HAL
archives-ouvertes.fr

Planning@SAP: An Application in Business Process Management

Joerg Hoffmann, Ingo Weber, Frank Kraft

▶ To cite this version:

Joerg Hoffmann, Ingo Weber, Frank Kraft. Planning@SAP: An Application in Business Process Management. 2nd International Scheduling and Planning Applications woRKshop (SPARK'09), Sep 2009, Thessaloniki, Greece. 2009. <inria-00430777>

HAL Id: inria-00430777
https://hal.inria.fr/inria-00430777

Submitted on 10 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Planning@SAP: An Application in Business Process Management

Jörg Hoffmann
SAP Research
Karlsruhe, Germany
joe.hoffmann@sap.com

Ingo Weber
University of New South Wales
Sydney, Australia
ingo.weber@cse.unsw.edu.au

Frank Michael Kraft
SAP
Walldorf, Germany
frank.michael.kraft@sap.com

Abstract

Business processes control the flow of activities within and between enterprises. Business Process Management is concerned, amongst other things, with the maintenance of these processes. In particular, it becomes ever more important to be able to quickly create modified processes for changed market conditions. We show that AI Planning can help with this, by automatically composing process skeletons. We formalize this as a particular form of planning with non-deterministic actions. Since there is no fixed "domain" – business processes may talk about almost anything – a major problem in applying the method is a practical way of obtaining the planning model. We show that, at SAP, one of the leading providers of enterprise software, one can obtain the models for free, by leveraging existing semi-formal models of software behavior. We finally show that, by arranging some known planning techniques in a suitable way, one can obtain tooling that solves practical examples in a matter of seconds, and that is hence suitable for use in a real-time BPM process modeling environment. Our prototype of such an environment is part of a research extension to the SAP NetWeaver platform.

Introduction

Business processes control the flow of activities within and between enterprises. Business Process Management (BPM) is concerned, amongst other things, with the maintenance of these processes. To minimize time-to-market in an ever more dynamic business environment, it is essential to be able to quickly create new processes. Doing so involves selecting and arranging suitable IT services from huge infrastructures such as those provided by SAP, which is a very difficult task. In the spirit of many recent lines of work on Web Service Composition (WSC), e.g. (Narayanan and McIlraith 2002; Pistore, Traverso, and Bertoli 2005; Hoffmann, Bertoli, and Pistore 2007; Hoffmann et al. 2008), we propose to annotate each IT service with a "semantic" description of its relevant properties, and to leverage these descriptions for helping with process creation, by Planning techniques.

This approach in itself is not particularly novel; even the application of Planning in the BPM context has been foreseen long ago already (Biundo et al. 2003). The exciting bit is that, at SAP, there is a near-perfect answer to the question that was left un-answered in pretty much every previous work we are aware of: *How to get the model?*

Coming up with the semantic annotations is one of the central issues in the Semantic Web area, and has also been recognized as a challenge for Planning (Kambhampati 2007). Blissfully ignoring this, two of the authors had been developing, since a while, PDDL-style planning languages and techniques for WSC.[1] Working towards the application of this technology within SAP, we were in for a surprise:

At SAP, over time, various semi-formal models of software behavior have been developed, both for documentation and for computer usage. Believe it or not, one of these models looks more or less like a PDDL domain description!

The "SAP PDDL" model is called *Status and Action Management (SAM).* It is an object-centered model, describing how the status of business objects may change depending on which "actions" – IT-level services affecting the objects – are executed. Based on SAM, it is easy to generate planning domains and tasks suitable for the automatic generation of processes that are characterized by their effect on the status of a given set of business objects. Such process generation underlies the task SAP customers are facing, in the BPM scenario described above.

We give a brief background on BPM and our general scenario in the next section. We then explain SAM, and discuss in detail a running example that we will use throughout the paper. Thereafter, we introduce a planning formalism suitable for capturing SAM, and we explain how we arranged a number of known planning techniques to obtain satisfactory empirical performance. We conclude with a discussion of the current status of our prototype, of open issues, and of the relevance of our findings to the Planning community.

Background

We briefly outline what Business Process Management is, and how AI Planning can help with it.

Business Process Management

Weske (2007) gives the following commonly used definition of what a *business process* is:

[1] The focus of this work was on dealing with restricted classes of ontology axioms, c.f. (Hoffmann, Bertoli, and Pistore 2007; Hoffmann et al. 2008).

