SAP Speaks PDDL

Joerg Hoffmann, Ingo Weber, Frank Kraft

SAP Speaks PDDL

Jörg Hoffmann*
INRIA
Nancy, France
joerg.hoffmann@inria.fr

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

In several application areas for Planning, in particular helping with the creation of new processes in Business Process Management (BPM), a major obstacle lies in the modeling. Obtaining a suitable model to plan with is often prohibitively complicated and/or costly. Our core observation in this work is that, for software-architectural purposes, **SAP is already using a model that is essentially a variant of PDDL.** That model describes the behavior of Business Objects, in terms of status variables and how they are affected by system transactions. We show herein that one can leverage the model to obtain (a) a promising BPM planning application which incurs hardly any modeling costs, and (b) an interesting planning benchmark. We design a suitable planning formalism and an adaptation of FF, and we perform large-scale experiments. Our prototype 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 transactions from huge infrastructures such as those provided by SAP. That is a very difficult and costly task. A well-known idea is to annotate each IT transaction with a planning-like description of its relevant properties, enabling AI Planning tools to compose (parts or approximations of) the desired processes fully automatically. Variants of this idea have been explored in Web Service Composition, e.g. (Narayanan and McIlraith 2002; Pistore, Traverso, and Bertoli 2005).

Planner performance is important in such an application: typically, the user – a business person wishing to create a new process – will be waiting online for the planning outcome. But the crucial question is: *How to get the planning model?* To be useful, the model needs to capture, in a precise way and at the right level of abstraction, the relevant properties of a huge IT infrastructure. Designing such a model is so costly that one will need good arguments indeed to persuade a manager to embark on that endeavor. (This relates to modeling of Web Services, see e.g. (Kambhampati 2007).)

When we first looked into applying planning in BPM at SAP, we did not expect to find a solution to the modeling problem. Imagine our surprise when we discovered that SAP had already designed the model we were looking for.

At SAP, various models of software behavior have been developed, mostly for software-architectural purposes. One of these models is essentially a variant of PDDL.

The "SAP PDDL" is called *Status and Action Management (SAM)*. SAM describes how "status variables" of Business Objects (BO) change their values when "actions" – IT-level transactions affecting the BOs – are executed. SAM is extensive, covering 404 kinds of BOs with 2418 transactions.

BOs in full detail are vastly complex. A single BO may contain 1000s of data fields. SAM describes their behavior at a level of abstraction that corresponds to the language of business users. SAM was originally intended (only) to provide a declarative way of checking transaction preconditions. Using SAM for planning, we obtain technology that automatically generates processes characterized by their effect on the status of BOs. Such process generation underlies the task SAP customers are facing, in the BPM scenario described above. The *only* cost of using our technology is the time it takes business users to specify the desired status changes – in their own language! We implemented a GUI in which that can be done using simple drop-down menus.

Translating SAM into PDDL is straightforward except for some subtleties in the treatment of non-deterministic actions. We design a suitable planning formalism, and we adapt FF (Hoffmann and Nebel 2001) accordingly. Our GUI and its back-end are integrated as a research extension into the commercial SAP NetWeaver platform. Initial steps towards pilot customer evaluation have been taken, but it is still a long way towards actual commercialization. An additional benefit of our work is a new, highly realistic, benchmark domain for Planning. *An anonymized PDDL version of SAM is publicly available.*[1] We run large-scale experiments with FF and show that, while many tasks can be solved sufficiently quickly, significant challenges remain.

*Work performed while authors were employed by SAP.

[1] http://www.loria.fr/∼hoffmanj/SAP-PDDL.zip

Background

We briefly outline what BPM is, and how AI Planning can help with it. Weske (2007) gives the following commonly used definition of what a *business process* is:

"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."

That is, business processes serve as an abstraction of the way enterprises do business. Technically, processes are control-flows often formalized as Petri nets, notated in a human-readable format. *Business process management* (BPM) aims at configuring and implementing processes in IT systems, monitoring and analyzing their execution, and re-designing them. A central activity in BPM is the creation and/or modification of business processes. This is done by business users, in suitable *BPM modeling environments*. There, processes are often shown as a kind of flow diagram, e.g. in the wide-spread BPMN notation (OMG 2008).

