$$\Gamma[\![\texttt{where}\ exp\ \texttt{and}\ exp'\ \texttt{prefer}\ pairs\ pairs']\!](U,t,p).$$

4 Evaluation

The prototype has been tested and evaluated, and it is slated for inclusion in Infosys' BPM platform, PEAS, with a GUI that is being developed. To prove tangible business benefits based on more than just anecdotal evidence, an empirical study is necessary; this remains future work.

To evaluate the domain fit we applied two methods: (1) a patterns-based analysis and (2) qualitative discussions sessions with industry experts, who confirmed that the language fits well with the needs for human-intentisive workflows (e.g., insurance claims, call centers).

A patterns-based analysis in the style of Russell *et al.* [11] yields a first approximation of the domain fit of an allocation language. The analysis proceeds by evaluating if the language supports, partially supports or does not support each of the patterns in a suite of 43 patterns collected from industry and research. Table 1 shows a patterns-based evaluation of our work allocation language juxtaposed with the outcome of related patterns-based analyses. Note that

	Research			Open src.			Commercial				
Pattern	***SoftAlloc***	**B4P/WS-HT**	**newYAWL**	**JBoss jBPM**	**OpenWFE**	**Enhydra Shark**	**Staffware**	**Websphere**	**FLOWer**	**COSA**	**iPlanet**
Direct allocation	+	+	+	+	-	+	+	+	+	+	+
Role-based	+	+	+	-	+	+	+	+	+	+	+
Separat. of duties	+/-	+	+	-	-	-	-	+	+	+/-	+
Case handling	+/-	-	-	-	-	-	-	-	+	-	-
Retain familar	+/-	+	+	+	-	-	-	+	+	+	+
Capability-based	+	-	+	-	-	-	-	-	+	+	+
History-based	+/-	+/-	+	-	-	-	-	-	-	+/-	+
Organizational	+	+/-	+	-	-	-	+/-	-	+	+	+
Round robin	+/-	-	+	-	-	-	-	-	-	+/-	+/-
Random	+	-	+	-	-	-	-	-	-	+	+/-
Shortest queue	+/-	+/-	+	-	-	-	-	-	-	+	+/-
Mult. resources	+	?	?	?	?	?	?	?	?	?	?
Soft constraints	+	?	?	?	?	?	?	?	?	?	?
Combine patterns	+	?	?	?	?	?	?	?	?	?	?

Table 1. A patterns-based comparison. Our evaluation is in the columns and rows in *italics*; the rest of the table is the result of related research by others [11, 10, 12, 16]. + indicates full support, x/- indicates partial support (e.g. through coding a bit), and - indicates no support. B4P/WS-HT is short for *BPEL 4 People/WS-HumanTask*.

only the 11 patterns that specify who is *ultimately* allowed to perform a task are included (cf. our discussion in Section 2.1) plus 3 new patterns. As mentioned earlier the language did not set out to specify runtime negotiation rules because these are more elegantly handled by the process engine. To be deemed to have a good domain fit and satisfy the requirements, SOFTALLOC must have fully support (+) or partially support (+/-) most of the desired patterns, and it must support soft constraints.

The patterns-based analysis shows that SOFTALLOC does indeed have a good domain fit. In fact, SOFTALLOC supports or partially supports the entire set of patterns inside the scope of the work. In many of the other products +/- means that the pattern requires coding/workarounds *every time* it is used; in SOFTALLOC +/- means that an interface to the existing systems of the company has to be coded *once* to get full support.

Some limitations apply to this methodology: whereas our language fares quite well in the comparison, the patterns-based analysis itself inadequately captures the expressive power of our language. The last three rows (*Multiple resources*, *Soft constraints*, 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. This clearly shows the limitation of approaches with some finite set of patterns chosen without any particular guiding principle other that what is available in current products. Similarly, the patterns-based analysis does not mention if patterns can be combined and if so with what constraints.

5 Related work

In the introduction we stated that we aim for a work allocation language that "(a) supports soft constraints, (b) plugs into any business process language in an aspect-like manner, (c) can be used with a wide range of platforms/legacy systems, (d) allows rules on many organizational levels to be combined, and (e) runs fast enough to allocate and reallocate at runtime". In examining the related work it is useful to keep these key features in mind:

Allocation does have some similarities with the scheduling done in a multiprocessor system, but in the case of work allocation the processors (i.e. the human resources) are rarely homogeneous. In this way the problem area seems closer related to Grid scheduling, and indeed some interesting ideas have cropped up in the Grid scheduling field (makespan reduction [2], algorithms/heuristics [17]). However, these either do not support soft constraints (a) or require full opmization (e).

We have already discussed the work on resource patterns by Russell *et al.* [11]. Senkul and Toruslu [13] have suggested a simple allocation language, which in their proposal is translated into the constraint language *Oz* and solved by a constraint solver. The approach solves the entire workflow (or the entire set of running workflows) in one go, and it is therefore not clear how the approach would accommodate runtime flexibility whilst being scalable—in other words it does not have feature (e). Another approach uses *defeasible logic* [4], and *BPEL 4 People* has attempted to make *WS-BPEL* better suited for human-intensive workflows [6], but these do not address soft constraints (a) and composition (d).