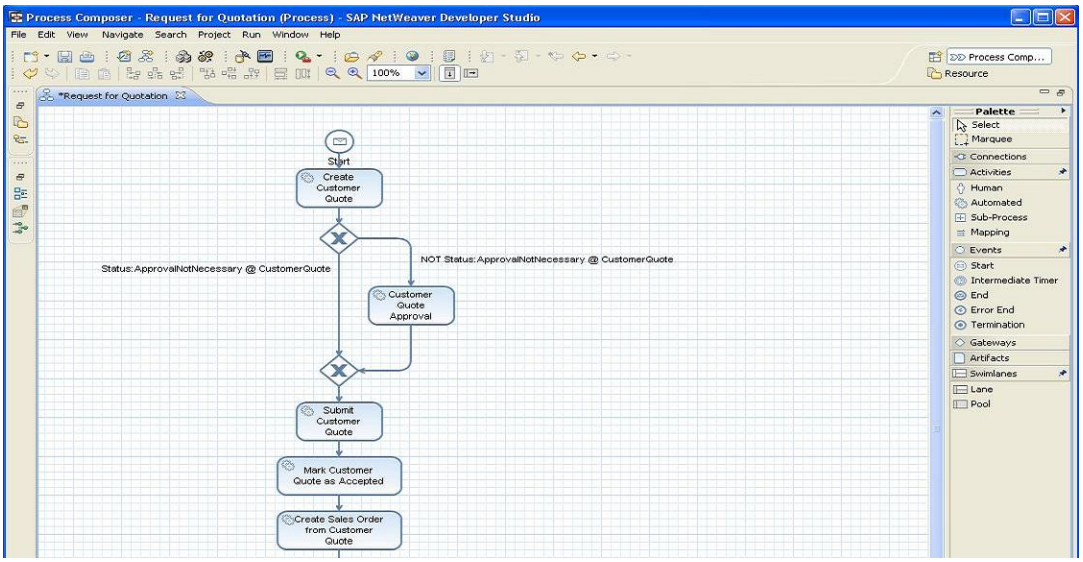

Figure 1: A screenshot of our BPM modelling environment prototype.

"A business process consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal."

In other words, business process models serve as an abstraction of the way enterprises do business. Technically, processes are control-flows often formalized as (particular kinds of) Petri nets, notated in human-readable syntax such as UML activity diagrams. *Business process management* (BPM) is a field of research, as well as an industry, aiming at understanding processes, configuring and implementing them in information technology (IT) systems, monitoring and analysing their execution, and re-designing them. A central activity in BPM is the modelling of business processes. This is done by humans, in suitable *BPM modelling environments*. Figure 1 shows a screen shot of the prototype we develop at SAP Research. The modelled process, a simple handling of customer quotes, is shown as a kind of flow diagram, in the wide-spread BPMN notation (OMG 2008).[2] Edges represent control flow, and rectangular boxes represent activities. Other boxes represent routing nodes; we tackle two kinds, namely XOR for alternative execution (marked by "x" as shown in the figure), as well as AND for parallel execution (marked by "+", not shown in the figure).

Today's business environment is increasingly dynamic. On the one hand, the requirements on business processes (such as legal and financial regulations) are subject to frequent and regional updates. On the other hand, market conditions change frequently and so must the business plans. It is hence of crucial importance to be able to adapt business processes, and to create new business processes, as quickly as possible. A major bottleneck here lies in the translation of high-level process models into implemented processes that can be run on the IT infrastructure. This step is very time-consuming since it requires intensive comunication between business experts (who model the processes) and IT experts (who implement them). If the IT infrastructure is from an external provider, then experts for that infrastructure (such as SAP consultants) usually need to be involved as well. This incurs significant costs for human labor, and potentially even higher indirect costs due to increased time-to-market. The basic idea of our application is to use AI Planning for composing processes automatically (to a certain extent), helping the business expert to come up with processes that are close to the IT infrastructure, and hence reducing the effort and costs associated with implementation.

[2]BPMN includes notations for hiararchical definition of processes, and for temporal constructs. Neither of these is supported by SAM, i.e., the SAP model we build on.

Planning in BPM

Service-Oriented Architectures are becoming increasingly popular for providing flexible IT infrastructures; in particular this holds true for the recent developments at SAP. One potential benefit of this flexibility is the ability to implement desired business processes more quickly, using Web services to implement the individual activities in the processes. Reaping this benefit is severely hindered by the complexity of the IT landscape. A way out that has become rather popular is to "semantically" describe each Web service, i.e., to describe its relevant properties in some declarative language (e.g. preconditions and postconditions), and to leverage these descriptions (e.g. by Planning techniques) for automation; see e.g. (Narayanan and McIlraith 2002; Pistore, Traverso, and Bertoli 2005; Hoffmann, Bertoli, and Pistore 2007; Hoffmann et al. 2008; Weber, Markovic, and Drumm 2008).

In our work, we focus on services described by pre/post conditions formulated as in PDDL-style languages. Obviously, planning based on such information cannot fully replace the human IT expert – the description of a service in terms of pre/post conditions will often be imprecise (not exactly represent all technical aspects of the service), and so the plan will not be guaranteed to work at IT-level. What

Action name	precondition	postcondition
Check CQ Completeness	CQ.archivation:notArchived(x)	CQ.completeness:complete(x) OR CQ.completeness:notComplete(x)
Check CQ Consistency	CQ.archivation:notArchived(x)	CQ.consistency:consistent(x) OR CQ.consistency:notConsistent(x)
Check CQ Approval Status	CQ.archivation:notArchived(x) AND CQ.completeness:complete(x) AND CQ.consistency:consistent(x)	CQ.approval:Necessary(x) OR CQ.approval:notNecessary(x)
CQ Approval	CQ.archivation:notArchived(x) AND CQ.approval:Necessary(x)	CQ.approval:granted(x)
Submit CQ	CQ.archivation:notArchived(x) AND (CQ.approval:notNecessary(x) OR CQ.approval:granted(x))	CQ.submission:submitted(x)
Mark CQ as Accepted	CQ.archivation:notArchived(x) AND CQ.submission:submitted	CQ.acceptance:accepted(x)
Create Sales Order from CQ	CQ.archivation:notArchived(x) AND CQ.acceptance:accepted(x)	CQ.followUp:documentCreated(x) (*)
Archive CQ	CQ.archivation:notArchived(x)	CQ.archivation:Archived(x)