Today's business environment is increasingly dynamic. It is important to be able to adapt business processes, and to create new processes, quickly. A major bottleneck is 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 communication 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, plus potentially high costs due to increased time-to-market.

The basic idea of our application is to use 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.

This application requires a planning model, i.e., a suitable description of the behavior of the relevant software entities in the IT infrastructure. In most cases, such a description cannot be fully precise (exactly represent all relevant technical aspects of the software artifacts dealt with), and so the plan will not be guaranteed to work at IT-level. Hence planning cannot fully replace the human IT expert.

What the planning can help with is choosing the correct combination of artifacts, 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. Note in particular that the IT landscape may be subject to changes. Indeed a major motivation behind SAP's SAM model is to be able to reflect such changes in a declarative way. Hence the "planning domain" may change, and the flexibility of domain-independent planning is a necessity.

There are two major requirements for practicality: (a) response times should be (almost) instantaneous; and (b) the modeling overhead should be low. As for (a), the planning will be in on-line interaction with a business user, so response time is limited to human patience. While this is important, point (b) is potentially more critical. Appropriately modeling the "domain" – a complex IT infrastructure – can be prohibitively costly. Not less importantly, "instances" must be provided by the business user, on-line. That must be possible in a matter of seconds, and in terms that business users are familiar with. The great news we share herein is that, at SAP, both these modeling issues can be resolved to satisfaction, by leveraging the SAM model.

SAM

As outlined in the introduction, SAM documents the behavior of actions – IT transactions – affecting the status of business objects. "Status" here is represented in terms of a value assignment to a set of finite-domain "status variables". SAM defines for each action a precondition and an effect, stating the required status of the relevant BOs, 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. This is important because execution of actions on BOs *not* satisfying the requirements (e.g., processing a BO whose data fields have inconsistent values) may have all sorts of subtle and harmful side effects, which can constitute a major maintenance problem.

For illustration, Figure 1 gives a SAM-like model for "customer quotes" CQ, our running example.[2] CQ in the figure is used to refer to the customer quote instance.

A few words are in order regarding our "abuse" of SAM for a purpose that was not in the intention of its designers. Just because SAM is *similar* to PDDL, that doesn't mean it makes sense to *use* it like PDDL. Are the processes composed this way useful? Given the lack of tests with real customers, we cannot answer this question conclusively. 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. That said, there is one potentially important shortcoming of SAM, in its current form, that we are aware of. SAM does not reflect any interactions across different kinds of BOs, although such interactions do exist. The interactions are not relevant to SAM's original purpose. But they might be relevant for composing higher quality processes. We get back to this in the conclusion.

Planning Formalization

We discuss the intended meaning of SAM and design a suitable planning formalism. Consider Figure 1. The intended meaning of (disjunctive) preconditions is exactly as in planning. The same goes for non-disjunctive effects. For disjunctive effects, matters 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 exe-

[2] At SAP, SAM models are stored in an XML format. For privacy, the shown object and model are *not* part of SAM as used at SAP. The SAM models are similar.

Action name	precondition	effect
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 Follow-Up for 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 1: Our SAM-like running example, modeling the behavior of "customer quotes" CQ.

cution 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 – up to 100s or 1000s of data fields for a single object – are abstracted in SAM, making the action non-deterministic from its perspective. At the same time, this "non-determinism" is not like throwing a dice: the outcome changes only if the object content was changed in the meantime. That is possible (the processes do not have full control over the objects), but it does not make sense to "wait for the right outcome", and SAM does not model actions allowing to notify someone that the content needs to be changed. We hence take these actions to be non-deterministic but do not allow to repeat them.

SAM explicitly provides an initial value for each status variable. Regarding the goal and the definition of plans, matters are more complicated again. Clearly, the SAM model of Figure 1 does not allow *strong* plans that guarantee success under all circumstances: checking completeness or consistency can always result in a negative outcome that forbids successful processing. This pertains not only to our illustrative example here – in a large experiment, we determined that 75% of the considered instances did not have a strong plan (we get back to this in the experiments section).