There exists a variety of business rule languages (e.g. OMG's SVBR[9]) and thus it would seem obvious to take one of these as a starting point as it naturally plugs in many contexts (b,c) and composes well (d). As we examined these languages we found that there was no direct facility for soft constraints (a). It could be mimicked in some cases, but at the cost of significant extra complexity.

Ways of solving *NP*-hard scheduling problems have been studied for decades in operations research, where the survey by Ernst is a good place to start [3]. Although such

approaches have superior expressiveness in modeling soft constraints (a), they are markedly more complicated to use, even for programmers, and they cannot generally guarantee optimal solutions in *P*-time (e).

Interdisciplinary papers have proposed auction/game theory-based [7] and *AI*-based [1] approaches to work allocation. These algorithms behave as dynamic systems and thus usually adapt very well to shifts in the supply and demand of tasks. However, they do not consider soft constraints directly (a).

6 Conclusion and future work

The patterns-based analysis shows that SOFTALLOC can express all patterns for which it was designed and all examples that were deemed necessary. Industry expert interviews further supported the conclusion that SOFTALLOC has a good fit to human-intensive workflows. The use of soft constraints has proven very beneficial, and as intended the language integrates with any system we have seen so far. The language was not built for direct use by business process designers/analysts. Instead a GUI (both for users and allocation rule designers) is being developed in the production setting where the language is to be used. The GUI coupled with the textual form presented in this paper will allow both programmers and domain specialists to use the language in their preferred way and support conflict detection.

Based on the discussion of resource patterns it would be interesting to construct a collection of soft constraint patterns or even develop new measures of expressiveness to be able to benchmark languages.

More benefits are yet to be reaped: by capturing work allocation rules directly, performance simulation can be used to identify bottlenecks, estimate capacity requirements, and suggest what resources to add. This is an important improvement over previous systems, where the lack of integration made performance analysis a non-routine job requiring specialized skills. In systems where work allocation rules are captured in a general-purpose language, workflow performance simulation is often infeasible.

Another promising idea is to leverage runtime statistics to improve the allocation optimization. Runtime statistics could include average completion time probability of task type per agent, inferred probability of delay, etc. All these statistics will result in more soft constraints that the scheduler can use in conjunction with the user-specified ones.

Acknowledgements

The authors would like to thank Shriram Krishnamurthi and the reviewers for some very useful comments.

References

[1] S. Abdallah and V. Lesser. Learning the task allocation game. In *AAMAS '06*, pages 850–857, New York, 2006. ACM Press.

[2] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Task scheduling strategies for workflow-based applications in grids. In *CCGRID '05*, pages 759–767. IEEE Computer Society, 2005.

[3] A. T. Ernst, H. Jiang, M. Krishnamoorthy, and D. Sier. Staff scheduling and rostering: A review of applications, methods and models. *Eur. J. Op. Res.*, 153, 2004.

[4] G. Governatori, A. Rotolo, and S. Sadiq. A model of dynamic resource allocation in workflow systems. In *ADC '04*, pages 197–206. Australian Computer Society, Inc., 2004.

[5] Y. Hamadi and C.-G. Quimper. The smart workflow foundation. Technical Report MSR-TR-2006-114, Microsoft Research, November 2006.

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

[7] E. Koutsoupias. Selfish task allocation. *Bulletin of EATCS*, 81:79–88, 2003.

[8] OASIS Open, Inc. *WS-BPEL 2.0 Committee Specification*, May 2006.

[9] OMG. Semantics of business vocabulary and business rules (SBVR). Technical report, Object Management Group, Sep. 2006. Second SBVR Interim Specification.

[10] N. Russell, A. H. ter Hofstede, W. M. van der Aalst, and D. Edmond. newYAWL: Achieving comprehensive patterns support in workflow for the control-flow, data and resource perspectives. Technical Report BPM-07-05, Eindhoven University of Technology, 2007.

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

[12] N. Russell and W. M. P. van der Aalst. Work distribution and resource management in bpel4people: Capabilities and opportunities. In Z. Bellahsene and M. Léonard, editors, *CAiSE*, volume 5074 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2008.

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

[14] C. Stefansen and S. E. Borch. Using soft constraints to guide users in flexible business process management systems. *International Journal of Business Process Integration and Management*, 2008.

[15] C. Stefansen, S. Rajamani, and P. Seshan. A work allocation language with soft constraints. Technical report, SETLabs, Infosys Technologies Ltd., Bangalore, India, March 2008.

[16] P. Wohed, B. Andersson, A. H. ter Hofstede, and N. R. W. M. van der Aalst. Patterns-based evaluation of open source BPM systems: The cases of jBPM, OpenWFE, and Enhydra Shark. Technical Report BPM-07-12, Eindhoven University of Technology, 2007.

[17] S. Zhang, Y. Wu, and N. Gu. Adaptive grid workflow scheduling algorithm. In *Grid and Cooperative Computing Workshops*, volume LNCS 3252, pages 140–147, 2004.