Figure 2: The SAM model underlying our running example, modelling the behavior of "customer quotes" CQ. The postcondition marked (*) is associated also with a second kind of business object, "sales order", an instance of which will be created as a side effect of the action.

the planning can help with is choosing the correct combination of services, plus putting them together in a way that is likely to be suitable. Given that the main pain-point is the size of the IT landscape, this can potentially be quite useful, provided the following two requirements are met:

1. **Response times are (almost) instantaneous.** The planning will be in on-line interaction with the human process modeller, i.e., the modeller will wait for the planning result. This limits response time to human patience.
2. **The modelling overhead is low.** The benefit of the planning needs to be weighted against the overhead of designing the planning model.

Both points are important, but the second point is – somewhat in contrast to the traditional focus in AI Planning research – potentially the more important one.[3] The model of the "domain", i.e., of the services infrastructure, is subject to frequent change. The model of individual planning tasks, i.e., initial state and goals, must be created on-line by the user. This must be possible in a matter of seconds, and without in-depth knowledge of the IT infrastructure. The great news we share herein is that, at SAP, both can be tackled based on a pre-existing model of software behavior.

Where to get the model?

As outlined in the introduction, we build on a model called *Status and Action Management (SAM)*. SAM documents the behavior of actions – IT services – affecting the status of business objects. "Status" here is represented in terms of a value assignment to a set of "status variables". SAM defines for each action a precondition and a postcondition, stating the required status of the relevant business objects, and how that status changes when executing the action, respectively. The original purpose of this model is, simply put, to provide a declarative way of detecting applicable actions. SAP applications check the current status against the SAM model, and provide to the user only those actions whose preconditions are satisfied. This guards against implementation bugs in the applications, since it is easier to maintain the action requirements on the level of SAM, than on the level of the actual program code. Indeed, execution of actions on objects not satisfying the requirements (e.g., processing a customer quote that is incomplete) may have all sorts of subtle and harmful side effects. This is one of the major problems in maintenance of SAP applications. Further, SAM serves for decoupling the implementation from updates to the available actions: instead of re-implementing the application, it suffices to adapt the SAM model.

While SAM's original purpose is quite different from planning, the similarity to PDDL is striking. As we show herein, this can be exploited for helping with process creation. SAP customers frequently need to implement new/adapted processes, making use of the many IT services the SAP systems already provide. The services are described in SAM; these descriptions are not visible to the customers – because they contain confidential information about SAP infrastructures – but they can be used for generating processes, using AI Planning. It then suffices for the customer to express the desired process in terms of business objects and their state changes – a language he/she is likely to speak. That said, the process will be a *skeleton* only: the control flow may need adaptations due to particular customer policies; and customers may wish to integrate additional services implemented in their own IT infrastructure.

For illustration, Figure 2 gives the SAM model for "customer quotes" CQ, our running example.[4] The intended meaning of (disjunctive) preconditions is exactly as in planning. The same goes for non-disjunctive effects.

[3] See (Kambhampati 2007) for related observations in the context of the Semantic Web.

[4] At SAP, SAM models are represented graphically; we use an equivalent table write-up. For privacy, the shown object and model are *not* part of SAM as used at SAP. The SAM models are similar.

Disjunctive effects are more complicated. What SAM models here is that the action modifies the status variables, and that several outcomes are possible; which outcome actually happened will be visible to the SAP application at execution time. What SAM does *not* model is that the outcome depends on the properties of the relevant object prior to the action application. For example, the outcome of "Check CQ Completeness" is fully determined by the contents of the object CQ. These contents – which may be vast, with 100s or even 1000s of data fields for a single object – are abstracted in SAM, making the action non-deterministic from its perspective. Further, the processes do not have control over the objects. At any point in time outside events may affect their content. Hence repeated consecutive application of, e.g., the "Check CQ Completeness" action does not always yield the same outcome – it may be that CQ has been modified in the meantime. Overall this means that, at the planning level, SAM disjunctive effects correspond to non-deterministic actions, different instances of which are independent, and whose outcome is observed at execution time.

We will detail the translation of SAM models into planning actions in the next section, after introducing our formalism. The initial state is trivial to generate since SAM explicitly provides an initial value for each status variable. Note here that, while the content of objects may change externally, this is not the case for the status variables – they are only changed by explicit application of SAM actions.