To address this, one can define more complicated goals, or a weaker notion of plans. The former is impractical because goals must be specified online by the user, and complex goals require familiarity with the underlying SAP system, which contradicts our value proposition. As for the latter, *strong cyclic plans* (Cimatti et al. 2003) do not make sense because "waiting for the right outcome" does not, c.f. above. The best one can do based on SAM is to highlight the potentially "bad" outcomes, and to make sure to not highlight any outcomes that could actually be solved. We hence settle for a notion of plans that allows non-deterministic actions to have failed outcomes, provided that at least one outcome is solved, and that the failed outcomes are provably unsolvable. For specifying the goal, all that is required is to give the desired attribute values: e.g., CQ.followUp:documentCreated(x) and CQ.archivation:Archived(x) in the case of Figure 1. In our GUI this is done in simple drop-down menus.

The above translates into the following planning formalism. Planning tasks are tuples (X, A, I, G). X is a set of *variables*; each $x \in X$ is associated with a finite domain $dom(x)$. A is a set of *actions*, where each $a \in A$ takes the form (pre_a, E_a) with pre_a being a partial variable assignment, and E_a being a set of partial variable assignments. The members $\mathit{eff}_a \in E_a$ are interpreted as alternative (non-deterministically chosen) *outcomes*. I is a variable assignment representing the *initial state*, and G is a partial variable assignment representing the *goal*.

A *fact* is a pair (x, v) where $x \in X$ and $v \in dom(x)$. We identify (partial) variable assignments with sets of facts in the obvious way. We denote with $dA := \{a \in A \mid |E_a| = 1\}$ and $ndA := \{a \in A \mid |E_a| > 1\}$ the sets of *deterministic* and *non-deterministic* actions, respectively. If $a \in dA$, then by eff_a we denote the single outcome of a.

A *state* s is a variable assignment. 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. An *action tree* is a tree whose nodes are actions and whose edges are labeled with partial variable assignments, so that each action a in the tree has exactly $|E_a|$ outgoing edges, one for (and labeled with) each $\mathit{eff}_a \in E_a$.

Definition 1 *Let (X, A, I, G) be a planning task, let s be a state, let $ndA_{av} \subseteq ndA$, and let T be an action tree. We say that T* solves *(s, ndA_{av}) iff either:*

1. *T is empty and $G \subseteq s$; or*
2. *the root of T is $a \in dA$, $pre_a \subseteq s$, and the sub-tree of T rooted at a's son solves $(s \oplus \mathit{eff}_a, ndA_{av})$; or*
3. *the root of T is $a \in ndA_{av}$, $pre_a \subseteq s$, and for each son of a reached via an edge labeled with $\mathit{eff}_a \in E_a$ we have that either (i) the sub-tree of T rooted at that son solves $(s \oplus \mathit{eff}_a, ndA_{av} \setminus \{a\})$, or (ii) there exists no action tree T' that solves $(s \oplus \mathit{eff}_a, ndA_{av} \setminus \{a\})$, where (i) is the case for at least one of a's sons.*

A plan *is an action tree that solves (I, ndA).*

Items 1 and 2 of this definition should be clear. Item 3 reflects our above discussion of the meaning of SAM. First, non-deterministic effects are directly observed, which is handled by separating the outcomes of a in item 3, rather than including them all into a single belief state. (A desirable side effect is that the object stati are always fully known, i.e., in spite of the non-determinism there are no non-trivial belief states.) Second, we allow solution trees containing unsolvable leaf nodes, as long as below every node there is at least one solved leaf – c.f. the arrangement of options (i) and (ii) in item 3. Third, we allow each non-deterministic action only once on each path through T, as is encoded by the use of the set ndA_{av} in Definition 1.