Regarding the goal and the definition of plans, matters are more complicated again. The reader will have noticed that the SAM model of Figure 2 does not allow "strong plans" that always lead to success: checking completeness or consistency can always result in a negative outcome that forbids successful processing. To address this, one can define more complicated goals, or a weaker notion of plans. The former is impractical because the goals must be specified by the user. Specification of complex goals requires familiarity with SAM/with the underlying SAP system, which contradicts our value proposition. We hence settle for a weak notion of plans, allowing dead end states as final nodes in the solution tree. At the process level, this means that no explicit failure handling is provided. To avoid endless repetition of non-deterministic actions (e.g., checking and re-checking completeness), we simply impose an upper bound (currently, 1). All this will be detailed in the next section. What our approach accomplishes is that, for the user, specifying the goal is exceedingly simple. *All that is required is to give the desired attribute values: CQ.followUp:documentCreated(x) and CQ.archivation:Archived(x) in the case of Figure 2. In our prototype, this is done in simple drop-down menus.*

Planning Formalism

According to what has been discussed in the previous section, we use the following planning formalism to capture SAM. Planning tasks are tuples (X, dA, ndA, I, G). X is a set of *variables*; each $x \in X$ is associated with a finite domain $dom(x)$. dA and ndA are sets of deterministic and non-deterministic *actions*, respectively. Each $a \in dA \cup ndA$ takes the form (pre_a, eff_a) with pre_a, eff_a being partial variable assignments; eff_a is interpreted differently depending on whether $a \in dA$ or $a \in ndA$ (details below). I is a variable assignment representing the *initial state*, and G is a partial variable assignment representing the *goal*. A *proposition* is is a pair (x, v) where $x \in X$ and $v \in dom(x)$. We identify (partial) variable assignments with sets of propositions in the obvious way.

A *state* s is a variable assignment. An action a is *applicable* in s iff $pre_a \subseteq s$. If f is a partial variable assignment, then $s \oplus f$ is the variable assignment that coincides with f where f is defined, and that coincides with s elsewhere. Using these notions, we can define what a plan is. Assume that s is a state and that $ndA_{av} \subseteq ndA$ is a subset of the non-deterministic actions of the task. We now define under what conditions a tree T of actions *solves* (s, ndA_{av}) *in* (X, dA, ndA, I, G). The set ndA_{av} here is needed for imposing that non-deterministic actions may occur only once (c.f. the discussion in the previous section). Formally, T solves (s, ndA_{av}) in (X, dA, ndA, I, G) iff either:

1. T is empty and $G \subseteq s$; or
2. the root of T is $a \in dA$, a is applicable in s, the tree node a has exactly one son, and the tree rooted at that son solves $(s \oplus \mathit{eff}_a, ndA_{av})$; or
3. the root of T is $a \in ndA_{av}$, a is applicable in s, the tree node a has one son labelled with p for every $p \in \mathit{eff}_a$, a has no other sons, and each $(s \oplus \{p\}, ndA_{av} \setminus \{a\})$ is either (i) unsolvable or (ii) solved by the sub-tree of T rooted at the respective son, where (ii) is the case for at least one of the sons.

A *plan for* (X, dA, ndA, I, G) is a tree that solves (I, ndA).

Item 1 of this definition is clear. Item 2 states the usual meaning of deterministic actions. Item 3 essentially says that, for every possible outcome of the non-deterministic action, we must either find a plan or prove unsolvability. At least one son must be solvable. The syntax of non-deterministic actions is restrictive, allowing just one proposition per alternative outcome. This suffices to model SAM; our implementation does not make this restriction. The formalism has three remarkable features:

- **Non-deterministic effects vs. observations.** These two are joined, because SAM does not distinguish them. The object stati are always fully known. The outcome of non-deterministic actions is directly observed at exection time.
- **Failed nodes.** We allow solution trees containing unsolvable leaf nodes, as long as below every node there is at least one solved leaf. This amounts to implicit treatment of failure, and keeps the goals sufficiently simple.[5]

[5]From a more general perspective, allowing failed nodes appears to make sense because, in practical planning problems of the BPM and Web services areas, it is often the case that actions may fail without the possibility of recovery (just think "insufficient credit card balance"); see (Mediratta and Srivastava 2006) for a related investigation. This notwithstanding, it is unclear to us at the time of writing to what extent and under which conditions failed nodes are adequate in BPM. If explicit failure handling is required, then in all likelihood they aren't (unless a failure can be handled simply by marking it up as being one).

- **Upper bound on repeating non-deterministic actions.** We allow each non-deterministic action only once. This serves to avoid repetitions, which are unlikely to be useful for the process skeleton we wish to create.[6] Importantly, having an upper bound is also instrumental for the definition of plans to make sense. In allowing sons of non-deterministic actions to be "unsolvable", the definition of solvability recurses on itself. While such recursion occurs also at other points in the definition, at those points termination is guaranteed because the tree T considered is reduced by at least one node. For "unsolvable" sons of non-deterministic actions, the upper bound guarantees termination. In each recursion step, the set of available non-deterministic actions is diminished by one.[7] Hence the recursion will eventually terminate in a planning task with only deterministic actions. Without a bound, the recursion step may result in the same planning task over again, allowing the construction of examples which, according to the above definition, are solvable iff they are unsolvable.

It remains to explain how SAM models are translated into our formalism. We explain this simply by translating the running example. We create one variable for each attribute. Precisely we set $X := \{Arch, Compl, Cons, Appr, Subm, Acc, FoUp, InitCQ, InitSO\}$. All but the last two of these are the obvious abbrevations of the attributes mentioned in Figure 2 (e.g. $Arch$ stands for archivation). $Init$ variables are introduced into every SAM-translation, for every kind of object appearing in it, to model creation of objects (which is implicit in SAM).[8] The domain of each of $Arch$, $Compl$, $Cons$, $Subm$, Acc, $FoUp$, $InitCQ$, $InitSO$ is $\{yes, no\}$; this serves to abbreviate the various names used for the respective attribute values in Figure 2. The domain of $Appr$ is $\{nec, notNec, grant\}$.

In what follows, for brevity we use propositional notation for yes/no valued attributes, e.g., writing $\neg FoUp$ instead of $(FoUp, no)$. From the SAM model, we get the initial values $I = \{\neg Arch, \neg Compl, \neg Cons, (Appr, nec), \neg Subm, \neg Acc, \neg FoUp, \neg InitCQ, \neg InitSO\}$. The user selects the goals in a drop-down menu, resulting in $G = \{FoUp, Arch\}$. The set dA constructed from the SAM model consists of:

- Create CQ: $(\emptyset, \{InitCQ\})$
- CQ Approval: $(\{InitCQ, \neg Arch, (Appr, nec)\}, \{(Appr, grant)\})$
- Submit CQ: $(\{InitCQ, \neg Arch, (Appr, notNec)\}, \{Subm\})$
- Submit CQ: $(\{InitCQ, \neg Arch, (Appr, grant)\}, \{Subm\})$
- Mark CQ as Accepted: $(\{InitCQ, \neg Arch, Subm\}, \{Acc\})$
- Create Sales Order from CQ: $(\{InitCQ, \neg Arch, Acc\}, \{FoUp, InitSO\})$
- Archive CQ: $(\{InitCQ, \neg Arch\}, \{Arch\})$

The only non-trivial aspect of this translation regards the Submit CQ action, which has been split into two actions bearing the same name. The action as per SAM has the disjunctive precondition "CQ.archivation:notArchived(x) AND (CQ.approval:notNecessary(x) OR CQ.approval: granted(x))". This is transformed into DNF and split into two actions, one for each disjunct in the DNF. The set ndA consists of:

- Check CQ Completeness: $(\{InitCQ, \neg Arch\}, \{Compl, \neg Compl\})$
- Check CQ Consistency: $(\{InitCQ, \neg Arch\}, \{Cons, \neg Cons\})$
- Check CQ Approval Status: $(\{InitCQ, \neg Arch, Compl, Cons\}, \{(Appr, nec), (Appr, notNec)\})$

This should be self-explanatory. Figure 3 shows a plan.

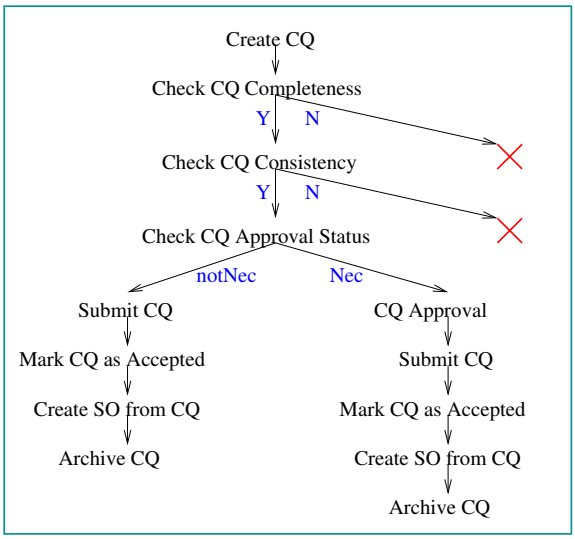

Figure 3: A plan for the running example.

Algorithms and Empirical Performance

As was mentioned, we can obtain an effective solver for practical examples by suitably arranging (and slightly adapting) some known planning techniques. We explain those in what follows, and where applicable point out the effect they have on performance. We first consider the search procedure. We then explain, in this order, our heuristic function, a relevance pruning mechanism, and a plan post-processor transforming plans into the workflows to be delivered back to the user.

[6] One might argue whether the bound 1 is too strict. As best we can tell from our experience with SAM, it is not. Recall that the reasons behind the outcomes are abstracted away anyhow, so applying other actions in between can never help, from the perspective of the planner.

[7] More generally, at least one upper bound decreases by one.

[8] In our implementation, we actually allow online creation of (potentially infinitely many) new objects (Hoffmann et al. 2008). We omit this here for the sake of simplicity.

Search Procedure

We use a variant of AO* forward search. The search space is an AND-OR tree whose nodes are, alternatingly, states (OR nodes) and actions (AND nodes). The OR children of a state correspond to the applicable actions. The AND children of actions correspond to the alternative outcomes. For deterministic actions, there is a single child so the AND node trivializes. The only change we make to AO* is that, on top of the usual "node solved" markers, we also propagate "node failed" markers. An OR node is marked as failed if either its heuristic value (see next section) is infinite, or if all its children are marked failed; the node is marked as solved if either its heuristic value is 0, or if one of its children is marked solved. An AND node is marked as failed if all its children are marked failed; the node is marked as solved if all of its children are marked either failed or solved, and at least one child is marked solved. Obviously, this corresponds exactly to the definition of plan given in the previous section. Note that the only difference between "AND" and "OR" nodes in this framework is that OR nodes are solved as soon as one of their sons is, while for AND nodes we have to prove failure of (or solve) all the other sons.