Figure 2 shows a plan for our running example from Figure 1. For presentation to the user, a post-process (omitted

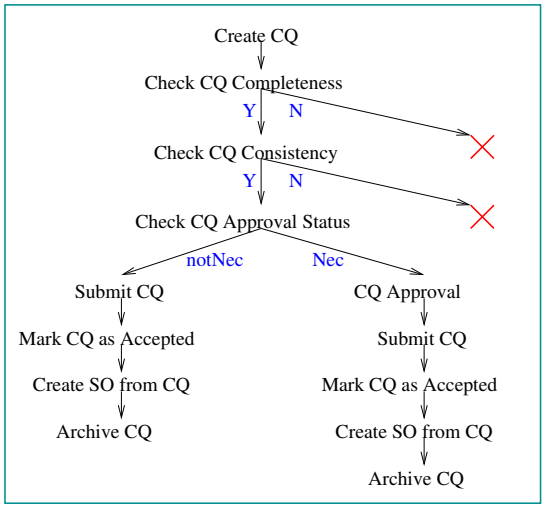

Figure 2: A plan for the running example.

for space reasons) transforms such plans into BPMN workflows, merging redundant sub-trees and using explicit routing nodes for parallel and alternative execution.

Planning Algorithms

We use a variant of AO* tree search. We refer to this variant as *SAM-AO**. Search is forward in an AND-OR tree whose nodes are states (OR nodes) and actions (AND nodes). For deterministic actions, there is a single child so the AND node trivializes. We propagate "node solved" and "node failed" markers. An OR node is failed if either its heuristic value (see below) is infinite, or if all its children are failed. The node is solved if either its heuristic value is 0, or if one of its children is solved. An AND node is failed if all its children are failed. The node is solved if all of its children are either failed or solved, and at least one child is solved. The algorithm terminates when the initial state is solved or failed. In the former case, a solved sub-tree is returned.

We prune duplicates by comparing every new state s backwards with its predecessor states up to the most recent non-trivial AND node (given the SAM semantics, pruning states across non-trivial AND nodes is not possible).

Proposition 1 *Let (X, A, I, G) be a planning task, and let h be a heuristic function. SAM-AO* with duplicate pruning terminates. Provided h returns ∞ only on unsolvable states, SAM-AO* terminates with success iff the task is solvable, and the action tree returned in that case is a plan.*

This follows by definition and the simple observation that duplicate pruning allows only finitely many nodes in a planning task without non-deterministic actions.

We use a simple variant of the FF heuristic function (Hoffmann and Nebel 2001). For the non-deterministic actions, we act as if we could choose the outcome.[3] That is, we compile each $a \in ndA_{av}$ – the non-deterministic actions that are still available at the respective point in the search tree – into the set of deterministic actions $\{(pre_a, \{eff_a\}) \mid eff_a \in E_a\}$. While this is simplistic, as we will see it leads to quite reasonable performance. 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 then returns ∞, enabling SAM-AO* to mark states as "failed".

Experiments

As stated, from a commercial point of view our techniques are in a preliminary stage. Their business value is yet to be determined. We now evaluate them from a scientific point of view. Our experiments are aimed at understanding:

(1) *Is the runtime performance of our planner sufficient for the envisioned application?*

(2) *How interesting is SAM as a planning benchmark?*

All experiments were run on a 1.8 GHZ CPU, with a 10 minute time and 0.5 GB memory cut-off. Our planner is implemented in C as a modification of FF-v2.3. Its input is the aforementioned PDDL version of SAM, where each fact encodes one status variable value, and where non-deterministic actions are represented in a PPDDL-style format.

The initial states in our PDDL encodings are empty – no BOs have been created as yet. Our PDDL allows to create at most a single BO of each type (allowing more would make no sense in terms of the business processes we are aiming to create). Hence a SAM planning instance is identified by its goal: a subset of variable values. This implies in particular that the number of SAM instances is finite. That number, however, is enormous; just for choosing the subset of variables to be constrained we have 2^{2413} options.[4] In what follows, we mostly consider goals all of whose variables belong to a single BO. This is sensible because, as previously stated, SAM currently does not reflect interactions across BOs. There are 1373259536 single-BO goals, which is still astronomic. However, when running FF on all 3482 instances whose goal contains only a single variable value, we found that 989 of those are unreachable. When excluding goals that contain any of these per-se unsolvable values, 9833400 instances remain.