Heuristic Function

We devise a simple variant of the FF heuristic function (Hoffmann and Nebel 2001). For the non-deterministic actions, we adopt the simplistic approach of acting as if we could choose the outcome. We compile each $a \in ndA_{av}$ – the non-deterministic actions that are still available at the respective point in the tree – into the set of deterministic actions $\{(pre_a, \{p\}) \mid p \in \mathit{eff}_a\}$. While this yields rather simplistic goal distance estimates, we found that it works well in our context. As in FF, we use the relaxed plans not only to obtain the goal distance estimates, but also to restrict the action choice to those that are "helpful".

One important aspect of the heuristic function is that, just as usual, it may stop without reaching the goals in the relaxed planning graph, which obviously proves the evaluated state to be unsolvable – there is not even a single sequence of action outcomes that leads to success. The heuristic function returns ∞ in this case, which is of paramount importance for our algorithm to be able to mark anything as "failed": states evaluated with ∞ are the only source of such markers in our implementation, where we do not check for duplicates.[9]

The reader will have noticed that the heuristic function is very optimistic, testing only a fairly strong sufficient criterion for unsolvability. However, fortunately (and probably not unexpectedly after reading up to this point), in our setting this sufficient criterion seems entirely appropriate. We have not yet found a single case of an unsolvable state that was not identified by the heuristic function. Obviously, the heart of this is that the cause for failure always is some non-deterministic action a with unwanted outcome, resulting in a variable value contradicting the preconditions of actions that necessarily lie on any path to the goal. Note the slightly more subtle point that, since we act as if we could select non-deterministic outcomes as we like, we could use the culprit non-deterministic action a inside the heuristic to establish the outcome we want. However, we allow each non-deterministic action only once, so a won't be available to the heuristic function.[10] For illustration, consider the "no" outcomes of "Check CQ Completeness" and "Check CQ Consistency" in Figure 3. Clearly, for the corresponding states the relaxed planning graph will not reach the goals because we no longer have these actions available, and the mandatory action "Check CQ Approval Status" never becomes applicable. Speaking in the terms of the application: if the Customer Quote is incomplete or inconsistent, then we cannot check its approval status.

We ran a few tests to confirm that the heuristic function yields an empirical advantage over blind search. More precisely, we tested against a variant of our planner with a heuristic function that returns ∞ if the relaxed planning graph does not reach the goals, and that returns 1 otherwise (all actions are explored, i.e., helpful actions are turned off). This is because any sensible SAM planning example that we can think of involves failed nodes, and the heuristic function is the only source of such nodes in our implementation. As one would expect, the blind version is much slower. Running the two methods on a (real) SAM planning task similar to our running example, if we include in the input only the services/actions that will appear in the solution, then with heuristic we take 0.25 seconds but without we take 1.75 seconds. Adding additional actions – of which SAM specifies 2700 – into the input, the picture becomes quickly more drastic. With only the 20 actions associated with the relevant business objects, the times are 0.28 vs. 5.73. Adding just 50 more actions, the planner takes more than an hour without heuristic, vs. 2.89 seconds with it.

Relevance Pruning

The FF-style heuristic function is not sufficient to obtain satisfactory performance: the planner input in practice will contain all 2700 SAM actions, and with that runtime is prohibitive (detailed results follow below). A natural question to ask is: are the 2700 actions all relevant to the task at hand? Expectedly, the answer to this one is "no", at least in the cases we tried. Typically, a planning task in SAM requires changing the status of a small number of business objects. The full SAM model reflects a very large number of (types of) business objects, and most of those are not relevant to the task. Now, it is well known, and happens also in our context, that the FF heuristic, in particular helpful actions pruning, largely precludes such irrelevant actions from affecting the search space. However, the 2700 actions, and the associated variables dealing with the status of the respective business objects, do result in a huge blow-up of the relaxed planning graphs, and hence dramatically decrease the runtime efficiency of the heuristic itself.

The obvious answer to the above is to perform an approximate relevance analysis before planning starts. In our cur-

[9] The latter is just due to historical reasons, since the code was developed for a more general planning framework in which duplicate checking is non-trivial.

[10] With more general bounds, a will no longer be available once the bound is exhausted.

rent implementation, we use the following trivial procedure: (1) start with the set G of predicate names containing all goal predicates; (2) take G' to be G plus the set of all predicates that appear in the preconditions of actions whose effects mention predicates in G; (3) if $G = G'$ then stop, else set $G := G'$ and goto (2). Clearly, actions that do not affect predicates in the final set G can be removed safely.