We made an instance generator (also included in the PDDL download) that allows to create instance subsets characterized by the number $|G|$ of variables constrained in the goal. For given $|G|$, the generator enumerates all possible variable tuples, and allows to randomly sample for each of them a given number S of value tuples. The maximum number of variables of any BO is 15. We created all possible instances for $|G| = 1, 2, 3, 13, 14, 15$ where the number of instances is up to around 50000. For all other values of $|G|$, we chose a value for S so that we got around 50000 instances. We omitted instances with a trivial plan, i.e., where the goal is true already in the initial state of the respective BO. The total number of instances created thus is 548987.

[3] This idea is known as "determinization" in probabilistic planning, see e.g. (Yoon, Fern, and Givan 2007).

[4] Anyway, pre-computing a table of plans is not an option because one of the main points about SAM is the ability to easily reflect and handle *changes* in the IT infrastructure.

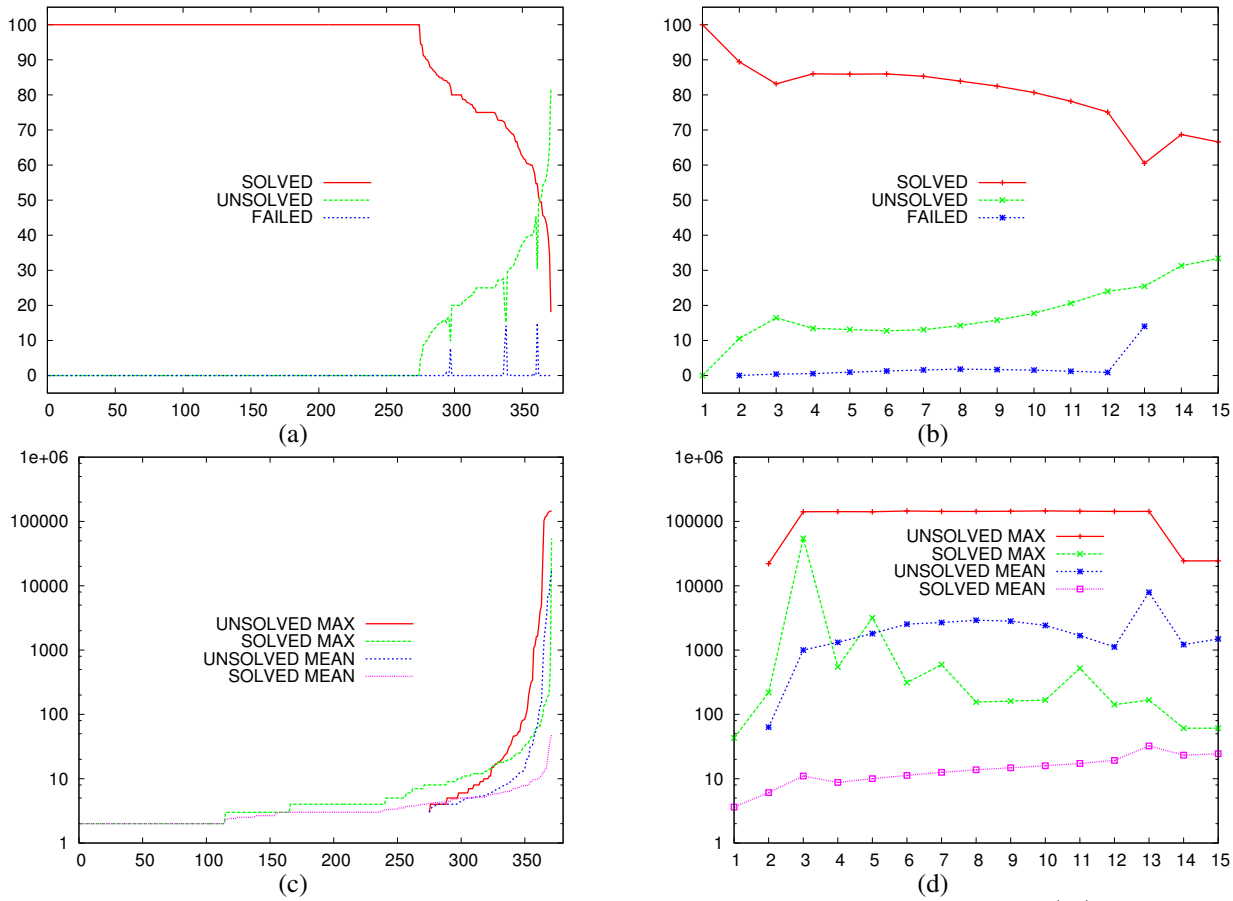

Figure 3: Coverage (a,b) and state evaluations (c,d) data, plotted over individual kinds of BOs (a,c) and $|G|$ (b,d). "SOLVED": plan found. "UNSOLVED": search space (with helpful actions pruning) exhausted. "FAILED": out of time or memory.

In a first experiment, we tried to find strong plans only. To enable proofs of unsolvability, we ran FF without helpful actions pruning, and we included in the input only the subset of actions affecting the relevant BO (else the state space is always much too vast to be exhausted). Overall, only 10% of the instances could be solved; on 15%, FF ran out of time or memory; 75% of them were proved unsolvable.

Strong plans are rare in SAM. We now report detailed results for the more generous semantics as per Definition 1, allowing failed action outcomes as long as they are provably failed. In our baseline, we include *all* actions in the input (not only those of the relevant BO), because (a) helpful actions pruning will detect the irrelevant actions anyway, and (b) in the long term SAM *will* model cross-BO interactions.

We identify two parameters relevant to the performance of FF: the kind of BO considered, and $|G|$. Figure 3 shows how coverage and state evaluations (number of calls to the heuristic function) depend on these parameters. Consider first Figure 3 (a). The x-axis ranges over BOs, i.e., each data point corresponds to one kind of BO.[5] The ordering of BOs is by decreasing percentage of solved instances. For each BO, the y-axis shows the percentage of solved, unsolved, and failed instances within that BO. We see that, in 275 of the 371 kinds of BOs, coverage is perfect. For another 65 BOs, coverage is above 70%. For the remaining 32 BOs, coverage is rather bad. The fraction of unsolved instances peaks at 81.92% and the fraction of failed instances peaks at 14.98%. Considering Figure 3 (b), we see how coverage is affected by $|G|$ (shown on the x-axis). $|G| = 1$ is handled perfectly, followed by a fairly steady decline as $|G|$ grows. An explanation for the discontinuity at $|G| = 3, 4$ could be that for $|G| = 3$ our experiment is exhaustive while for $|G| = 4$ we only sample. The discontinuity at $|G| = 13, 14$ is largely due to BO structure. Few BOs have more than 12 variables. Those with 13 variables happen to be exceptionally challenging, while almost all of the instances for $|G| > 13$ are from a BO that is not quite as challenging.

Consider now Figures 3 (c) and (d) which provide a deeper look into performance on those instances where FF terminated regularly (plan found or helpful actions search space exhausted).[6] In (c), the most striking observation is that, for 350 of the 371 BOs, the maximum number of state evaluations is below 100; for solved instances, this even holds for 364 BOs. In fact, in all but a single BO we always need less than 1000 evaluations to find a plan. The single

[5] We consider only 371 BOs because in the other 33 BOs, all possible goals either contain an unreachable goal value (c.f. above), or are satisfied already in the initial state of the BO.

[6] The ordering of BOs in (c) is by increasing y-value for each curve individually; otherwise the plot would be unreadable.

harder BO contains the peak of 54386 evaluations, finding a plan with 38 actions in it. From the "SOLVED MEAN" curve we see that this behavior is extremely exceptional – the mean number of state evaluations per BO peaks at 47.78. Comparing this to the "UNSOLVED MEAN" curve, we see that a large number of search nodes is, as one would expect, much more typical for unsolved instances.

In Figure 3 (d), we see that the overall behavior of state evaluations over $|G|$ largely mirrors that of coverage, including the discontinuities at $|G| = 3, 4$ and $|G| = 13, 14$ (the "UNSOLVED MAX" curve is flat because larger search spaces lead to failure). The most notable exception is the fairly consistent decline of "SOLVED MAX" for $|G| > 3$. It is unclear to us what the reason for that is.