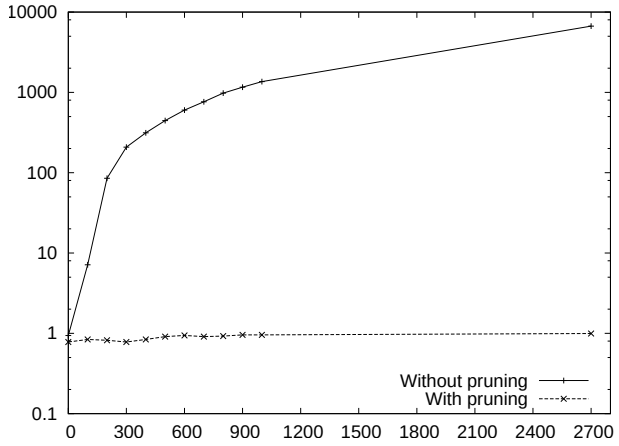

Figure 4: Empirical effect of relevance pruning.

In practice, this simple pruning technique is vastly effective. Figure 4 shows empirical data for the same SAM task discussed above, scaling the number of additional actions: the x-axis shows the number of actions in the input that will not be part of the solution. The pruning technique filters out all but about a dozen irrelevant actions – simply because they are associated with irrelevant business objects. Hence we get almost constant runtime, in stark contrast to the planner not using the pruning.[11]

There are two important remarks to be made at this point, with good and bad news. The good news is that the example for which we show data here is *not* a small toy problem. The example is of a typical size for SAM. So we can reasonably expect similar performance across the entire application. The bad news is that, to a considerable degree, the drastic performance of the pruning is due to current shortcomings of SAM. There exist many interactions between different types of business objects, at IT implementation level, that are not reflected in the current SAM model because they are not relevant to SAM's original purpose. However, the interactions might very well be relevant for composing higher quality processes. We are currently investigating this. Of course, it is not foreseeable how our planner will perform on an enriched SAM model including the interactions.

Post-Processing Plans

A final challenge is to turn plans into workflows, i.e., to post-process the plan, bringing it into the form expected by the business process modeller. To do so, we perform a sequence of steps involving failed nodes, XOR constructs, and parallelism. We end up with the process depicted in Figure 5. When following the subsequent explanations, it will be instructive to compare this to Figure 3.

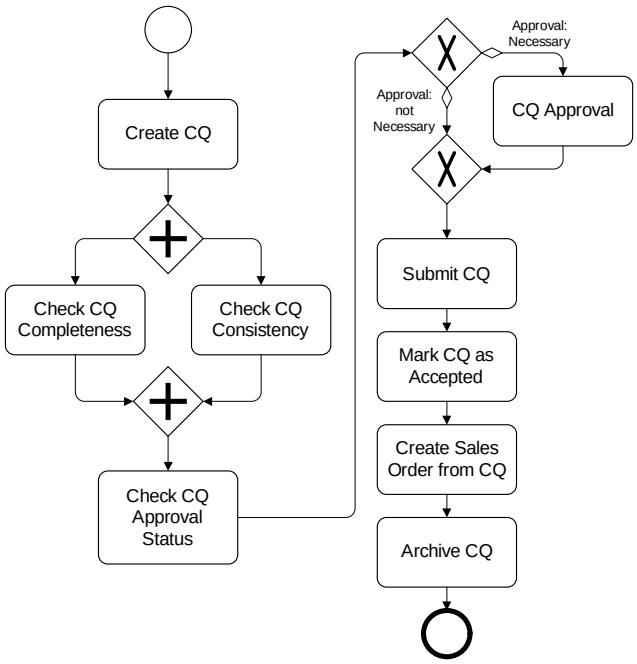

Figure 5: Final process composed for the running example.

First, since there is no explicit failure handling, we remove each failed node together with the edge leading to it. In our running example, Figure 3, this concerns the "N" branches of "Check CQ Completeness" and of "Check CQ Consistency". Next, we need to re-unite XOR branches using XOR joins – ideally in a way that avoids redundancies in the process – and we need to distinguish between checking a property and directing the control flow. The latter is easy. We consider, in turn, each node that has more than 1 son.[12] We replace each such node with an activity node that bears the same name, followed by an XOR split. In the example, this concerns "Check CQ Approval Status". To re-unite XOR branches, we now traverse the plan tree in depth-first order. At each node n_2 encountered, we test whether the sub-tree T rooted at n_2 is identical to a sub-tree, rooted at a node n_1, that we encountered previously (that test is performed efectively via a hashing function). If so, we create a new XOR join node n'; we re-direct the incoming edges of n_1 and n_2 to point to n' instead; and we connect one copy of T to the outgoing edge of n'. We then continue the depth-first traversal behind n'. In the example, the first nodes n_1 and n_2 we find are the two copies of "Submit CQ".

Upon completion of the above algorithm, we insert a start node as the new root of the plan. If there is more than one leaf node, then we join all these leaves via a new XOR join. We thereafter insert an end node as the new (only) leaf of the plan. Note that, here and in the above, the XOR joins

[11] The extent of the blow-up without pruning appears excessive, and is probably at least in part due to our current implementation, and due to the creation of new objects on the fly. We are currently looking into improving this.

[12] Note that this no longer includes non-deterministic actions all but one of whose outcomes directly lead to failure, like "Check CQ Completeness" and "Check CQ Consistency" in our example.

inserted always result in a workflow that is *sound* according to the usual criteria (Aalst 1997), essentially because there is no parallelism that the XOR joins could interfere with.