What is the conclusion regarding the issues (1) (planner performance) and (2) (benchmark challenge) we wish to understand? For issue (1), our results look fairly positive. In particular, consider only the solved instances (plan found). As explained above, the number of state evaluations is largely well-behaved. In addition, the heuristic function is quite fast. The maximum runtime is 27.41 seconds, the second largest runtime is 2.6 seconds. So a practical approach for use in an online business process modeling environment could be to simply apply a small cut off, e.g. 5 seconds or at most a minute. What that leaves us with are, in total, 17.12% unsolved instances and 2.4% failed ones. Are those an important benchmark challenge for future research? Answering this question first of all entails finding out whether (a) we can solve these instances without helpful actions pruning, and (b) if not, whether they are solvable at all.

We ran FF without helpful actions pruning on the unsolved and failed instances, again enabling unsolvability proofs by giving as input only the actions of the relevant BO. All failed instances are still failed in the new FF configuration. Of the previously unsolved instances, 64.95% are failed, 35% are proved unsolvable, and 0.05% are solved (the largest plan contains 140 actions). The number of state evaluations is vastly higher than before, with a mean and max of 10996.72 respectively 289484 for *solved* instances. But the heuristic is extremely fast with a single BO, and so finding a plan takes mean and max runtimes of 0.12 and 2.94 seconds. The influence of $|G|$ and the kind of BO is similar to what we have seen. All in all, changing the planner configuration can achieve some progress on these instances, and it seems that many of them are unsolvable. But certainly they are a challenge for research. We note also that heuristics are required to deal with SAM successfully. For almost 80% of our test instances, blind search (SAM-AO* with a trivial heuristic) exhausts computational resources even if the input contains only the actions of the relevant BO.

Interestingly, FF does not scale gracefully to planning tasks with several BOs. We selected for each of the 404 BOs one solved instance $m(BO)$ with maximum number of state evaluations. We then generated 404 planning tasks where task k combines the goals $m(BO)$ for all BOs up to number k. FF's state evaluations increase sharply with k. The largest instance solved is $k = 101$, with 29078 evaluations. For comparison, the sum of state evaluations when solving the 101 sub-tasks individually is 486. A possible explanation is that, adding more goals for additional BOs, more actions are helpful. The increased number of nodes multiplies over the search depth. This is not relevant at the moment, where SAM does not model cross-BO interactions. But trouble may lie ahead once it does.

Conclusion

As mentioned, our techniques form a research extension to SAP's commercial NetWeaver platform. This concerns in particular the CE Process Composer, NetWeaver's BPM modeling environment. Our prototype is currently in the initial steps towards pilot customer evaluation. It must be said that it is still a long way towards actual commercialization. The potential pilot customer has not made a firm commitment yet, and while our prototype works just fine its positioning inside the SAP software architecture entails some political and technical difficulties.

As was previously pointed out, the current SAM model does not reflect dependencies across BOs. Such dependencies exist, e.g., in the form of transitions that several BOs must take together. A corresponding extension is currently underway 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 the extension for improved planning. Our experiments indicate that additional/modified techniques may be required for satisfactory performance of such planning.

To close, we would like to emphasize that SAM's methodology is not specific to SAP, and that it establishes a direct connection between AI Planning and Software Engineering. Pre/post-condition based models of software behavior are being used in the industry. Intriguingly, this came about with no AI intervention whatsoever. SAM's inventors had no idea that there exists a whole scientific community working since 40 years with precisely this kind of model. We believe that this new genuine connection may turn out fruitful for both fields on a scale much greater than the contribution of the particular technical work presented herein.

References

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence* 147(1-2):35–84.

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

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*.

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

OMG. 2008. Business Process Modeling Notation. http://www.bpmn.org/.

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

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

Yoon, S. W.; Fern, A.; and Givan, R. 2007. FF-Replan: A baseline for probabilistic planning. In *Proc. ICAPS'07*.