A BPM modeller also expects that, where possible, workflows perform activities in parallel – an essential property to ensure effective execution. This intention corresponds well to those behind the various kinds of parallelism investigated in planning. Still the correspondence isn't exact in the details. We wish to cater for sequences of actions running in parallel. On the one hand, this is more general than Graphplan-style planning with sets of mutually non-interfering actions. On the other hand, it is less general than temporal planning since it does not distinguish action durations and does not allow consideration of arbitrary start/end time points. Partial-order planning is most closely related in that it considers the relative order of actions. Methods for post-processing total-order plans into partial-order plans have been investigated in depth already, see e.g. (Bäckström 1998). For our particular purpose, we have designed a simple method based on finding non-interacting sub-sequences of actions in between the XOR splits and joins that were introduced previously. The method is not yet implemented, but obviously will take negligible runtime cost. We omit its description for lack of space.

Discussion

As mentioned, our techniques are implemented within the BPM modelling environment shown in Figure 1. Along with several other research prototypes, they form a research extension (called Galaxy) to the SAP NetWeaver platform. This concerns in particular the NetWeaver CE Process Composer, NetWeaver's BPM modelling environment.

Galaxy is currently in the initial steps towards pilot customer evaluation. It must be said that it is still a long way towards actual commercialisation. The potential pilot customer has not made a firm commitment yet, and there are technical and political difficulties inside SAP: Galaxy, respectively the implementation of the transactions modelled in SAM, reside in (largely) separate parts of the SAP software architecture. Besides, it is of course as yet unclear what the outcome of a customer evaluation will be. Hence the most important open question is: Do our techniques bring real added value for SAP customers? Apart from the preliminary status of our prototype, answers to this question will be difficult to quantify. For privacy reasons, they may be impossible to come by. Our impression based on demos at SAP is that, at least for quick experimentation and kick-starting the process design, the technique will be quite handy.

On the algorithmic side, the story thus far appears to be closed, with existing planning techniques being fully satisfactory. However, as mentioned, the current SAM model falls short of reflecting several dependencies *within* business objects – for some status variables, the value is a function of other variables' values – as well as *across* business objects – some transitions can only be taken synchronously with transitions of other objects. These shortcomings are currently being addressed in a research activity lead by SAP Research Brisbane, with the purpose of more informed model checking based on SAM models. We expect to be able to build on these extensions for improved planning. Naturally, especially regarding the cross-object dependencies, additional/modified planning techniques may be required for satisfactory performance of such planning.

Directions for future research also encompass plan quality criteria, i.e., allowing the user a trade-off between different plans that are valid for the same task, as well as using the technique not for composing plans from scratch but instead for filling in "place-holders" in partially specified processes. Neither of these should be difficult to realize, based on known results from planning. But their suitable configuration for customer use is tricky and requires more concrete insights into the abilities and needs of such customers.

We believe that our work is good news for the Web Service Composition area. It is reassuring that pre/post condition based description of services is not just an artefact of AI researchers eager for new playgrounds – the same paradigm was developed independently at SAP, with no AI influence whatsoever. More ambitiously, here might be an application where this form of WSC brings real business value. If so, this would be good news for AI Planning as a whole.

References

Aalst, W. 1997. Verification of Workflow Nets. In *Application and Theory of Petri Nets 1997*.

Bäckström, C. 1998. Computational aspects of reordering plans. *JAIR* 9:99–137.

Biundo, S.; Aylett, R.; Beetz, M.; Borrajo, D.; Cesta, A.; Grant, T.; McCluskey, L.; Milani, A.; and Verfaillie, G. 2003. PLANET Technological Roadmap on AI Planning and Scheduling. http://planet.dfki.de/service/Resources/Roadmap/Roadmap2.pdf.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Hoffmann, J.; Bertoli, P.; and Pistore, M. 2007. Web service composition as planning, revisited: In between background theories and initial state uncertainty. In *Proc. AAAI'07*.

Hoffmann, J.; Weber, I.; Scicluna, J.; Kaczmarek, T.; and Ankolekar, A. 2008. Combining scalability and expressivity in the automatic composition of semantic web services. In *Proc. 8th International Conference on Web Engineering (ICWE'08)*.

Kambhampati, S. 2007. Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain models. In *Proc. AAAI'07*.

Mediratta, A., and Srivastava, B. 2006. Applying planning in composition ofweb services with a user-driven contingent planner. IBM Research Report RI 06002.

Narayanan, S., and McIlraith, S. 2002. Simulation, verification and automated composition of web services. In *Proc. WWW'02*.

OMG. 2008. Business Process Modeling Notation, V1.1. http://www.bpmn.org/. OMG Available Specification, Document Number: formal/2008-01-17.

Pistore, M.; Traverso, P.; and Bertoli, P. 2005. Automated composition of web services by planning in asynchronous domains. In *Proc. ICAPS'05*.

Weber, I.; Markovic, I.; and Drumm, C. 2008. A conceptual framework for semantic business process configuration. *Journal of Information Science and Technology* 5(2):3–20.

Weske, M. 2007. *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag.