Workshop

Planen und Konfigurieren (PuK '06)

Jürgen Sauer
(Editor)

KI2006

14 - 19 JUNE 2006
BREMEN, GERMANY

29th Annual German Conference on Artificial Intelligence
http://www.ai-conference.de/ki06

ISBN 3-88722-670-4
Universität Bremen

Vorwort

Die Fachgruppe Planen/ Scheduling und Konfigurieren/ Entwerfen im FB „Künstliche Intelligenz“ der GI vereint Forscher und Praktiker in den genannten Bereichen und bietet ein gemeinsames Forum zum Austausch von Ergebnissen und Erfahrungen. Die Bereiche Planen, Scheduling, Konfigurieren und Entwerfen verwenden ähnliche Methoden und KI-basierte Techniken, aber häufig findet kein gemeinsamer Erfahrungsaustausch statt. Genau diese Lücke versucht der PUK-Workshop zu schließen, indem neue Fragestellungen, Lösungskonzepte und realisierte Systeme vorgestellt werden können, die zu einem der beteiligten Forschungsgebiete gehören.

Der PUK geht in diesem Jahr in die 20. Runde; eine lange, vielfältige und bewährte Tradition, die insbesondere aufzeigt, wie essentiell dieses Themengebiet in der KI-Forschung verankert ist. Der 20. PuK ist auch der Anlass zu einem Themenheft der KI-Zeitschrift, in dem auf ältere, aktuelle und zukünftig mögliche Entwicklungen eingegangen werden soll. Aus diesem Grund ist der Workshop zweigeteilt. In einem Teil werden aktuelle Ergebnisse aus den Bereichen Planen und Konfigurieren vorgestellt und diskutiert. Im zweiten Teil werden in einer größeren Runde das Konzept und die Inhalte des Themenheftes vorbereitet.

Organisation

PD Dr. Stefan Edelkamp
Tel. ++49-231-755-5809
Fax ++49-231-755-5802
stefan.edelkamp@cs.uni-dortmund.de
ls5-www.cs.uni-dortmund.de/~edelkamp

Universität Dortmund
Lehrstuhl V
Fachbereich Informatik
Otto-Hahn-Strasse 14
44227 Dortmund

Apl. Prof. Dr. Jürgen Sauer
Tel. ++49-441-798-4488
Fax ++49-441-798-4472

juergen.sauer@uni-oldenburg.de
www.wi-ol.de

Universität Oldenburg
Fakultät II, Department
für Informatik, Abt.
Wirtschaftsinformatik
Uhlhornsweg 84
26129 Oldenburg

Inhalt

External Program Model Checking

Stefan Edelkamp, Shahid Jabbar,
Dino Midzic, Daniel Rikowski, and Damian Sulewski

Computer Science Department
University of Dortmund
Otto-Hahn Straße 14

Abstract. To analyze larger models for model checking, external algorithms have shown considerable success in the verification of communication protocols. This paper applies external model checking to software executables. The state in such a verification approach itself is very large, such that main memory hinders the analysis of larger state spaces and calls for I/O efficient exploration algorithms. We propose a general state expanding algorithm based on a search tree skeleton with outsourced states. External collapse traded time for space. Additionally, heuristics accelerate the search process, guide it towards the error and shorten the length of the counterexample. Different caching and exploration strategies are evaluated. We found a counterexample in a C++-program for the Dining Philosophers' problem with 300 philosophers in a successful exploration lasting for over 100 hours while consuming 0.5 gigabytes RAM and 19 gigabytes hard disk.

1 Introduction

Model checking is a formal verification method for state based systems, which has been successfully applied in various fields, including process engineering, hardware design and protocol verification. Recent applications of model checking technology deal with the verification of software implementations (rather than checking a formal specification). The advantage of this approach is manifold when compared to the modus operandi in the established software development cycle. For safety-critical software, the designers would normally write the system specification in a formal language like Z [35] and (manually) prove formal properties over that specification. When the development process gradually shifts towards the implementation phase, the specification must be completely rewritten in the actual programming language (usually C++). On the one hand, this implies an additional overhead, as the same program logic is merely reformulated in a different language. On the other hand, the re-writing is prone to errors and may falsify properties that hold in the formal specification.

Modern software model checkers rely on the extension or implementation of architectures capable of interpreting machine code. These architectures include virtual machines [38] and debuggers [26]. Such unabstracted software model checking does not suffer from any of the problems of the classical approach.

Neither the user is burdened with the task of building an error-prone model of the program, nor there is a need to develop a parser that translates (subsets of) the targeted programming language into the language of the model checker. Instead, any established compiler for the respective programming language can be used.

Given that the underlying virtual machine works correctly, we can assume that the model checker is capable of detecting all errors and that it will only report real errors. Also, the model checker can provide the user with an error trail on the source level. Not only does this facilitate to detect the error in the actual program, the user is also not required to be familiar with the specialized modeling languages, such as Promela. As its main disadvantage, unabstracted software model checking may expose a large state description, since a state must memorize the contents of the stack and all allocated memory regions. As a consequence, the generated states may quickly exceed the computer's available memory. Moreover, larger state descriptions slow down the exploration. Therefore, the most important topic in the development of an unabstracted software model checker is to devise techniques that can handle the potentially large states.

The list of techniques for state space compression is long (cf. [3]): *partial-order reduction* prunes the state space based on stuttering equivalences tracking commuting state transitions, *symmetry detection* exploits problem regularities on the state vector, *binary state encoding* allows to represent larger sets of states, *abstraction methods* analyze smaller state spaces inferred by suitable approximations, and *bit-state hashing* and *hash compaction* compress the state vector down to a few bits, while failing to disambiguate some of the synonyms. One important aspect (not mentioned in [3]) are *search heuristics* that guide the search process to the location of the error. Such directed model checking approaches have shown significant advances in the verification of communication protocols [7], selective mu-calculus [32], Java programs [10] and computer hardware [2].

But even with refined exploration techniques, model checking is bounded by the main memory resources. Several memory-limited model checking algorithms have been developed, e.g. [8, 12, 17] but still the core limitation hurdles the model checking of large programs. Infact, the use of virtual memory as a remedy to this problem can instead slow down the performance significantly. Since a general purpose vitual memory scheme, as is offered by many operating systems, does not know anything about the future memory accesses in a model checking algorithm, excessive page faults are inevitable.

External memory algorithms [31] use secondary storage devices, such as hard disk, to solve *large problems* - the problems that have a space requirement larger than the main memory available. They differ from the vitual memory scheme and are more informed about the future access to the data. Milestones for external model checking are [36, 21]. External search algorithms have also shown remarkable performance in the large-scale analysis of games [20], where external breadth-first search has fully explored a single-agent challenge using 1.4 terrabytes hard disk space. Recently, this form of large-scale exploration has been ported to enhance (directed) model checking. In [15] an external and

guided derivate of the explicit-state model checker SPIN has been introduced. As external model checking refers to the independent exploration of set of states the algorithms have been successfully parallelized [16], showing an almost linear speed-up. Based on the work of [33] this large-scale model checking algorithm has been extended to the validation of LTL properties [5].

For the purpose of this paper, we consider the external exploration in model checking unabstracted programs on the object-code level. As systems states for program executables are less accessible and highly dynamic, the algorithmic approach deviates considerably from previous work. The paper is structured as follows. First we introduce to program model checking on the object-code level and to the system states of a program. Next we turn to large-scale model checking and its limitation for the validation of programs. Then we propose the framework for external program model checking that we have developed. It consists of a compromise between main and external memory usage based on an annotated search tree skeleton. For the presentation of the approach we consider the distributed storage of states. Afterwards, we turn to the implementation of an external C++ model checker and present obtained time and space trade-offs in the experiments.

2 Object-Code Program Model Checking

The state of a computer program consists of static components, such as the global variables, as well as dynamic components, like the program stack and the pool of dynamically allocated memory.

Figure 1 shows the components that form the state of a concurrent program for object-code model checking. As model checking is particularly interesting for the verification of concurrent programs, the state description include all relevant information about an arbitrary number of running processes (threads). Threads may claim and release exclusive access to a resource by *locking* and *unlocking* it. Resources are usually single memory cells (variables) or whole blocks of memory. The state description include information about the location and size of dynamically allocated memory blocks, as well as the allocating thread. The memory is divided in three layers: The outer-most layer is the physical memory which is visible only to the model checker. The subset *VM-memory* is also visible to the virtual machine and contains information about the main thread, i.e., the thread containing the main method of the program to check. The program memory forms a subset of the VM-memory and contains regions that are dynamically allocated by the program. Before the next step of a thread can be executed, the content of machine registers and stacks must refer to the state immediately after the last execution of the same thread, or if it is new, directly after initialization. The *memory-pool* is used to manage dynamically allocated memory. The *lock-pool* stores information about locked resources.

An increased difficulty of program model checking the object code lies in the handling of untyped memory as opposed by Java, where all allocated memory can

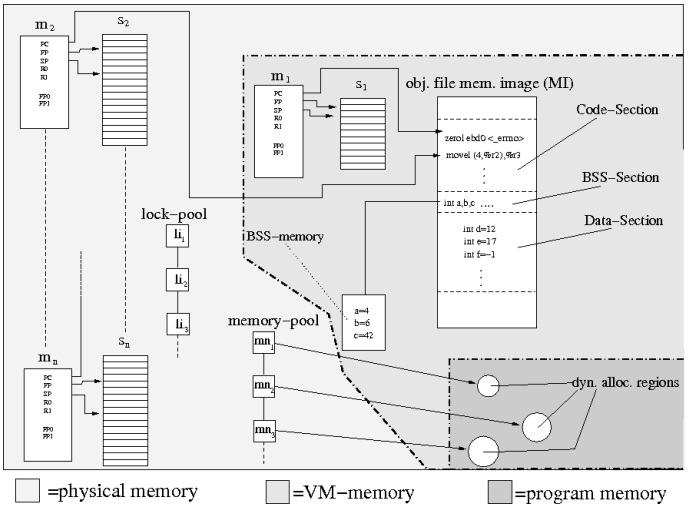


Fig. 1. System state of a program.

be attributed to instances of certain object types. Also, a standardized interface for multi-threading frequently does not exist.

3 External Model Checking

External model checking algorithms explicitly manage the memory hierarchy and can lead to substantial speedups compared to caching and pre-fetching heuristics of the underlying operating system, since they are more informed to predict and adjust future memory access.

External guided model checking refers to early results of externalizing the AI search algorithm A* [28] for optimal plan-finding in single-agent challenges. A* is a single-source shortest-path implicit graph algorithm with included estimate costs. When ran on virtual memory, A* becomes I/O bound due to excessive page faults. External A* [6] operates on a matrix of files (buckets) that is addressed by the generating path length and the heuristic estimate. Each access to a file is sequential and buffered. Duplicate elimination is delayed [19]. Based on External A*, in [15] an experimental explicit-state model checker for safety properties has been implemented on top of SPIN [13], parallelized [16] and extended to LTL properties [5]. As designed for error detection, all external algorithms operate

on-the-fly. The search for safety properties is shown to be I/O optimal[1], while for LTL properties the exact I/O complexity is still open[2].

Different to the approach presented below, the externality applied to traditional model checking is *strict*, so that main memory resources remain constant during the verification run. The set of states in the current bucket and the set of generated states are all contained on disk and streamed during bucket expansion. This allows to pause and resume the exploration at any given state. However, storing all states in program model checking individually on disk, challenges existing hardware resources.

Exploiting redundancies using a symbolic representation with decision diagrams has been successfully applied and externalized in the context of AI planning [4]. Each bucket in External A* is represented by and stored as a BDD. On the other hand, the highly dynamic structure of states in program model checking does not suggest the usuage of BDDs. Therefore, we have relaxed the requirement of a constant main memory.

4 The Algorithm

The general state-expanding algorithm we propose is based on the idea of *mini-states*. Essentially, a mini-state is a pointer to a full system state residing on the secondary memory. A mini-state consists of the hash value of its corresponding state, a pointer to the state - in the form of a file pointer, and its predecessor information to reconstruct the solution path. Additional information includes its depth and its heuristic estimate to the target state. All in all, a mini-state has a constant size in contrast to a state that can change its size due to dynamic memory allocation.

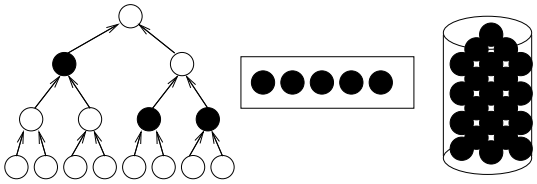
For the sake of brevity, we restrict to properties that can be validated by looking at a state. Recall that in general state-expanding algorithms, full states have to be accessed either to get explored or to be referred to for duplicate detection.

4.1 Externalization

In a search algorithm a full state is only needed in two scenarios: expansion and duplicate detection. Exploiting the idea of mini-states, we propose to perform the search on a tree skeleton defined on the mini-states, while actual states reside on the secondary memory. A request for expansion now reads the state from the disk based on the file pointer stored in the corresponding mini-state. Once read,

[1] $O(sort(|E|)/p + scan(|V|))$ I/Os for an undirected state space graphs with consistent estimate, where $|V|$ and $|E|$ are the number of traversed states and transitions, p is the number of processors and where $scan(n)$ and $sort(n)$ are the I/O complexities to scan and sort n objects.

[2] The algorithms suggested amounts to $O(sort(|E|)/p + t \cdot scan(|V|))$ I/Os for a general state space graphs with consistent estimate, where t is the depth of the solution.



```
ExpandExtern(mini-state s)
x ← Read(s)
for x' ∈ expand(x)
    c ← hash(x')
    dup ← false
    for s' ∈ H[c]
        if (Read(s')=x')
            dup ← true; break
    if (dup = false)
        {allocate mini-state s''}
        s''.pred ← s
        s''.o ← x.code(x')
        Write(x')
```

Fig. 2. General state expansion with external storage: Externalization of state in a search tree using a cache and an external state pool (left). Colored nodes state in the cache, hollow nodes illustrate mini-states without any representation in main memory. Pseudo-Code (right)

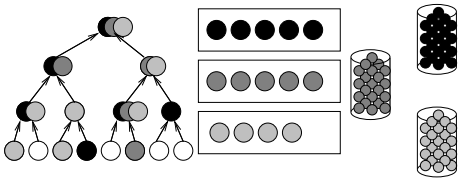
the state is expanded and its children are again saved in the form of mini-states in the internal memory and as full states on the secondary memory.

Duplicate detection is done based on a hash-table storing only the mini-states. The hash value of a mini-state is the hash value of the full state and is calculated when a state is generated.

In the worst case, we perform one I/O operation for every access to a state. To lessen the average number of I/O operations, we associate an internal cache data structure that allows to retrieve and store in main memory, a small set of states from secondary memory. Though this cache seems very much like virtual memory as offered by almost all operating systems, it can be configured to follow the best replacement strategy suited to the search algorithm. The cache principle is illustrated in Figure 2 (left).

The advantage of external state representation is that we can restore each state that we want from disk, even if it is not in main memory. To do so, we let each mini-state also refers to a file pointer location on disk. In case the efforts for state reconstruction become too large we can change to external states. Once read, the external state becomes full states in the search tree.

The pseudo-code for external search is based on completing mini-states from disk is given in Figure 2 (right). For a mini-state s, $s.o$ denotes the transition (e.g. the sequence of machine instructions), which transforms the predecessor $s.pred$ into s. Similarly, for a full state x, $x.code(x')$ denotes the operation which transforms x to it's successor state x'. Note, that transition have a constant-sized representation, which is usually the program counter of a thread running in x. The notion $expand(x)$ refers to the expansion of a full state x and generating a



```
ExpandCollapse(mini-state s)
(x_1, ..., x_k) ← Read(s)
for (x'_1, ..., x'_k) ∈ expand(x_1, ..., x_k)
    (c_1, ..., c_k) ← hash(x'_1, ..., x'_k)
    dup ← false
    for s' ∈ H[c_1, ..., c_k]
        if (Read(s')=(x'_1, ..., x'_k))
            dup ← true; break
    if (dup = false)
        {allocate mini-state s''}
        s''.pred ← s
        s''.o ← x.code(x')
        H[c_1, ..., c_k] ← H[c_1, ..., c_k] ∪ s''
        Write(x'_1, ..., x'_k)
```

Fig. 3. General state expansion with external storage and collapse compression. Externalization of state in a search tree using caches and distributed state memorization. Colored nodes illustrate partial state information, hollow nodes illustrate mini-states without any information (left). Pseudo-code (right).

list of successors. The hash table H contains the mini-state representatives of all previously generated states.

4.2 External Collapse Compression

Either in main or on secondary memory, storing the entire state information individually is inefficient. A state consists of several parts: stack, memory pool, global variables, etc. A typical transition or a program statement changes only some of them, resulting in several states having common parts. Expoiting these redundancies can lead to a compression scheme where common parts are stored only once.

Collapse compression [14, 22] is a sophisticated approach to store states in an efficient way. Collapsing is based on the observation that although the number of distinct states can become very large, the number of distinct parts of the system are usually smaller. These parts of the state can be shared across all the visited states that are stored, instead of storing the complete encoding of state every time a new state is visited. So, different components are stored in separate hash tables. Each entry in one of the tables is given a unique number. A whole system state is identified by a vector of numbers that refer to corresponding components in the hash tables. This greatly reduces the storage needs for storing the set of already explored states. The principle of collapse compression is illustrated in Figure 3 (left).

Similar to the exposition in [30] we collapse the global store, the memory pool objects, and the threads. These are combined to individual hash addresses for a final encoding. As a hash function for the individual memory regions and stack frames we take $h(v_1, \ldots, v_n) = \sum_{i=1}^{n} v_i \cdot |\Sigma|^i$ modulo a prime q, where Σ is

set of computer characters (usually encoded in form of a byte). The final value $hash(x_1, \ldots, x_k)$ as denoted in the pseudo-code (Figure 3 (right)) is in fact a hash function on $h_1(x_1), \ldots, h_k(x_k)$.

One advantage of this construction is that the hash function can be computed incrementally, based on the change in the state [25]. For example considering the stack frames, components are added or removed only at the beginning and the end, so that the resulting hash address can be computed incrementally in constant time:

$$h(v_1, \ldots, v_n, v_{n+1}) \equiv \sum_{i=1}^{n+1} v_i \cdot |\Sigma|^i \equiv h(v_1, \ldots, v_n) + v_{n+1} \cdot |\Sigma|^{n+1} \, mod \, q$$

$$h(v_1, \ldots, v_{n-1}) \equiv \sum_{i=1}^{n-1} v_i \cdot |\Sigma|^i \equiv h(v_1, \ldots, v_n) - v_n \cdot |\Sigma|^n \, mod \, q$$

For the memory pool, we use a balanced AVL [1] tree t with m inner nodes We define a recursive hash function h' for node N in t as follows: If N is a leaf then $h'(N) = 0$. Otherwise, we set $h'(N)$ to $h'(N_l) + h(v(N)) \cdot |\Sigma|^{|N_l|} + h'(N_r) \cdot |\Sigma|^{|N_l|+|v(N)|} \, mod \, q$. Here $|N|$ denotes the accumulated length of the vectors in a subtree with root N, and $v(N)$ stands for the subvector associated to N, while N_l, N_r denote the left and right subtrees of N respectively. If R is the root of t, then $h'(R)$ gives the same hash value as h applied to the concatenation of all subvectors in order. For incrementally hash the memory pool, we require logarithmic time.

External distributed state storage, external collapse compression for short, refers to the setting that different entities of state vectors are stored individually on disk. Stacks, memory regions, and individual storage units are maintained in different files, all buffered in cache data structures. This may increase the number of I/Os in case one state item is not contained in the cache but greatly reduces external resources. The pseudo code for external collapse compression is given in Figure 3 (right).

4.3 State Caching Strategies

Memory-limited search has a long tradition in Model Checking. For example, *state-space caching* [12] stores the states in DFS creating a cache of visited states until all main memory resources are exhausted. Many multiple redundant explorations are due to different interleavings of partial orderings of transitions. They have been tackled using different partial ordering methods such as *ample* [3], *persistent* or *sleep* sets [8].

Different to earlier approaches to external exploration we require caching strategies to retrieve/flush states into/from main memory. There are different caching strategies. Some known ones are *Last-In-First-Out (LIFO)*, *First-In--First-Out (FIFO)*, *Least-Recently-Used (LRU)*, *Least-Frequently-Used (LFU)*, *Flush-When-Full (FWF)*, etc. [37]. Different to the operating system we have

the advantage to adapt the caching strategy to the exploration algorithm. Another simple option is a *two-dimensional cache* as an array of fixed length l. The elements of the array are buckets and each bucket can contain k elements. The size of the cache is $m = kl$. Calculating the address for the entries is achieved by a usual hash function. The strategy to delete a node when exceeding the depth k we might consider is *first-in first-out*, such that the first inserted node in turn is replaced. A read failure is given, if the value is not represented within the stack anymore. It is easy to derive that the expected life time for an element is m. Therefore, let $p = 1/l$ be the probability, that a chosen element fits into a bucket and $q = (l-1)/l$ be its dual probability. Then the random variable X denoting the number of assignments up to the k-th success is negative binomial distributed, such that $P(X = r) = \binom{r-1}{k-1} p^k q^{r-k}$ for $r \in \{k, k+1, \ldots\}$. Therefore, $E(X) = k/p = kl$.

As the number of states to be expanded is also limited for AI search, different strategies have been proposed. IDA* [18] invokes a series of cost-bounded DFS searches, but shows exponential behavior in many state-space graphs. MREC [34] is an algorithm, that exploits the entire memory for exploration and reassigns space as needed, propagating cost values upwards. Node caching strategies like SNC [27] introduce randomness to the storage of nodes. The overall probability that a state is cached is $1-(1-p)^t$, where $p \in [0,1]$ is a fixed parameter and t is the number of times that a state is revisited. For $p = 0$ the algorithm correlates with IDA* and for $p = 1$ it matches MREC.

We implemented separate caches, one for the data section, one for the binary section, one for the stack contents and one for the rest of the system state. All of the components can be individually flushed to and read from disk. For the data and binary section we incrementally check at construction time, whether a change has occurred, for the stack we check for redundancies at insertion time. In all three cases, the cache data structure is realized by using an AVL tree sorted by the individual hash addresses. The forth cache is a simple FIFO structure. If a state is generated, we first check by a hash comparison if it is new. If a hash conflict is determined the state is retrieved from the cache (or, if not present, from the hard disk). If the list exceeds a certain predefined value, all elements that are not yet residing on disk are flushed. The last state element is deleted making space for the next one to come.

For external storage we have decided to store system states in blocks of the same size. Each block correspond an own file. The number of elements to be stored should be adjusted that the file size is close to but does not exceed 2 GB, a common file size limit on current computers. If one block gets exhausted, a new file with a rising index file is created. Using number to index the files allows to keep the information overhead in the mini-state at the acceptable level of one integer value.

Philos.	Explored States	Path Length	RAM	Harddisk	Time
Original Implementation					
5	78,173	19	457MB	-	369s
Collapse Compression					
5	78,173	19	233MB	-	346s
Externalization					
6	642,982	22	195MB	568MB	90m
7	546,995	25	233MB	8,6GB	50h

Table 1. Exploration results for breadth-first search.

5 Experiments

We implemented external exploration on top of our tool StEAM [24]. StEAM is an experimental model checker for concurrent C++ programs[3].

We draw experiments on a Linux System with 1.7 GHz CPU, 512 MB RAM and a IDE hard disk that was limited to 20 GB.

Our running case study are the Dining Philosophers problem that illustrate the scalability of the approach. With p we denote the number of philosophers. We compare different search methods, namely breadth-first, depth-first and best-first search and different memory saving strategies: incremental state storage as in original StEAM, collapse compression, and externalization. For secondary search all information of a state except the stacks are externalized. The cache sizes are fixed to 1,024 states.

First we consider breadth-first search. Our results are shown in 1. For the original implementation and internal collapse compression we show the largest instance that we could solve with respect to the available memory bound. The result in external search show that with internal collapse compression alone we cannot solve problems with more than 6 philosophers. For external search at $p = 8$ we arrived at the limit of our hard disk capacity, while the internal memory consumption suggest that we can scale higher. We observe that the step-optimal counterexample lies at depth $4 + 3p$, where p is the number of philosophers.

Depth-first search results are depicted in Table 2. The effect is that the exploration can solve by far larger instances. The unfortunate side-effect is that the counterexamples becomes quite lengthy. With the original implementation and internal collapse compression we approached to the limit of main memory with 20 and 25 philosophers, respectively. As we see, externalization allows to scale the problem size to at least twice as many philosophers. As the counterexample for $p = 50$ exceeds the length of ten thousands steps, we expect the burden for

[3] The program model checker StEAM including the proposed externalization options is available at `http://sourceforge.net/projects/bugfinder` in Linux packages (.deb and .rpm). The web page also provides information on the tool from an developer and application programmer point of view. Compared to the implementation as provided by [24] the implmentation has been cleaned and documented with *doxygen*. For detecting memory leaks *valgrind* proved to been the most valuable resource.

Philo.	Explored States	Path Len.	RAM	Harddisk	Time
Original Implementation					
20	48,998	3,898	426MB	-	587s
Collapse Compression					
25	76,954	5,123	405MB	-	17m
Externalization					
50	304,929	10,938	343MB	3GB	4h

Table 2. Exploration results for depth-first search.

Philos.	Explored States	Path Len.	RAM	Harddisk	Time
Original Implementation					
50	9,465	353	239MB	-	10m
Collapse Compression					
50	9,465	353	211MB	-	10m
100	36,430	703	566MB	-	160m
Externalization					
150	80,895	1,053	256MB	2.3GB	14h
300	319,290	2,103	545MB	19GB	104h

Table 3. Exploration results for greedy best-first search using most-blocked heuristic.

the application programmer to trace the error to be too large and did not try larger instances.

The results for greedy best-first search are shown in Table 3. As a heuristic we chose the most-blocked heuristic that simply counts the number of process that cannot execute a transition due to existing locks.

As expected, directed search accelerates the exploration enormously while keeping the counterexample length at an acceptable level. With internal collapse compression we slightly exceeded the limit of main memory with 100 philosophers, at which the system started to swap.

Externalization allows to scale the problem size to 300 philosophers. We also measured the number of hard disk accesses. This exploration last for more than 4 days without exceeding the memory available. This indicates that our implementation is almost free of memory leaks. The number of write accesses is 319,291 and the number of read accesses is 7,280. Moreover, the number of items in the stack cache equals 1,203.

In all the above mentioned experiments we used a single file to store all the states. This policy turned out to work only for the domains where there was a good distribution of the hash values and the range of the hash function was large enough to result in few collisions. Such is the case with Dinning Philosophers.

While experimenting on a c++ program for solving 8-puzzle instances, we observed that the single file based storage scheme resulted in an increase in time due to large number of I/Os. The reason turned out to be small range of the hash function that results in large number of collisions. Dividing the storage file

Instance	Explored States	Path Len.	RAM	Harddisk	Time	Hashcodes	Collisions
Original Implementation							
207165384	248,243	141	552MB	-	3,933s	2,906	21,538,404
267105384	437,400	150	861MB	-	6,933s	2,938	63,637,143
267150384	607,331	159	1125	-	8,961s	2,997	119,295,029
Collapse Compression							
207165384	248,243	141	294MB	-	3,798s	2,906	21,538,404
267105384	437,400	150	407MB	-	6,776s	2,938	63,637,143
267150384	607,331	159	509MB	-	9,490s	2,997	119,295,029
267154380	840,493	168	648MB	-	13,310s	2,985	222,463,964
267154308	1,405,027	177	974MB	-	21,713s	2,982	598,825,442
Sinlge File Externalization							
207165384	248,243	141	173MB	64MB	4,503s	2,906	21,538,404
267105384	437,400	150	193MB	112MB	9,324s	2,938	63,637,143
267150384	607,331	159	212MB	156MB	14,574s	2,997	119,295,029
267154380	840,493	168	237MB	215MB	23,296s	2,985	222,463,964
Hash File Externalization							
207165384	248,243	141	173MB	71MB	4,422s	2,906	21,538,404
267105384	437,400	150	197MB	123MB	8,553s	2,938	63,637,143
267150384	607,331	159	216MB	167MB	12,955s	2,997	119,295,029
267154380	840,493	168	237MB	228MB	18,479s	2,985	222,463,964

Table 4. Comparison of state storage strategies on 8-Puzzle instances.

according to the hash values solved our problem. For each hashcode that was generated we assigned a new file, where states having that particular hashcode are saved. While resolving a hash conflict we suggest to read the file block-wise in a small internal memory cache. This resulted in an increase in time performance as compare to the single-file based storage where the algorithm has to perform several jumps in the file to resolve a conflict. In Table 4 we show a comparison of different state storage strategies while solving 8-puzzle instances. We compare the performance of this new storage scheme with the single-file based implementation. A gain in time is clearly observable. This new scheme though has to be dealt with care. For problems where the number of unique hashcodes is very large, the I/O performance of the algorithm can infact decrease because of accessing very small files. One should also take care that the number of files could grow more than the allowable limit of the operating system.

6 Conclusion

Tailoring a model checking engine to an existing virtual machine for model checking c++ is a challenging task, that was thought to be infeasible, e.g. by the creators of JPF [38].

With this work we have provide the first implementation of an external program model checker, which does not rely on abstract models. The main difference

to previous attempts in external model checking is that a skeleton of the search tree resides in main memory.

The savings obtained by external exploration are considerable and likely another important step towards practical applicability. Externalization positively combines with directed exploration for accelerated error detection.

Despite considerably long CPU times induced by hard disk access for external exploration in larger problem instances, the exploration efficiencies are still remarkable. To the authors knowledge (and even when equipped with a considerably small hardware resources of about 500 MB RAM and 20 GB hard disk), we could present the largest explorations in program model checking that have been achieved so far. Other program model checkers like VeriSoft are reported to solve the philosophers problems (including state-less search and partial order reduction with persistent and sleep sets) with at most 10 philosophers [8]. Moreover, even directed model checkers like HSF-SPIN that rely on an abstract model and that do not take the burden of exploring the object-code [23] show considerable work in solving larger philosophers problems as even the much simpler structured state vector appears considerably large. For externalizing HSF-SPIN [15], 2.29 GB were reported for $p = 100$ and 10.4 GB for $p = 150$ also using the most-blocked/active-process heuristic.

The approach for analyzing c++ based on model checking the object code is still experimental. While the tool and the virtual machine both compile on gcc 4.0, the compiler for the virtual machine still relies on gcc 2.95. For many input file this is not a limitation as recent developments of the compiler are mostly more restrictive on their inputs. However as the libraries are much different, we plan to extend the virtual machine to work on cross-compiled code such that our model checker can work on different processor models too.

We currently work on data and predicate abstraction [9] to be used in an abstract-refinement loop [11] or for constructing abstraction databases [29]. As abstractions convert a deterministic program into a non-deterministic one, we aim to use our external model checker to explore the abstract models to guide the search in the concrete program. Inspired by XSPIN, a GUI for SPIN model checker, we have also started to develop a GUI for the model checking tool in form of a plugin for Eclipse. So far the frontend can be used to select the parameters of and call the model checker StEAM on the developed sources in the know c++ environment CDT. Moreover, we are able to display the counterexample trail and exploration statistics of the model checker in an XML browser.

References

1. G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962.
2. R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *DAC*, pages 29–34, 2000.
3. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.
4. S. Edelkamp. External symbolic heuristic search with pattern databases. In *ICAPS*, pages 51–60, 2005.

5. S. Edelkamp and S. Jabbar. Large-scale directed LTL model checking. In *SPIN*, 2006.
6. S. Edelkamp, S. Jabbar, and S. Schrödl. External A*. In *KI*, pages 226–240, 2004.
7. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *STTT*, 5(2-3):247–267, 2004.
8. P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.
9. S. Graf and H. Saidi. Construction of abstract state graphs of infinite systems with PVS. In *CAV*, pages 72–83, 1997.
10. A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *ISSTA*, pages 12–21, 2002.
11. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *SPIN*, pages 235–239, 2003.
12. G. Holzmann. Tracing protocols. *AT& T Technical Journal*, 64(12):2413–2434, 1985.
13. G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
14. G. J. Holzmann. State compression in spin. In *Third Spin Workshop*, Twente University, The Netherlands, 1997.
15. S. Jabbar and S. Edelkamp. I/O efficient directed model checking. In *VMCAI*, pages 313–329, 2005.
16. S. Jabbar and S. Edelkamp. Parallel external directed model checking with linear I/O. In *VMCAI*, 2006.
17. C. Jard and Th. Jeron. Bounded-memory algorithms for verification on-the-fly. In *CAV*, 1991.
18. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
19. R. E. Korf. Breadth-first frontier search with delayed duplicate detection. In *IJCAI-workshop: Model Checking and Artificial Intelligence (MoChArt)*, pages 87–92, 2003.
20. R. E. Korf and P. Schultze. Large-scale parallel breadth-first search. In *AAAI*, pages 1380–1386, 2005.
21. L. M. Kristensen and T. Mailund. Path finding with the sweep-line method using external storage. In *ICFEM*, pages 319–337, 2003.
22. F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *SPIN*, pages 80–102.
23. A. Lluch-Lafuente. *Heuristic Search in the verification of Communication Protocols*. PhD thesis, Computer Science Institute, Freiburg University, 2003.
24. T. Mehler. *Challenges and Applications of Assembly-Level Software Model Checking*. PhD thesis, University of Dortmund, 2006.
25. T. Mehler and S. Edelkamp. Dynamic incremental hashing in program model checking. *ENTCS*, 2005.
26. E. Mercer and M. Jones. Model checking machine code with the GNU debugger. In *SPIN*, pages 251–265, 2005.
27. T. Minura and T. Ishida. Stochastical node caching for memory-bounded search. In *National Conference on Artificial Intelligence (AAAI)*, pages 450–456, 1998.
28. J. Pearl. *Heuristics*. Addison-Wesley, 1985.
29. K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *TACAS*, pages 497–511, 2004.
30. Robby, M. B. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic software. *ENTCS*, 89(3), 2003.

31. P. Sanders, U. Meyer, and J. F. Sibeyn. *Algorithms for Memory Hierarchies*. Springer, 2002.
32. A. Santone. Heuristic search + local model checking in selective mu-calculus. *IEEE Transactions on Software Engineering*, 29(6):510–523, 2003.
33. V. Schuppan and A. Biere. From distribution memory cycle detection to parallel model checking. *STTT*, 5(2–3):185–204, 2004.
34. A. K. Sen and A. Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *IJCAI*, pages 297–302, 1989.
35. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 1992.
36. U. Stern and D. Dill. Using magnetic disk instead of main memory in the murphi verifier. In *CAV*, pages 172–183, 1998.
37. A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.
38. W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *ICSE*, pages 3–12, 2000.

7 Appendix: c++-Sources Dining Philosophers

Philosopher.h

```
#ifndef PHILOSOPHER
#define PHILOSOPHER

#include "IVMThread.h"

class Philosopher : public IVMThread {

 private:
  static int id_counter;
  short * leftfork;
  short * rightfork;

 public:
  Philosopher();
  Philosopher(short * leftfork, short * rightfork);
  virtual void start();
  virtual void run();
  virtual void die();
};
#endif
```

Philosopher.cc

```
#include "icvm_verify.h"
#include "IVMThread.h"
#include "Philosopher.h"

class IVMThread;
extern int g[10];

Philosopher::Philosopher(short * lf, short * rf) :
                                  IVMThread::IVMThread() {
  leftfork=lf; rightfork=rf;
}

Philosopher::Philosopher() {}

void Philosopher::start() {
  run();
}

void Philosopher::run() {
  int i;
```

```
  while(1) {
     VLOCK(leftfork);
     VLOCK(rightfork);
     VUNLOCK(rightfork);
     VUNLOCK(leftfork);
  }
}

void Philosopher::die() {}
int Philosopher::id_counter;
```

philosophers.c

```
#include <stdlib.h>
#include <assert.h>
#include "Philosopher.h"

class Philosopher;

Philosopher ** p;
short forks[255];

void initThreads (int n) {
  p=(Philosopher **) malloc(n*sizeof(Philosopher *));
  for(int i=0;i<n;i++) {
    p[i]=new Philosopher(&forks[i], &forks[(i+1) % n]);
    p[i]->start();
  }
}

int main(int argc, char ** argv)
{
  int n;
  BEGINATOMIC;
  if(argc<2) {
    fprintf(stdout, "---=====Missing Parameter!=====---\n");
    exit(0);
  }
  n=atoi(argv[1]);
  initThreads(n);
  ENDATOMIC;
  return 1;
}
```

Requirements-driven Software Development System (ReDSeeDS) – A Project Outline

Thorsten Krebs[1] and Lothar Hotz[2]

[1] HITeC c/o Fachbereich Informatik, Universität Hamburg
Hamburg, Germany, 22527
krebs@informatik.uni-hamburg.de
[2] HITeC c/o Fachbereich Informatik, Universität Hamburg
Hamburg, Germany, 22527
hotz@informatik.uni-hamburg.de

Abstract. This paper presents an outline of the upcoming project ReDSeeDS. This project aims at creating a sophisticated framework for reuse not only of previously created software systems as a whole, but also considering their parts, including the problem (requirements) as well as the solution (architecture, design and code) A standard way of formalizing knowledge about complete software cases will be developed. Previous cases can be retrieved based on a similarity measure and reused for similar new software development. Requirements engineering, reusable asset libraries, model transformation, knowledge-based querying, knowledge representation and case-based reasoning will be coherently combined to enable the requirements-driven software development.

1 Introduction

In this paper we introduce the EU-funded project *ReDSeeDS* (*Requirements-driven Software Development System*). This project combines some well-known approaches from software engineering and artificial intelligence to achieve a framework for reuse-based software development. Previous development of software systems is stored in *cases* and can be retrieved based on a requirements specification. The previous system can then be adapted to case-specific requirements with help of model-driven development.

The ReDSeeDS project aims at creating a sophisticated framework for reuse not only of previous software systems but also considering requirements, architecture, design and code. All this information will be stored containing both meta data and the development solution itself – together forming a software *case*. This means that software cases can be identified and retrieved based on all information that is contained within this meta data, not only based on the previous solution.

The remainder of this paper is organized as follows: in Section 2 we outline the problems current reuse-based software development struggles with in practice. After that we detail how these problems can be tackled in Section 3 and describe the underlying formalisms. Finally, we give a summary and conclude this paper in Section 4.

2 The Problem

It is a common trend that software systems become more and more complex. This is due to increasing customer requirements and advancing possibilities in using technical and electronic system. Of course there is an interaction between these two issues: customers demand more functionality as technical possibilities advance, and vice versa technical advancements are driven by customer requirements. Companies invest huge amounts in development of new software systems in order to stay competitive.

Reuse has long been identified as a key in enhancing software development. But due to the complexity and amount of software systems produced, software development projects tend to ignore previous software cases. There are a number of reasons for this phenomenon, as for example that different persons work on the different projects and do not share knowledge about the cases, previous solutions are forgotten about in long development cycles, and most notably that it is hard to identify the similarity of a previous case – i.e. its requirements and solutions.

The main issue to solve this problem is developing a standard way of formalizing knowledge about complete software cases. Precisely, a software case consists of a requirements model, an architectural model, a detailed design and code. The requirements model represents the problem (or *problem space*), while the architectural model, detailed design and code together represent the solution (or *solution space*). This knowledge must be captured and stored for every software system in order to be retrievable for later development. Knowledge within requirements specifications, design documents and code is usually not well-structured, however. This fact hampers the ability to automatically process the knowledge and identify appropriate former cases.

Thus, effective mechanisms are needed to formalize knowledge about software cases, finding similarities between requirements specifications and displaying similarities as well as differences between software cases. Similarity can be measured by comparing the requirements models of a current and a previous case. The retrieved case is then intended to be reused by modifying those places that need rework (i.e. the places indicated by differences) and keeping those places that can be reused without modification (i.e. those places indicated by similarities).

After detailing the problems current practice struggles with, in the following section we present a first glance at the underlying formalisms that are intended to be used for solving the before mentioned problems.

3 The Solution – First Ideas

The intended framework consists of three elements:

Language A language is needed to precisely represent reusable software cases. Therefore, this language needs to cover requirements specification, modeling, mapping and transformation, and software case querying.

Engine The engine makes use of the new language and realizes the reuse mechanism by enabling identification and retrieval of previous software cases.

Methodology The methodology describes the complete life cycle of software cases which is defined by capturing and retrieving cases. The cases themselves are defined with the new language, and storage and retrieval is performed with the new engine.

In order to implement an effective software development system that achieves such objectives, different areas of software engineering and artificial intelligence need to be combined. Basic approaches are requirements engineering, reusable asset libraries (e.g. software product lines), graphical as well as textual modeling approaches, model-driven development and case-based reasoning.

The next sections describe approaches the requirements-driven software development system is based on from the view point of knowledge-based configuration and representation. Furthermore it is described how this system can benefit from a combination of those, respectively. Thus, we question "what can knowledge-based configuration and representation provide for the indicated topics?".

3.1 Requirements Engineering

For describing requirements of a software system, restrictions on the system or its parts and features of the system as a whole need to be defined. A typical approach from knowledge-based configuration is to provide a requirements specification language and a mapping to knowledge representation facilities like concepts, attributes and constraints. This mapping establishes a direct relation between the problem space and the solution space. With this formal approach requirements can be processed by the inference machine that belongs knowledge-based configuration. For an example for complex requirements specification in knowledge-based configuration systems we refer the interested reader to [1].

3.2 Software Product Lines

In recent years work on software reuse strongly focused on software product lines (SPL) [2, 3]. SPLs distribute the development effort of the reusable assets over the customers but providing an asset repository that can be used to derive new products.

In the ConIPF project, software product lines and knowledge-based configuration have been coupled [4]. This combination makes use of the key benefits of both worlds, which are:

- The product line approach provides a basic framework in which it is distinguished between development for reuse and development with reuse, called *domain engineering* and *application engineering*, respectively.
- Knowledge-based configuration uses formal representation facilities for modeling the reusable assets in a configuration model. Based on a correct and consistent model and a sound configuration process, this approach guarantees a correct, complete and consistent solution to a configuration problem.

The configuration model provides a textual description of the application domain. Also, the configuration solution is a textual description of the assets that are needed to assemble the configured product. In the asset store there are the reusable assets; i.e. models, documents and most notably the code. Based on the configuration solution the necessary assets are identified, retrieved from the asset store and assembled.

3.3 Model-driven Development

Model-driven development is based on modeling and meta-modeling approaches. Model-driven Architecture (MDA)[3] extends them with the concept of model transformation. Model-driven development uses a modeling language (an underlying abstract schema and a visible concrete syntax – e.g. Unifies Modeling Language (UML)[4]) and its semantics (the meaning of the language facilities) defining the use of models.

Model transformation is the next step that converts a model into another model. With this the stepwise transformation from a requiremtents model to executable code is envisaged.

3.4 Case-based Reasoning and case-based configuration

A major part of human expertise is believed to be past experiences. Case-based reasoning provides a model for representing experience in so-called *cases* (i.e.former configuration problems) and reusing them for solving new configuration problems. The knowledge base in case-based reasoning is the set of stored cases (i.e. the past experience). Analyzing a case is delayed until its retrieval for solving a new problem. However, at that time the problem is less haunting, due to the fact that one only needs to understand how differences between the problem in the recalled experience and the current problem affect the solution proposed in the recalled experience. The solution for the retrieved case is examined and applied to the current problem with suitable modifications [5].

A reasonable definition of a case is "a contextualized piece of knowledge representing an experience". There are two types of cases that can be distinguished:

- *Normative cases* capture the typical situation (e.g. reference configurations)- i.e. a situation that occurs more than once and that occurs in the same or at least very similar ways.
- The second type is *special cases*, the "out-of-the-normal" situations (e.g. customer-specific product extensions). This means remembering where variation from the normal took place.

However, not all the cases where some variation from the "normal" took place are relevant. Thus, special cases are only taken into account when they are significant for the overall context of the case. Given the configuration process, a case can be reused by using it as a solution, parts of it as a solution or modifying it to suit the current configuration problem (see e.g. [6]).

[3] http://www.omg.org/mda/
[4] http://www.uml.org

3.5 Software Development Methodologies

There are various software development methodologies that can be used, extended or seen as guidelines for defining a new methodology. Some well-known methodologies are the Feature-oriented Reuse Method (FORM) [7] which is based on the Feature-oriented Domain Analysis (FODA) [8], KobrA which is about component-based product line engineering with UML [9], PuLSE-I which deals with deriving instances from a product line infrastructure [10] and Configuration in Industrial Product Families (ConIPF) [4].

The ConIPF methodology offers full reuse of the complete asset repository to the complete staff due to its representation in the configuration model. This approach still lacks knowledge about the previously derived products, however. This means that still double work may be done where reuse would have been possible. By making use of a case-based approach, software product lines could be enhanced or even replaced with a requirements-driven reuse framework.

4 Summary

The ignorant reader may argue that the ReDSeeDS project tries to define yet another software development methodology. Well, this is true – at least to some extent. But this project combines several areas of research that have never been combined before.

A coherent combination of requirements engineering, reusable asset libraries, model transformation and querying can achieve effective reuse of the different assets types in software engineering. These asset types are requirements models, architecture models, detailed design and code. Current state in software reuse is limited to the solution and does not take the problem specification – i.e. the requirements specification – into account. But it is this requirements specification that can best express similarities and differences between software systems.

References

1. Thäringen, M.: Wissensbasierte Erfassung von Anforderungen. In Günter, A., ed.: Wissensbasiertes Konfigurieren. Infix (1995)
2. Bosch, J.: Design & Use of Software Architectures: Adopting and Evolving a Product Line Approach. Addison-Wesley (2000)
3. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
4. Hotz, L., Wolter, K., Krebs, T., Deelstra, S., Sinnema, M., Nijhuis, J., MacGregor, J.: Configuration of Industrial Product Families - The ConIPF Methodology. Aka Verlag, Berlin (2005)
5. Sasikumar, M.: Case-based Reasoning for Software Reuse. In: Knowledge Based Computer Systems-Research and Applications (International Conference on Knowledge-Based Computer Systems), Bombai, India, Narosa Publishing House, London (1996) 31–42
6. Pfitzner, K.: Case-based Configuration of Technical Systems. Künstliche Intelligenz **7/93** (1993) 24–30

7. Kang, K.C., Lee, J., Donohoe, P.: Feature-oriented reuse method (form). IEEE Software **19**(4) (2002) 58–65
8. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021 (1990)
9. Atkinson, C.: Component-based Product Line Engineering with UML. Addison-Wesley (2002)
10. Bayer, J., Gacek, C., Muthig, D., Widen, T.: PuLSE-I: Deriving Instances from a Product Line Infrastructure. In: 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems, Edinburgh, Scotland (2000) 237–245

Semantic Web Technology as a Basis for Planning and Scheduling Systems

Bernd Schattenberg and **Steffen Balzer** and **Susanne Biundo**
Dept. of Artificial Intelligence
University of Ulm, D-89069 Ulm, Germany
{firstname}.{lastname}@uni-ulm.de

Abstract

This paper presents an architecture for planning and scheduling systems that addresses key requirements of real-world applications in a unique manner. The system provides a robust, scalable and flexible framework for planning and scheduling software through the use of industrial-strength middleware and multi-agent technology. The architectural concepts extend knowledge-based components that dynamically perform and verify the system's configuration. The use of standardized components and communication protocols allows a seamless integration with third-party libraries and existing application environments.

The system is based on a proper formal account of hybrid planning, the integration of HTN and POCL planning. The theoretical framework allows to decouple flaw detection, modification computation, and search control. In adopting this methodology, planning and scheduling capabilities can be easily combined by orchestrating respective elementary modules and strategies. The conceptual platform can be used to implement and evaluate various configurations of planning methods and strategies, without jeopardizing system consistency through interfering module activity.

Introduction

Hybrid planning – the combination of hierarchical task network (HTN) planning with partial order causal link (POCL) techniques – turned out to be most appropriate for complex real-world planning applications (Estlin, Chien, & Wang 1997), like crisis management support (Biundo & Schattenberg 2001; Castillo, Fdez-Olivares, & González 2001). Here, the solution of planning problems often requires the integration of planning from first principles with the utilization of predefined plans to perform certain complex tasks.

Previous work (Schattenberg, Weigl, & Biundo 2005) introduced a formal framework for hybrid planning, in which the plan generation process is functionally decomposed into well-defined flaw detecting and plan modification generating functions. As an important feature of this approach, an explicit triggering function defines, which modifications are suitable candidates for solving which flaws. This allows to completely separate the computation of flaws from the computation of possible plan modifications, and in turn both computations can be separated from search related issues. The system architecture relies on this separation and exploits it in two ways: module invocation and interplay are specified through the triggering function while the explicit reasoning about search can be performed on the basis of flaws and modifications without taking their actual computation into account. This explicit representation of the planning strategy allows for the formal definition of a variety of strategies, and even led to the development of novel so-called *flexible* strategies.

The functional decomposition induces a modular and flexible system design, in which arbitrary system configurations –viz. planning and scheduling functionalities– can be integrated seamlessly. A prototype of this architecture served as an experimental environment for the evaluation of flexible strategies as well as a conceptual proof for the expandability of the system with respect to new techniques: namely, the integration of scheduling (Schattenberg & Biundo 2002; 2006) and probabilistic reasoning (Biundo, Holzer, & Schattenberg 2004; 2005).

While (Schattenberg, Weigl, & Biundo 2005) presented the theoretical framework for a straight-forward system design for hybrid plan generation, a number of functional and non-functional requirements are obviously not met by such an architectural nucleus when it comes closer to real-world application scenarios like crisis management support, assistance in telemedicine, personal assistance in ubiquitous computing environments, etc. Like any other mission critical software in these contexts, planning and scheduling systems should feature characteristics which call for highly sophisticated software support:

1. declarative, automated system configuration and verification – for fast, flexible, and safe system deployment and maintenance, and for an easy application-specific configuration tailoring
2. scalability, including transparency with respect to system distribution, access mechanisms, concurrency, etc. – for providing computational power on demand without additionally burdening system developers
3. standards compliance – for integrating third-party systems and libraries, and for interfacing with other services and software environments

Each of these characteristics represents a challenge in its own for any software environment, and this is in particular the case for planning and scheduling applications. This paper describes a novel planning and scheduling system ar-

chitecture which essentially addresses all of the above challenges, and shows how the formal framework of (Schattenberg, Weigl, & Biundo 2005) has been incorporated. It shows not only how modern software technology –in particular middleware and knowledge-based systems– can be successfully applied to a prototypical academic planning software, but also illustrates how (in principle) any planning and scheduling system can benefit from it. The resulting system performs a dynamical configuration of its components and even reasoning about the consistency of that configuration is possible. The planning components are transparently deployed, distributed (including an optimized concurrency), and load-balanced while retaining a relatively simple programming model for the component developer. Standardized protocols and components finally provide easy access to other software products and services.

The rest of this document is organized as follows: The next section presents the formal framework of hybrid planning on which our approach is based. Then a reference planning process model is defined, as an overview for the architecture. This is followed by a description of the architecture components, how the middleware is used and how a refined planning process model is realized. After that, there is a section devoted to the use of knowledge representation mechanisms in the system. The paper concludes with an overview over related work and some final remarks.

Formal Framework

Our planning system relies on a formal specification of hybrid planning (Schattenberg, Weigl, & Biundo 2005): The approach features a STRIPS-like representation of action schemata with PL1 literal lists for preconditions and effects and state transformation semantics based on respective atom sets. It discriminates primitive operators and abstract actions (also called complex tasks), the latter representing abstractions of partial plans. The plan data structure, in HTN planning referred to as *task network*, consists of complex or primitive task schema instances, ordering constraints and variable (in-)equations, and causal links for representing the causal structure of the plan. For each complex task schema, at least one *method* provides a task network for implementing the abstract action.

Planning problems are given by an initial task network, i.e. an abstract plan, a set of primitive and complex task schemata, and a set of methods specifying possible implementations of the complex tasks. A partial plan is a solution to a given problem, if it contains primitive operators only, the ordering and variable constraints are consistent, and the causal links support all operator preconditions without being threatened.

Flaws

The violation of solution criteria is made explicit by so-called *flaws* – data structures which literally "point" to deficiencies in the plan and allow for the problems' classification: A flaw $\mathtt{f}$ is a pair $(flaw, E)$ with $flaw$ indicating the flaw class and E being a set of plan components the flaw refers to. The set of flaws is denoted by $\mathcal{F}$ with subsets $\mathcal{F}_{flaw}$ for given labels $flaw$.

E.g., the flaw representing a threat between a plan step te_k and a causal link $\langle te_i, \phi, te_j\rangle$, is defined as: $(\mathtt{Threat}, \{\langle te_i, \phi, te_j\rangle, te_k\})$. In the context of hybrid planning, flaw classes also cover the presence of abstract actions in the plan, ordering and variable constraint inconsistencies, unsupported preconditions of actions, etc.

The generation of flaws is encapsulated by detection modules, i.e. functions that take as an argument a plan and return a set of flaws. Without loss of generality we may assume, that there is exactly one such function for each flaw class. The function for the detection of causal threats, e.g., is defined as follows:

$f^{det}_{CausalThreat}(P) \ni (\mathtt{Threat}, \{\langle te_i, \phi, te_j\rangle, te_k\})$ iff:
$te_k \not\prec^* te_i$ or $te_j \not\prec^* te_k$ in the transitive closure $\prec^*$ of P's ordering relation and the variable (in-) equations of P allow for a substitution σ such that $\sigma(\phi) \in \sigma(\mathrm{del}(te_k))$ for positive literals ϕ and $\sigma(|\phi|) \in \sigma(\mathrm{add}(te_k))$ for negative literals ϕ.

Modifications

The refinement steps for obtaining a solution out of a problem specification (which means to get rid of any flaws) are explicit representations of changes to the plan structure. A plan modification $\mathtt{m}$ is a pair $(mod, E^{\oplus} \cup E_{\ominus})$ with mod denoting the modification class. $E^{\oplus}$ and $E_{\ominus}$ are elementary additions and deletions of plan components, respectively. The set of all plan modifications is denoted by $\mathcal{M}$ and grouped into subsets $\mathcal{M}_{mod}$ for given classes mod.

The following structure represents adding an ordering constraint between to plan steps te_i and te_j: $(\mathtt{AddOrdConstr}, \{\oplus(te_i \prec te_j)\})$. Further examples of hybrid planning modifications are the insertion of new action schema instances, variable (in-) equations, and causal links, and of course the expansion of complex tasks according to appropriate methods.

As with the flaws, the generation of plan modifications is encapsulated by modification modules. These functions take a plan and a set of flaws as arguments and compute all possible plan refinements that solve flaws. E.g., promotion and demotion as an answer to a causal threat is defined as:

$$f^{mod}_{\mathtt{AddOrdConstr}}(P, \{\mathtt{f}, \ldots\}) \supseteq \{\ (\mathtt{AddOrdConstr}, \{\oplus(te_k \prec te_i)\}),\ (\mathtt{AddOrdConstr}, \{\oplus(te_j \prec te_k)\})\}$$

for $\mathtt{f} = (\mathtt{Threat}, \{\langle te_i, \phi, te_j\rangle, te_k\})$.

Refinement-based Planning

It is obvious that some classes of modifications address particular classes of flaws while others do not. This relationship is explicitly represented by the so-called *modification triggering function* α which relates flaw classes with suitable modification classes (cf. (Schattenberg, Weigl, & Biundo 2005)). As an example, causal threat flaws can in principle be solved by expanding abstract actions which are involved in the threat, by promotion or demotion, or by separating variables through in-equality constraints (cf. (Biundo & Schattenberg 2001)):

$$\alpha(\mathcal{F}_{\mathtt{Threat}}) = \mathcal{M}_{\mathtt{ExpandTask}} \cup \mathcal{M}_{\mathtt{AddOrdConstr}} \cup \mathcal{M}_{\mathtt{AddVarConstr}}$$

Please note, that the triggering function states nothing about the relationship of the actual flaw and modification instances.

Apart from serving as an instruction, which modification generators to consign with which flaw, the definition of the triggering function gives us a general criterion for discarding un-refineable plans: For any detection and modification modules associated by a trigger function α, f_x^{det} and $f_{y_1}^{mod}, \ldots, f_{y_n}^{mod}$ with $\mathcal{M}_{y_1} \cup \ldots \cup \mathcal{M}_{y_n} = \alpha(\mathcal{F}_x)$: if $\bigcup_{1 \leq i \leq n} f_{y_i}^{mod}(P, f_x^{det}(P)) = \emptyset$ then P cannot be refined into a solution.

A generic algorithm can then be defined which uses these modules (see Alg. 1): In a first phase, the results of all detection module implementations are collected. In a second phase, the resulting flaws are propagated according to the triggering function[1] α to the respective modification module implementations. If any flaw remains un-answered, a failure is indicated. A strategy module selects in a third phase the most promising modification, which is then applied to the plan. The algorithm is then called recursively with that modified plan. The strategy also serves as a backtracking point of the procedure.

Algorithm 1 A generic planning algorithm, based on explicit flaw and modification computation

plan(P, T, M**):**
$F \leftarrow \emptyset$
for all f_x^{det} **do**
  $F \leftarrow F \cup f_x^{det}(P)$
if $F = \emptyset$ **then**
  return P
$M \leftarrow \emptyset$
for all $F_x = F \cap \mathcal{F}_x$ with $F_x \neq \emptyset$ **do**
  answered $\leftarrow$ **false**
  for all f_y^{mod} with $\mathcal{M}_y \subseteq \alpha(\mathcal{F}_x)$ **do**
    $M' \leftarrow f_y^{mod}(P, F_x)$
    if $M' \neq \emptyset$ **then**
      $M \leftarrow M \cup M'$
      answered $\leftarrow$ **true**
  if answered $=$ **false then**
    return fail
return plan(apply$(P, f_z^{strat}(P, F, M))$, T, M)

(Schattenberg, Weigl, & Biundo 2005) demonstrated how planning strategies are formally defined in that framework and illustrated its potential. Several adaptations of strategies taken from the literature were presented, as well as a set of novel *flexible* planning strategies. The latter exploit the explicit flaw and modification information, which allows for selection schemata that are not defined along flaw or modification type preferences, but perform an opportunistic way of plan generation. In a first series of experiments, a set of flexible and fixed, classical strategies competed on a former planning competition benchmark for HTN systems, the UMTranslog domain as it has been shipped with the UMCP system. It turned out, that flexible strategies are not only competitive to their fixed ancestors, but also showed high optimization potential and –due to their opportunistic modus operandi– can easily be combined.

[1] This makes the algorithm completely independent from the actually deployed module implementations.

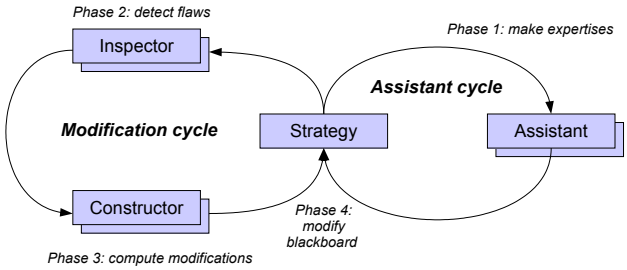


Figure 1: The reference planning process model for PANDA

Architecture Overview

In following the proposed design of the last sections, the basic architecture of PANDA (*Planning and Acting in a Network Decomposition Architecture*) is that of a multiagent-based blackboard system. The agent societies map directly on the presented module structure, with the agent metaphor providing maximal flexibility for the implementation.

Inspectors are implementations of flaw detection modules. There is one such agent per flaw class.

Constructors are agent incarnations of the plan modification generating modules. We may assume, that each modification class is represented by one such agent.

Assistants provide shared inference and services which are required by other agents. Assistant agents propagate implications of temporal action information transparently into the ordering constraints, simplify variable constraints, etc.

Coordinators implement the planning strategy module by synchronizing the execution of the other agents and performing the modification selection. Currently only one coordinator is allowed in the system – the so called *strategy*.

Figure 1 shows the reference planning model for PANDA, which defines the agent interaction. A planning *cycle* corresponds to an iteration of a monolithic algorithm (cf. Alg. 1). It consists of two sub-cycles that are divided into 4 phases (see Fig. 1), in which the agents execute concurrently.

Phase 1: Assistants repeatedly derive additional information and post it on the blackboard. This phase ends when no member of the assistant community added information anymore.

Phase 2: Inspectors analyze the current plan residing on the blackboard and post the results, i.e. the detected and prioritized flaws, to the strategy and to the constructors assigned to them.

Phase 3: Constructors compute all possible modifications for the received flaws and send them along with a priorization to the strategy.

Phase 4: The strategy compares all results received from the inspectors and constructors and selects one of the modifications to be executed on the current plan. A planning cycle is hereby completed and the system continues with phase 1 to execute the next planning cycle.

Phase transitions are performed only by the strategy when all participating agents have finished execution. Thus, the phase transitions can be viewed as synchronization points within the planning process. The strategy modifies the plan until no more flaws are detected or an inspector published a flaw for which no resolving modification is issued. In the first case, the current plan constitutes a solution to the given planning problem, in the latter case the planning process has reached a dead end and the system has to backtrack in order to execute a different modification on the strategy's stack. The blackboard is implemented as a stack that stores the current plan, all derived information, and performed modifications. This structure enables the strategy to backtrack the system to a certain point.

The following sections will show, how the reference planning process model has been implemented, using middleware and knowledge-based technology. The chosen multiagent-system is based on on an industrial-strength middleware and uses an explicit knowledge representation in the implementation of the necessary protocols. A refined version of the reference model will then allow us to exploit agent concurrency more efficiently.

A Knowledge-based Middleware

Core Components

An obvious implementation for a planning system following the reference process model would still run in a sole Java virtual machine, viz. on a single computational resource. This stands in contrast to the requirements that complex and dynamic application domains demand. For crisis management support, e.g., information must be gathered from distributed and even mobile sources, the planning process requires a lot of computational power, etc. So scalability and distribution play key roles in the proposed system architecture, while maintaining the (simple but effective) reference process.

The main aspect in middleware systems like application servers is to hide the mechanisms that enable the distributed handling of objects from the programmer. Thus, it is possible to develop distributed applications much more efficiently. In other words, such middleware systems make distribution issues *transparent* to the programmer. Examples for transparency in middleware systems are location transparency, scalability transparency, access transparency, concurrency transparency etc. (Emmerich 2000). Scalability transparency for example means that it is completely transparent to the programmer how a middleware system scales in response to a growing load. In summary, middleware systems take care of the complexity of handling distributed objects and provide an abstract and easy to use API to the programmer.

In order to benefit from application server technology, the PANDA system builds upon the open-source implementation JBoss (Stark 2003). The most important components that a *Java 2 Enterprise Edition – J2EE* (Sun 1999) based application server delivers w.r.t. this work are the following:

- *Enterprise Java Beans – EJBs* are the objects that are managed by an application server. All transparency aspects apply to them. They are the building blocks of a distributed J2EE application (Sun 2003).
- The *Java Naming and Directory Interface – JNDI* is the directory service that enables location and access transparency. It provides a mapping between Java names and remote interfaces of Java objects. The access to all EJBs and other services of the application server is provided through this interface.
- The *Java Messaging Service – JMS* enables asynchronous and location transparent communication between Java components (especially EJBs) beyond virtual machine boundaries. So this service will be of interest when it comes to communication between the different components of PANDA.

In addition to the features described above, application server implementations also cover aspects like security, database access, transaction management etc. They all belong to the J2EE specification. However, a full discussion of their benefits for the PANDA system is beyond the scope of this paper.

Although the application server technology provides powerful mechanisms, we still need more support for realizing the multiagent system functionalities of the reference model. Instead of investigating proprietary agent life-cycle management and communication mechanisms, we decided to take advantage of the work of the Foundation for Intelligent Physical Agents (FIPA), which has been developing standards for that area since 1996. The second core component which is chosen to implement all agent specific features, i.e. to take care of the agent computing capabilities of the system, is therefore BlueJADE (Cowan & Griss 2002). BlueJADE integrates the well-known multiagent framework JADE (Bellifemine *et al.* 2005) with JBoss. This integration puts the agent system life cycle under full control of the application server, that means all distribution capabilities of the application server apply to the agent societies. Access to the JADE agent API is provided by BlueJADE's ServiceMBean interface. It has been selected as the agent computing platform for PANDA because of the following key features:

It is a FIPA-compliant agent platform and provides a library of ready-to-use agent interaction protocols. This enables the PANDA system to interact with other multi-agent-systems and their services.

It is a distributed system, i.e. its agents can be spread transparently over several agent containers running on different nodes in a network, including the migration of running agents between containers. These features can be exploited for distributed information gathering and (automated) load balancing. It has to be noted, that this kind of distribution management is "on top" of the middleware facilities: agent migration typically anticipates pro-actively the computation or communication load in a relative abstract manner, while middleware migration reacts on such load changes based on very low-level operating system specific sensors. It makes sense to provide both mechanisms in parallel, e.g. to migrate scheduling inspector agents, which are known to require much computational resources onto dedicated compute servers.

The LEAP extension (Caire 2005) adds support for ubi-

quitous computing. Agents are able to run even on mobile devices such as Java capable cellular phones, PDAs, etc., which are all coveted user-interfaces in many application domains.

BlueJADE supports application defined content languages and ontologies. So a DAML-based content language can be easily integrated (this will be discussed later).

The system comes with a set of sophisticated graphical debugging tools. This speeds up the development process significantly.

The knowledge representation and reasoning facilities which are used throughout the system constitute the third core component. During its development, the PANDA framework required an increasing amount of knowledge that represents planning related concepts (flaw and modification classes, etc.), the system configuration (which inspectors, constructors, and strategies to deploy), and the plan generation process itself (the reference process, including the backtracking procedure, etc.). Most of this knowledge is typically represented implicitly through algorithms and data structures. To make it explicit and modifiable without touching the system's implementation, it must be extracted and represented in a common knowledge base which uses a representation formalism that is expressive enough to capture all modeling aspects on one side and that allows efficient reasoning on the other side. As a result of this, the system can be configured generically and that configuration can be verified on a higher semantic level.

There is a large number of knowledge representation systems available on the market which promise to meet the requirements. But since special regard is spent on standards compliance, the *DARPA Agent Markup Language – DAML* (Horrocks, Harmelen, & Patel-Schneider 2001) has been chosen as the grounding representation formalism for this task. It combines the key features of description logics (Baader *et al.* 2003) with Internet standards such as XML or RDF (Manola & Miller 2003) and – even more important – powerful reasoners and other freely available tools exist to integrate the language into applications. In our case, the knowledge encoded in DAML must be made available to the Java programming language. Therefore, a Java object model is necessary that provides mappings in both directions – from Java to DAML and vice versa. The JENA API (McBride 2000) from Hewlett Packard delivers an in-memory object model of a DAML document along with a rich API to query and manipulate it. By using DAML as the content language for the BlueJADE agent communication and also as the language for describing system configurations and communication means, we achieve a homogeneous representation in the system.

Last, but not least, it is of course necessary to integrate a suitable description logic system to store the knowledge and to reason about it. The RACER system (Haarslev & Möller 2001) has the essential capabilities that are required: a DAML codec, an efficient reasoning component, and a knowledge store based on a client-server architecture.

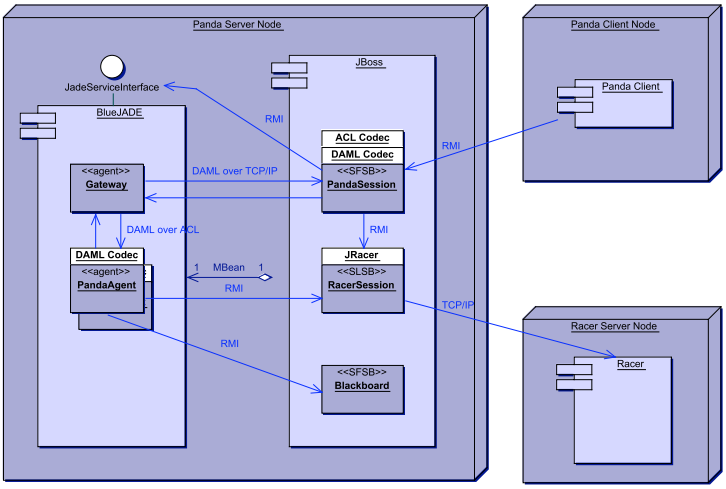


Figure 2: Static system structure of PANDA

The System Structure

Figure 2 shows PANDA's static system structure as a UML deployment diagram. The association arrows indicate which components communicate with each other. Their labels denote the transport protocols being used. BlueJADE is aggregated by JBoss as a (Service-) MBean. It's functionality is exposed via the *JadeServiceInterface*.

The PANDA *Client* component represents the client application that controls the PANDA system. Currently, an RMI-based communication is used. The PANDA client obtains an interface to the PANDA system by querying JBoss's directory service JNDI. But also web-based approaches using SOAP or HTTP are supported. In this way, JBoss provides technologies like Web Services and Java Servlets.

Regarding the integration with the JBoss infrastructure, the PANDA prototype defines three specializations of EJBs for non-agent system components: the interface to the RACER system, to the blackboard, and to the agent society (from outside the system).

Access to the Racer server is provided by the *RacerSessionBean*. The main reason for integrating the Racer system via an EJB proxy is that all components that depend on the Racer system – i.e. EJBs and Agents – are able to access it transparently. They do not need to know its IP address or socket number. Furthermore, the RacerSessionBean can be viewed as generic integration approach for all kinds of reasoner architectures. The communication between Racer and the RacerSessionBean is realized with the *JRacer* client API which translates Java method calls into Lisp function calls. It should be emphasized that each component that obtains a reference to the RacerSessionBean gets its own instance – as usual for SessionBeans. Therefore, queuing of requests is delegated to the Racer server. In a similar fashion, the *BlackboardSessionBean* represents a proxy to the blackboard component.

The *PandaSessionBean* represents the facade by which the PANDA client configures and controls the planning process. It uses the RacerSessionBean to derive the agents and

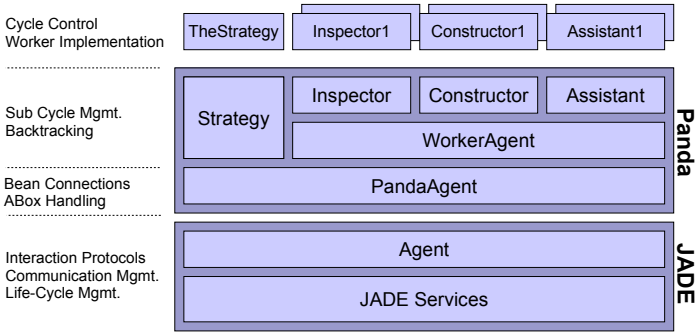


Figure 3: The logical layer structure of the agent framework

their implementations that must be instantiated and creates them using the *JadeServiceInterface*. Communication with the agent framework is done via the *JadeBridge* class of the BlueJADE package, which creates and accesses agent messages (see below) in an object-oriented manner.

Basically, two main *classes of agents* exist in the BlueJADE agent container. The first is the class of standard agents that come with the JADE and BlueJADE software packages. They provide the FIPA infrastructure, several debugging tools and JADE specific communication services. The *Gateway* agent's role is to mediate messages between JADE agents and components outside the JADE agent container. It's counterpart in the EJB container is the JadeBridge. The Gateway sends and receives stringified messages in the *Agent Communication Language* (ACL) via a TCP/IP socket connection.

Custom agents in the PANDA system, i.e. all agent types from the reference model, are all derived from the `PandaAgent` class which encapsulates low level data conversion and communication mechanisms. The PANDA agents form the second class in the JADE agent container. The `PandaAgent` class on its part is derived transitively from the JADE agent base class `Agent` which provides the integration into the JADE agent container (cf. Figure 3).

The reference model (Fig. 1) omits the actual means for *calling* agents, in a distributed implementation, these remote calls are typically message based. From the agent container's point of view, agents in the JADE agent container and the *PandaSessionBean* communicate by using messages encoded in the agent communication language *FIPA-ACL* (FIP 2002b) (in short ACL). ACL is a language based on speech-act theory, i.e. every message describes an action that is intended to be carried out with that message simultaneously (e.g. the request "compute detections"). Such intentions are called *performatives*. ACL defines formal semantics for performatives (FIP 2002a) that induce basic interaction protocols upon which more complex protocols like contract nets and auctions are built.

Besides parameters that are necessary for communication like performative name, participant information, etc., an ACL message includes parameters that describe the content that is intended for the receiving participant like the *content language* the content is encoded in, the domain the content refers to, etc. In order to be qualified for the use as a content language in an ACL message, a language must meet certain requirements that are induced by the semantics of the performatives. For example a *request* requires always at least an action to be delivered with the content. Otherwise the agent that receives the request does not know what it is requested to do. Furthermore, when an agent *informs* another agent about the result of an action, the content must contain the propositions that represent the result. Thus, a content language must at least provide representations of actions and propositions, so the agents are able to interact in a meaningful way. The content language that is used by the PANDA agents is described below.

The Planning Process

Figure 4 gives an overview of the refined model of the planning process that was taken as the basis for implementation. The white colored states specify the life-cycle management of a planning session (initializing the process, starting planning, suspending it, etc.). Each state transition is labeled with the triggering ACL message and its originator: `sender:performative` followed by `action` or `proposition`. Senders can also be described by their class, e.g. `Worker` denotes all worker agents. The same applies to propositions and actions, e.g. `Compute` denotes the action `Compute` and all sub-actions like `Inspect`, `Construct`, etc.

The planning process starts in an artificial `undefined` state in which all agents are deployed and send agreements for their initialization process. After that, the strategy informs all agents, that the system is initialized, and this is where the reference model started with phase 1: The assistants are requested to perform their inference on which they have to agree. After their processing (leaving the `assisting` state), the inspectors are requested to search for flaws (phase 2), and so on.

Please note, that not all states have been modeled in the state machine. Most states are abstract in order to reduce complexity of the state automaton while maintaining a degree of granularity that allows the user to monitor the planning process. E.g., the state `backtracking` summarizes all possible sub-states that describe the interaction between each particular worker agent and the strategy.

The refined planning process model has two major improvements over the reference model: First, it extends agent concurrency. The reference planning process model in Fig. 1, defines the agent classes to execute one after another, synchronized by phase transitions. In that model, concurrency is only allowed within a particular phase. But especially between phase 2 and 3 such a synchronization is too strict, because a constructor must wait until the last inspector has finished, even if a constructor has already received all flaws it requires for computation. Constructors should therefore be able to decide on their own when to start execution. The refined process model reflects this by a combined `inspecting&constructing` state. The constructors' behavior has therefore been changed from reactive to proactive – resulting in a stronger notion of agency.

Second, an enhanced backtracking procedure allows for the implementation of optimized and more sophisticated

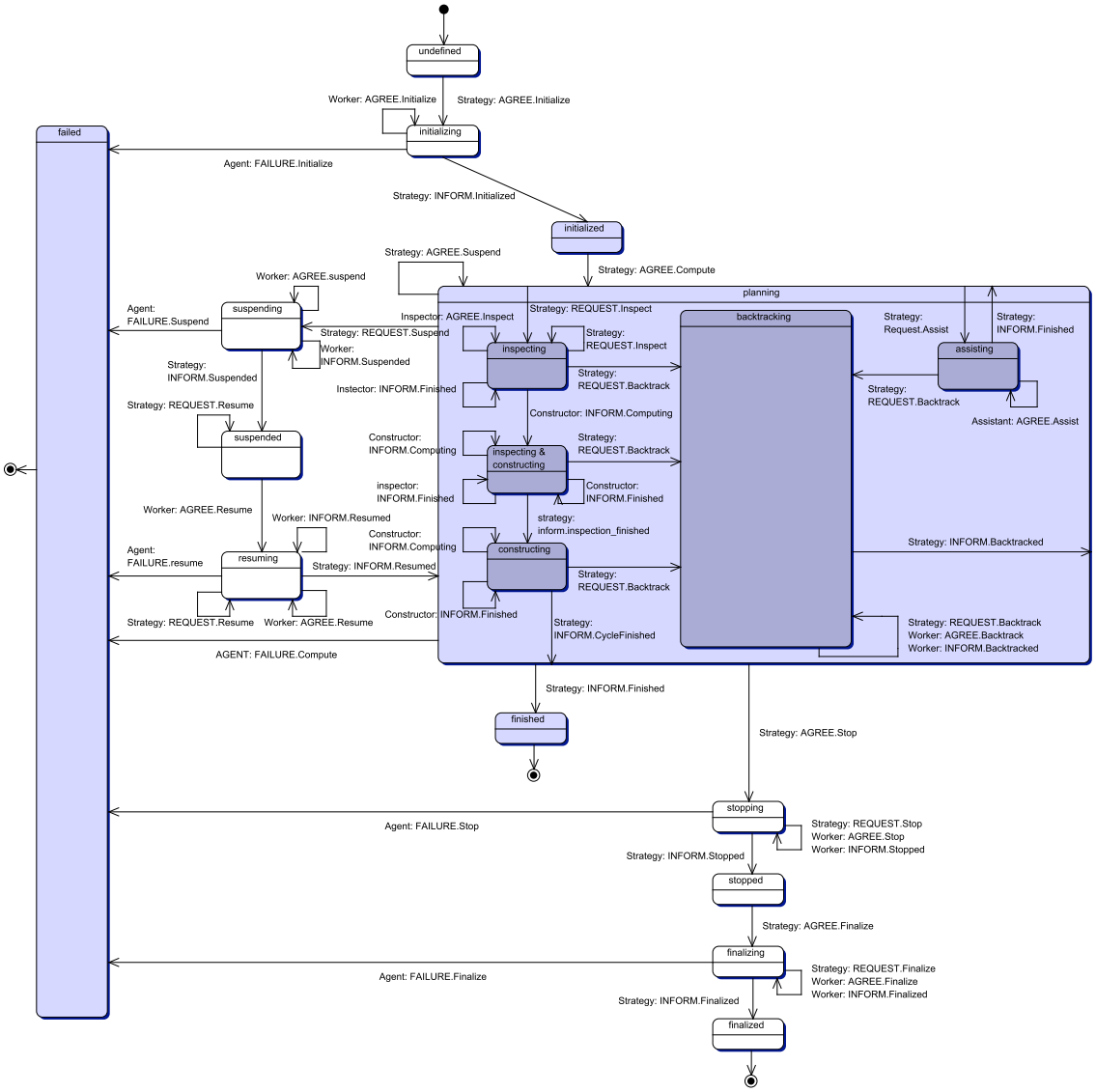


Figure 4: The refined PANDA planning process model

reasoning techniques, including worker agents, i.e. assistants, inspectors, and constructors, to be equipped with a state history or caches, etc. To keep backtracking consistent, the worker agents now participate in the backtracking process: they have to synchronize via agreement statements and then notify the strategy when they are finished (cf. state transitions from `backtracking`).

Thus, the agent behavior is extended by a backtracking mechanism with three core capabilities: First, a synchronized restart of the system must be guaranteed, i.e. a restart can only take place, if all worker agents have finished backtracking. Second, the different granularity of the state histories of agents working in different sub-cycles is considered. Assistants can be executed multiple times in a planning cycle, whereas inspectors and constructors will be executed only once. Therefore, assistants must be backtracked independently from Inspectors and Constructors. Third, in order to backtrack the system immediately, the strategy must be able to interrupt the worker agents' execution, the agents therefore perform their computations in a non-blocking way.

In summary, the enhanced mechanisms for concurrency and backtracking allow the system to benefit from early fail decisions in terms of an increased performance: Non-repairable inconsistencies are typically very quickly detected and processed by constructors.

Ontology-based Components

Like it has been mentioned before, DAML is used as representation formalism to describe and share knowledge in the PANDA system, ranging from flaw communication to system state transitioning. It is one of the emerging standards in the Semantic Web community for representing and communic-

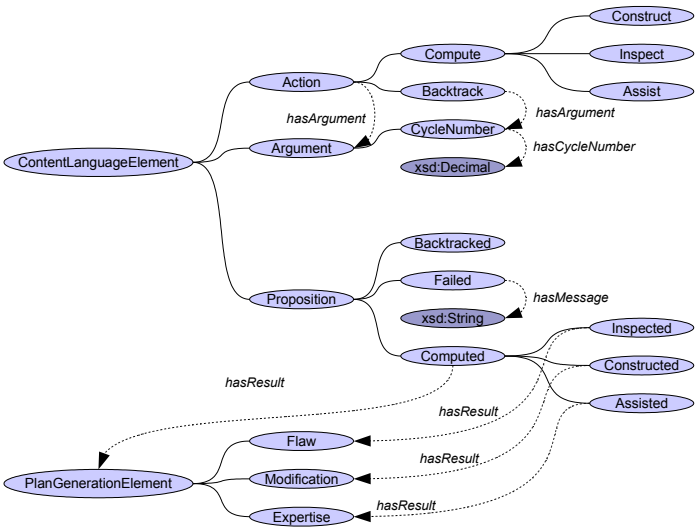


Figure 5: The content language ontology

ating knowledge (Horrocks, Harmelen, & Patel-Schneider 2001). It ensures interoperability with third-party systems like RACER and forms the basis for communicating knowledge among agents. Most important, it enables knowledge to be represented in a uniform, explicit, and declarative manner, so the system becomes more robust, flexible, and maintainable.

To be able to use DAML as a content language in ACL messages (recall the speech act structure), at least actions and propositions must be able to be represented within DAML (Schalk *et al.* 2002). This is sufficient for the needs of PANDA. Figure 5 shows the ontology that provides the concepts to enable DAML-based communication. Property cardinalities have been omitted for clarity.

Actions can have arguments, e.g., the action sub-concept `Backtrack` must come with a `CycleNumber` whose value is represented as the XML-schema type `decimal`. So an action could be compared with a method signature without argument order. In PANDA, every argument of an action is modeled in the ontology in order to give it a formal semantics. Therefore, in contrast to (Schalk *et al.* 2002), the argument order does not have to be considered. Propositions are currently only used to represent `ActionResults`. The sub-concept `Computed` carries the `PlanGenerationElements` that are the results of the worker agents' computations, e.g. a `Constructed` proposition has an `Modification` element as a result. The content of an ACL message is represented by instances of the PANDA system ontology embedded in a DAML-document. JENA takes care of encoding and decoding the DAML content of ACL messages. For any content that has to be sent, its JENA object model is constructed using the described ontology. After that, the model is serialized and inserted into the appropriate ACL message. The decoding of DAML content works exactly the opposite way. The object model of the DAML content is constructed by parsing its serialized representation and can then be queried with the JENA API.

DAML plays its second key role in the automated configuration of the agent container (Figure 6 shows the underlying ontology). The configuration process is composed of two sub-processes. First, the agents that are part of the planning process must be instantiated. The PandaSessionBean uses RACER to derive the leaf concepts of `PandaAgent` and to determine the implementation assignments `ImplementationElements` of the Worker-Agents. In the example of Figure 6, the assigned implementation for the `Inspector1` agent is an instance of Java class `panda.jade.agent.Inspector1Impl`. After being created, the PANDA agents insert their descriptions into the ABox of RACER, so RACER keeps track of the deployed agent instances.

Second, the communication links, viz. the implementation of the triggering function α, must be established between the instantiated agents. RACER is used by each PANDA agent on startup to derive its communication links to other agents, i.e. from which agents it will receive and to which agents it has to send messages. This is done by using the ontology to derive the dependencies between agents from defined dependencies between the flaws and modifications: The system ontology specifies which agent instance implements which type of `Inspector`, and it does the same for the constructor agents. RACER derives from that, which flaw and modification types will be generated by the agent instances, and if the model includes an α-relationship between them (`solved-by`), the agent instances' communication channels are linked. Based upon the subsumption capabilities that come with DAML and description logics, it is even possible to exploit sub-class relationships between `PlanGenerationElements` (illustrated by the bold printed concept connections in Fig. 6). An example for a modification class hierarchy are ordering relation manipulations with sub-classes *promotion* and *demotion*. Regarding flaws, the system ontology distinguishes *primitive* open preconditions and those involving *decomposition axioms* (Biundo & Schattenberg 2001).

A knowledge based configuration offers even more benefits: Imagine a less informed configuration mechanism, say, reading a respective file in XML format, that holds the descriptions on the agents to be loaded and the message links to be established between them as a representation of the triggering function α. Semantic verification can then only be based on type checking by, e.g., Java class loaders. In the presented architecture, the system model can be checked on startup for possible inconsistencies in a verification step of the planning process in state `initializing` before plan generation starts (cf. Fig. 4). An example for such an inconsistency is a constructor that is missing a link to a flaw, warnings can be issued for flaws and modifications without implementations of their generating agents, etc.

Related Work

There are two major agent-based planning architectures on the market. In the O-Plan system (Tate, Drabble, & Kirby 1994), a blackboard is examined by (in our terminology combined inspector and constructor) modules that write

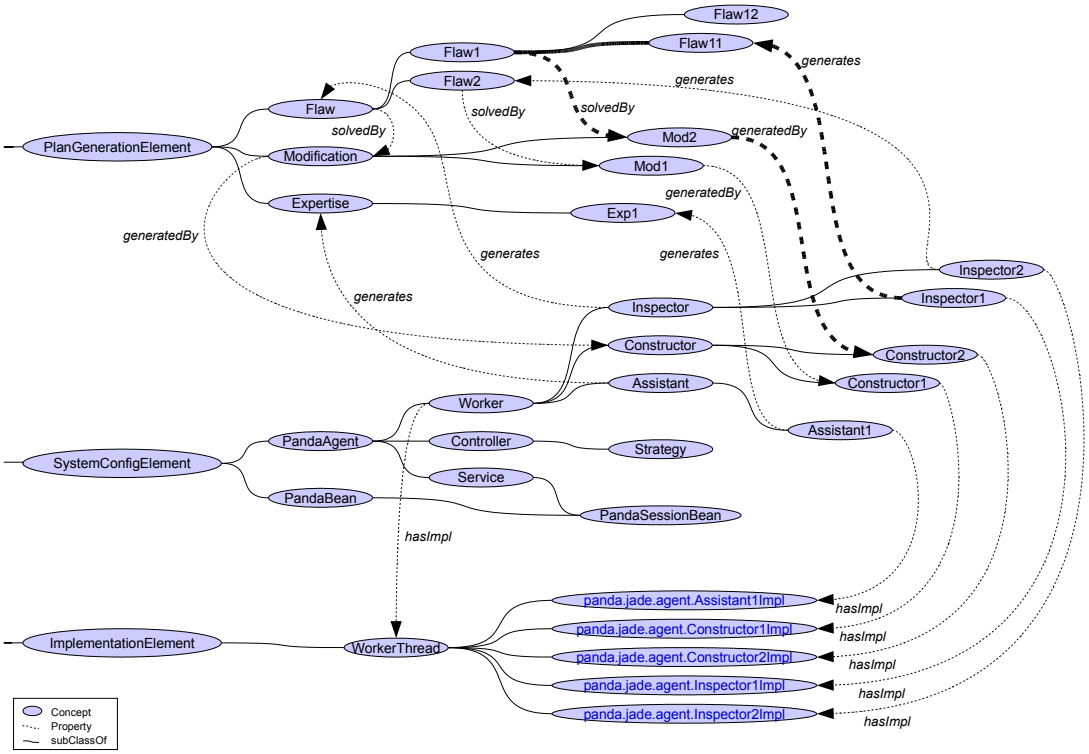


Figure 6: The system configuration ontology

their individually highest ranked flaw on the agenda of a search controller. This controller selects one agenda entry and triggers the respective module to perform its highest prioritized modification. O-Plan has been extended by a workflow-oriented infrastructure, called the I-X system integration architecture (Tate 2000). A plug-in mechanism serves as an interface to various (application tailored) tools. The planner itself is a monolithic system structure.

The Multi-agent Planning Architecture MPA (Wilkins & Myers 1998) relies upon a very generic agent-based approach. It executes an agent society in which designated coordinators decompose the planning problem into sub-problems, which are solved by subordinated groups of agents that may again decompose the problem again. Single agents return their solutions to their associated managers, which synthesizes the overall solution of its sub-agents. Communication of queries and results is based on the highly abstract KQML formalism. To our knowledge, no (standardized) middleware functionality has been incorporated.

The SIADEX architecture (de la Asunción *et al.* 2005) uses XML-RPCs for building a distributed planning environment, that is accessible via standardized HTTP and Java protocols. The architecture decouples a (monolithic) planning server, knowledge base management, and execution monitoring.

In order to address connectivity issues, some planning systems offer their functionality as web service. Examples are the CGI-based O-Plan interface (Tate & Dalton 2003) and the approach in (Tsoumakas *et al.* 2005), where a planner uses SOAP for communication and WDSL for presentation of the service. Although this view helps in enhancing the accessibility of planning software, the system (development) itself is not directly supported.

A representative for an application framework for building planning applications is Aspen (Fukunaga *et al.* 1997). It provides planning-specific data infrastructure, supportive inference mechanisms, and algorithmic templates, in order to facilitate rapid planning application development "out-of-the-box". It does not support the development of (standardized) concurrent planning functionality.

None of the above systems or architectures features a flexible, knowledge-based configuration of the plan generation process.

Conclusions and Future Work

We have presented a novel architecture for planning systems. It relies on a formal account of hybrid planning, which allows to decouple flaw detection, modification computation, and search control (Schattenberg, Weigl, & Biundo 2005). Planning capabilities, like HTN and POCL, can easily be combined by orchestrating respective elementary modules via an appropriate strategy module. The implemented system can be employed as a platform to implement and evaluate various planning methods and strategies. It can be easily extended to additional functionality, like integrated scheduling (Schattenberg & Biundo 2002; 2006) and probabilistic reasoning (Biundo, Holzer, & Schattenberg 2004; 2005), without implying changes to the deployed modules

– in particular flexible strategy modules – and without jeopardizing system consistency through interfering activity.

This work has investigated three main areas of interest of the PANDA planning system. The incorporation of higher semantics by making use of knowledge representation and inference techniques extends the capabilities of the system significantly in both functional and non-functional manner. Verification can be performed on a much higher level, and the system becomes more flexible and configurable the more hard-coded knowledge is extracted and described declaratively. With the use of application server technology and standardized communication protocols, PANDA has laid the foundation for a distributed system that is able to handle real-world application scenarios in an adequate manner. Still much work has to be done to evolve PANDA to a full-fledged planning web service, but application server technology seems to provide the appropriate architectural basis.

We plan to deploy this system as a central component in projects for assistance in telemedicine applications as well as for personal assistance in ubiquitous computing environments.

Future versions will not only keep the agents but also the planning state and agent messages in the system ontology and ABox in order to extend the verification capabilities of the system, including multiple ABoxes to enable multi-session planning. This will make the PANDA system even more robust w.r.t. corrupted agent behavior by reasoning in real-time over the dependencies between system states, possible actions and sent messages that trigger state transitions. To achieve this, knowledge about communication (i.e. message performatives, sender, receivers etc.) and interaction (i.e. FIPA-protocols and the planning process model) will be incorporated into the description logics representation of the system.

By extracting the hard-coded planning process model, describing it in a declarative manner, and executing it on a generic engine that uses this description as process template, changes to the planning process would not involve a change of code anymore. The process model itself an then be verified by transforming it into a petri net representation (Narayanan & McIlraith 2002).

References

Baader, F.; Calvanese, D.; McGuinness, D.; and Nardi, D. 2003. *The Description Logic Handbook*. Cambridge.

Bellifemine, F.; Caire, G.; Trucco, T.; and Rimassa, G. 2005. JADE programmer's guide. `http://jade.tilab.com/doc/programmersguide.pdf`.

Biundo, S., and Schattenberg, B. 2001. From abstract crisis to concrete relief – A preliminary report on combining state abstraction and HTN planning. In Cesta, A., and Borrajo, D., eds., *Proceedings of the 6th European Conference on Planning (ECP-01)*.

Biundo, S.; Holzer, R.; and Schattenberg, B. 2004. Dealing with continuous resources in AI planning. In *Proceedings of the 4th International Workshop on Planning and Scheduling for Space (IWPSS'04)*, number 228 in WPP, 213–218. ESA-ESOC, Darmstadt, Germany: European Space Agency Publications Division.

Biundo, S.; Holzer, R.; and Schattenberg, B. 2005. Project planning under temporal uncertainty. In Castillo, L.; Borrajo, D.; Salido, M. A.; and Oddi, A., eds., *Planning, Scheduling, and Constraint Satisfaction: From Theory to Practice*, volume 117 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. 189–198.

Caire, G. 2005. LEAP users guide. `http://jade.tilab.com/doc/LEAPUserGuide.pdf`.

Castillo, L.; Fdez-Olivares, J.; and González, A. 2001. On the adequacy of hierarchical planning characteristics for real-world problem solving. In Cesta, A., and Borrajo, D., eds., *Proceedings of the 6th European Conference on Planning (ECP-01)*.

Cowan, D., and Griss, M. 2002. Making software agent technology available to enterprise applications. Technical Report HPL-2002-211, Software Technology Laboratory, HP Laboratories, Palo Alto. `http://www.hpl.hp.com/techreports/2002/HPL-2002-211.pdf`.

de la Asunción, M.; Castillo, L.; Fdez.-Olivares, J.; García-Pérez, O.; González, A.; and Palao, F. 2005. Knowledge and plan execution management in planning fire fighting operations. In Castillo, L.; Borrajo, D.; Salido, M. A.; and Oddi, A., eds., *Planning, Scheduling, and Constraint Satisfaction: From Theory to Practice*, volume 117 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. 159–168.

Emmerich, W. 2000. *Engineering Distributed Objects*. Wiley. ISBN: 0-471-98657-7.

Estlin, T. A.; Chien, S. A.; and Wang, X. 1997. An argument for a hybrid HTN/operator-based approach to planning. In Steel, S., and Alami, R., eds., *Proceedings of the 4th European Conference on Planning (ECP-97)*, volume 1348 of *LNAI*, 182–194. Springer.

FIPA - Foundation for Intelligent Physical Agents. 2002a. *FIPA-ACL Communicative Act Library Specification*. `http://www.fipa.org/specs/fipa00037/SC00037J.pdf`.

FIPA - Foundation for Intelligent Physical Agents. 2002b. *FIPA-ACL Message Structure Specification*. `http://www.fipa.org/specs/fipa00061/SC00061G.pdf`.

Fukunaga, A.; Rabideau, G.; Chien, S.; and Yan, D. 1997. Towards an application framework for automated planning and scheduling. In *Proceedings of the 1997 International Symp. on AI, Robotics & Automation for Space*.

Haarslev, V., and Möller, R. 2001. Description of the racer system and its applications. In Goble, C.; McGuinness, D. L.; Möller, R.; and Patel-Schneider, P. F., eds., *Working Notes of the 2001 International Description Logics Workshop (DL-2001)*, volume 49 of *CEUR Workshop Proceedings*. ISSN 1613-0073.

Horrocks, I.; Harmelen, F.; and Patel-Schneider, P. 2001. DAML+OIL Specification (March 2001). `http://www.daml.org/2001/03/daml+oil-index.html`.

Manola, F., and Miller, E. 2003. RDF primer. `http://www.daml.org/2001/03/daml+oil-index.html`.

McBride, B. 2000. Making software agent technology available to enterprise applications. `http://www-uk.hpl.hp.com/people/bwm/papers/20001221-paper/`.

Narayanan, S., and McIlraith, S. A. 2002. Simulation, verification and automated composition of web services. In *WWW '02: Proceedings of the 11th International Conference on World Wide Web*, 77–88. ACM Press. ISBN 1-58113-449-5.

Schalk, M.; Liebig, T.; Illmann, T.; and Kargl, F. 2002. Combining FIPA ACL with DAML+OIL - a case study. In Cranefield, S.; Finin, T.; and Willmott, S., eds., *Proceedings of the Second International Workshop on Ontologies in Agent Systems*.

Schattenberg, B., and Biundo, S. 2002. On the identification and use of hierarchical resources in planning and scheduling. In Ghallab, M.; Hertzberg, J.; and Traverso, P., eds., *Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems (AIPS-02)*, 263–272. AAAI.

Schattenberg, B., and Biundo, S. 2006. A unifying framework for hybrid planning and scheduling. In Freksa, C., and Kohlhase, M., eds., *KI 2006: Advances in Artificial Intelligence, Proceedings of the 29th German Conference on Artificial Intelligence*, LNAI. Springer. to appear.

Schattenberg, B.; Weigl, A.; and Biundo, S. 2005. Hybrid planning using flexible strategies. In Furbach, U., ed., *KI 2005: Advances in Artificial Intelligence, Proceedings of the 28th German Conference on Artificial Intelligence*, volume 3698 of *LNAI*, 258–272. Springer.

Stark, S. 2003. *JBoss Administration and Development*. JBoss Group, LLC, second edition. JBoss Version 3.0.5.

Sun Microsystems. 1999. *Simplified Guide to the Java 2 Platform, Enterprise Edition*. `http://java.sun.com/j2ee/white/j2ee_guide.pdf`.

Sun Microsystems. 2003. *Enterprise JavaBeans 2.1 Documentation*. `http://java.sun.com/products/ejb/docs.html`.

Tate, A., and Dalton, J. 2003. O-plan: a common lisp planning web servide. In *Proceedings of the International Lisp Conference*, 12–25.

Tate, A.; Drabble, B.; and Kirby, R. 1994. O-Plan2: An architecture for command, planning and control. In Zweben, M., and Fox, M., eds., *Intelligent Scheduling*. Morgan Kaufmann. 213–240.

Tate, A. 2000. Intelligible ai planning. In *Research and Development in Intelligent Systems XVII, Proceedings of the ES2000, The 20th British Computer Society Special Group on Expert Systems International Conference on Knowledge Based Systems and Applied Artificial Intelligence*, 3–16. Springer.

Tsoumakas, G.; Meditskos, G.; Vrakas, D.; Bassiliades, N.; and Vlahavas, I. 2005. Web services for adaptive planning. In Castillo, L.; Borrajo, D.; Salido, M. A.; and Oddi, A., eds., *Planning, Scheduling, and Constraint Satisfaction: From Theory to Practice*, volume 117 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. 159–168.

Wilkins, D., and Myers, K. 1998. A multiagent planning architecture. In Simmons, R.; Veloso, M.; and Smith, S., eds., *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, 154–163. AAAI.

The Potted Plant Packing Problem

Towards a practical solution

René Schumann and Jan Behrens

OFFIS, Escherweg 2 26121 Oldenburg Germany
rene.schumann|jan.behrens@offis.de

Abstract. The potted plant packing problem was presented in [1] as a practical planning problem. The plants are packed on standardised trollies for transportation, with transport costs being directly dependent upon the number of trollies used. As a result, the effective packing of trollies is an important practical problem. Effective packing of trollies becomes even more important as this sector evolves into a consumer market.
This paper presents our further work towards a solution of this problem. We designed a framework algorithm and developed a prototype capable of solving a simplified version of the problem. The paper at hand focuses on the algorithms and the prototype that were developed so far.

1 Introduction

The transportation of plants is comparably expensive as they require careful treatment due to there sensitivity. For standardised transport, potted plants are loaded on transport trollies (shown in figure 1). The cost of transportation depends on the number of these trollies. In order to minimise transportation costs, effective packing of trollies is necessary. This becomes even more important with the ongoing shift towards DIY stores and discounters and their demand for smaller orders at shorter distribution intervals. Ongoing research at OFFIS also evaluates strategies for this sector [2].
This paper discusses the potted plant packing problem, which is a special 3D bin packing problem. It is a practical planning problem one of our costumers asked us to solve. The problem was first introduced in [1]. So far we worked on a solution that can be integrated into, or supplement the software of our costumer. The scientific challenge of this problem is not only finding a solution to the packing problem, but also overcoming the difficulties in formalising the problem description in an accepted representation. At first we will give a short summery of the problem statement, followed by a presentation of our steps towards a formalisation of the problem. Section 3 thereafter presents the methods developed so far. The prototype that bases on the implementation of these methods is then presented in section 4. Finally we discuss further extensions needed for practical use.

2 The packing problem

2.1 Problem statement

The chief task is computing a valid packing instruction for a given transportation order. Such a packing instruction must contain directives for the exact placement of each and every plant that is part of the order. Any such directive holds information to the plant's placement on its layer - and each layer's exact placement (mounting height) within the trolley. To clarify the problem, a trolley is shown in figure 1. A number of further constraints and additional rules have to be ob-

Fig. 1. trollies for plants, picture taken from [3]

served as part of the problem. For example; it is allowed to stack plants on a layer, the placement of layers into trollies has to respect the stability of trollies, and the total trolley height usually has to be less than the available internal truck height.

2.2 Towards a formal representation of the packing problem

A first step towards a formal description of the potted plant packing problem was proposed in [1]. The problem was presented as a 6-tuple according to the representation of scheduling problems presented in [4]. The 6-tuple consists of

- resources (trollies),
- objects (plants),
- order (defining the quantity of plants to pack),
- hard constraints (e.g. stability),
- soft constraints (contiguously placement of plants of an order item) and
- an objective function (minimising the needed number of trollies).

Another common description for packing problems was presented by Dyckhoff [5]. Dyckhoff points out a number of typical characteristics of cutting and packing problems. He advocates the usage of some of these characteristics to identify similar groups of cutting and packing problems. These characteristics are:

- dimensionality (1, 2, 3, n)

- kind of assignment (B, V)
- assortment of large objects (O, I, D)
- assort of small objects (F, M, R, C)

For a detailed discussion of this notation see [5] or [6]. Thus a problem description of a cutting or packing problem is a 4-tuple, describing the problem in respect of the mentioned characteristics.
Following Dyckhoff's description, the potted plant packing problem belongs to the class $3/V/D/R$. This notation encodes that,

- the problem has three relevant dimensions (length, width, height),
- all small objects (plants) have to be placed within a large object (trolley),
- that the large objects (trollies) can have different dimensions, (trolley with or without add-on modules),
- and that there are many small objects of relatively few different figures.

This typology is widely used to describe the main characteristics of cutting and packing problems. Currently an improved typology is discussed by [6]. According to [6] the here discussed problem can be seen as a special type of the Three Dimensional Multiple Bin Size Bin Packing Problem. A current survey about existing literature of packing and cutting problems can be found in [6], too.
The main disadvantage of these notations is that the problem can not be described entirely. Additional constraints have to be added to describe the actual problem. Nevertheless, these notations are widely in use because they allow a classification of such problems.

3 Towards a practical solution for the packing problem

As already mentioned, it is quit challenging to describe such a practical planning problem in terms of a formal model. Even more challenging is solving such a problem.
As already discussed in [1], the problem can be decomposed, thereby decreasing its complexity. Our decomposition consists of the following sub-problems:

- Distribution of plants on layers
- Distribution of layers on trollies

As a consequence of this decomposition a computed solution may not be optimal, but it can be computed faster due to the reduction of complexity. Both sub-problems will be discussed in more detail in the following subsections. But first we would like to introduce some techniques used to scale down the search space and optimise online computation time.

3.1 Scaling down the search space

In the context of this problem, the search space is defined as the number of possible valid packing instructions. The number of such valid packing instructions

is chiefly determined by the number of different plants. It should be obvious that the search space's size therefore mainly affects the distribution of plants on layers. As already stated in the problems formal representation in Dyckhoff's notation, there are a large number of small objects with differing dimensions. Currently we are dealing with at least 1,600 different articles. A diminution of the number of these potential pack able items - and thereby of the search space - is desirable. This should have a positive impact on overall performance, simplify the problem, and make it more tractable. To implement such a diminution, we decided to build categories of plants with similar dimensions. A category can be seen as a box or cylinder which can contain different plants with similar dimensions. Figure 2 illustrates this. Building categories necessitate a design decision

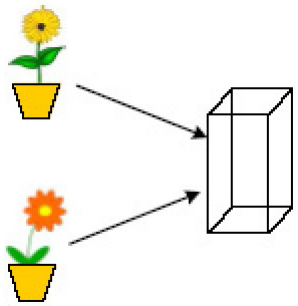


Fig. 2. concept of categories

concerning the number of different categories. Reducing the number of categories implies a decreasing precision, but also reduces the heterogeneity of figures to pack. This can best be illustrated when looking at the two extremes:

1. There exists only one category, resulting in a maximisation of wasted space and a minimisation of the number of categories.
2. Each group of plants with identical dimensions has its own category, resulting in a minimisation of wasted space and a maximisation of the number of categories.

As explained above, those extremes have a great impact on the search space's size. To find the optimal number of categories, we advocate the computation of the wasted space for a given number of categories and analysing the gradient of the resulting curve. The graph of such an analysis is shown in figure 3.

3.2 Reducing online computation time

To further reduce online computing time, we decided to use pre-computed packing patterns stored in a database. A packing pattern represents an entire packed

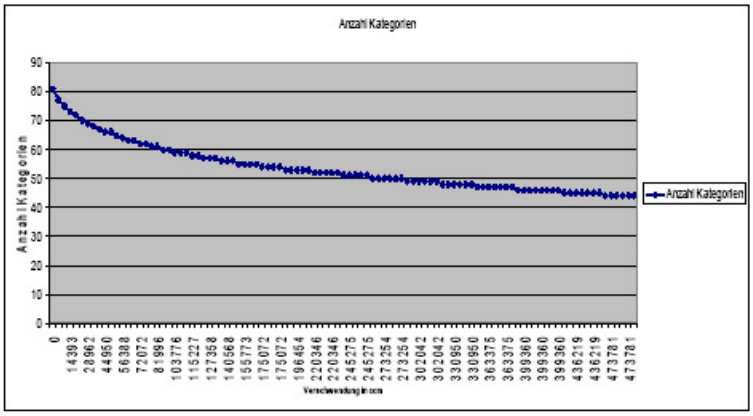


Fig. 3. x-axis: wasted space; y-axis: number of categories

layer. Such a packing pattern is optimal in such a sense that no additional plant of any category already placed thereon can be added. It has to be stated that such offline computations can be very time consuming. This is because of the possibly large number of categories and their potential combinations. Our experiments started with approximately 350 different plants, mapped into 59 categories. The computation of all possible packing patterns containing plants of three or less different categories would take several years[1] and would have to evaluate nearly 20 million layers. As a consequence, the complete offline computation of all possible packing patterns seems impractical. We therefore decided to compute offline patterns containing plants of one category only. To enable a self learning system and increase the number of packing patterns over time, we store all layers computed online into a database. Subsequent computations will thereby benefit from layers computed before and the patterns derived from them.

3.3 The packing algorithm's framework

This section describes the main planning process, which is an extended version of the algorithm presented by us in [1]. First, all plants of an order are put into queues, with a dedicated queue for all categories having the same height. Then all plants of the first none empty queue are added to the orders working set. If a packing pattern can be applied to a part of the items in the working set, a corresponding layer is introduced and those elements are removed from the working set. This is repeated until all queues are empty and no more packing pattern can be applied. If at this time the working set still contains further elements, additional layers have to be computed by the online layer packing

[1] Estimated 20 years time, based on experiments with our Java prototype run on a single 2GHz processor with 1GB RAM.

1. $Ts = \emptyset$ // trolley set
2. $Ws = \emptyset$ // working set of packing units
3. $Ls = \emptyset$ // layer set
4. $Ap = \emptyset$ // applicable packing pattern
5. FOR EACH $o \in O$ // where O is an order and o a order item
   - $P =$ packing-unit(o)
   - $q =$ queueForCategory(category($p \in P$)) //a Queue for each category
   - enqueue(q,p)
6. WHILE NOT($\exists q \in Q; |q| > 0$ OR $|Ap| > 0$)
   - IF ($|Ap| > 0$)
     - p = findBestPattern(Ap)
     - $Ls = Ls \cup$ newEbene(p)
     - $Ws = Ws -$ elementsOf(p)
   - ELSE
     - $Ws = Ws \cup$ deque($q; q \in Q, \forall q' \in Q : |q'| > 0, |q| > 0,$ h(q) $\geq$ h(q'))
   - $Ap =$ findapplicable(Ws)
7. IF $|Ws| > 0$
   - $Ls = Ls \cup$ onlineLayerComputations(Ws)
8. $Tr =$ distributeLayers(Ls)

Algorithm 1: packing frame algorithm

algorithm until all plants have been packed. Thereafter all computed layers are packed into trollies. The trollies and their layers then form the complete packing instruction.

3.4 Packing of layers

For our first prototype we simplified the problem described above in such a way that we only looked at plants distributed in round pots. Therefore the packing of layers corresponds to a relatively common packing problem; the problem of packing circles into a rectangular. In the notation of Dyckhoff the problem can then be formalised as $2/V/I/R$. Meaning that

- the problem has two relevant dimensions (length, width),
- all small objects (plants) have to be placed within the large objects (layers),
- large objects are identical in size,
- and there are many small objects with relatively few different dimensions.

The height of a plant can be ignored when only considering its placement as a circle in a rectangular. It has to be considered of course, as the dimension defining the layer's overall height. The overall height of a layer ($h(l)$) is indicated by the height of the tallest plant ($h(p)$) placed thereon. This can be stated as

$$h(l) = h(p_i); p_i \in elements(l); \forall p_j : p_j \in elements(l), h(p_i) \geq h(p_j).$$

The combined height of all layers has a strong influence on the packing instruction's quality of the second planning step. As a consequence it is desirable that

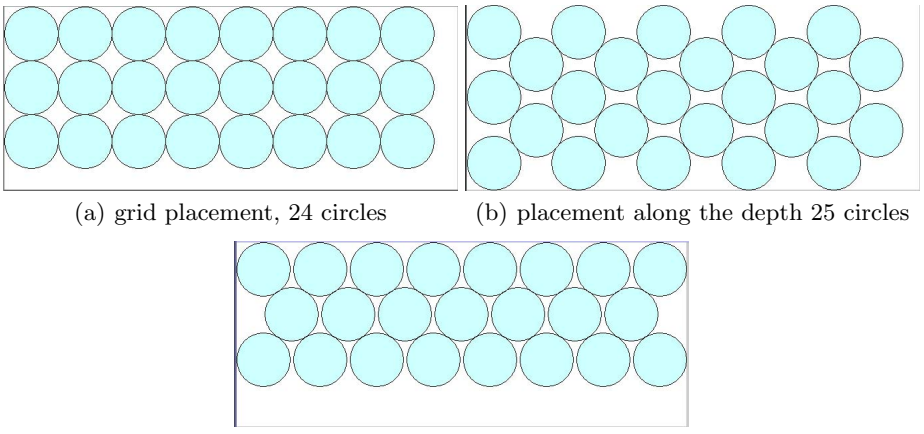
the summed height of all layers is minimal.

$$h(\mathcal{L}) = \sum_{1 \leq i \leq n} h(l_i)$$

To minimise $h(\mathcal{L})$, all plants within an order are sorted descending by their height. The sequence in which plants are added to the working set depends on this sorting. Because of the order in which plants are then added (and thus packed) to the working set, layers will generally contain plants of similar height. Thereby minimising the combined height of all layers.
The problem of packing circles into a rectangular can be divided into two sub-problems, namely the packing of circles of equal and unequal sizes. These cases differ in their complexity and are discussed in two separate sections.

Packing of equal circles into a rectangular Finding an optimal solution for the problem of packing equal circles into a rectangular is a NP-hard task, and in fact optimal solutions are only known for up to 20 circles (see [7] for instance). However, heuristics can be designed which compute solutions with sufficient quality very fast. These heuristics base on a regular placement of circles. We implemented three regular placement strategies, namely

- grid placement
- placement along the depth
- placement along the width



(a) grid placement, 24 circles (b) placement along the depth 25 circles

(c) placement along the width, 23 circles

Fig. 4. regular circle placements

As one can see in figure 4 the number of placements can differ depending on the strategy. So far we can not predict which of these will offer the best performance

for a given circle and rectangular size. Since these heuristics are very fast, we therefore always compute all three alternatives and then choose the best.

Packing of unequal circles into a rectangular The placement of circles of differing sizes into a rectangular is more difficult. Only few publications have yet dealt with this problem, for example [8], [9], [10] and [11].
We implemented the solutions proposed by [9] and [11]. Our experiments showed that the solution quality of the simulated annealing approach by [9] was not acceptable in our context. We were not able to compute solutions of a quality similar to that presented by the authors. This is most likely due to an erroneous implementation on our side. As a consequence we implemented the maximum hole degree algorithm (B1.0) presented in [11]. The main idea of this algorithm is the subsequent placement of circles into corners. A corner can is defined by two sides of the rectangular, a rectangular side and a circle, or two circles. The first two circles are placed by a simple placement strategy. Then for each circle not placed already within the rectangular, all possible corner placements are computed. The circle being associated with the placement having the minimal distance to another circle or side is then chosen. This is repeated until no more valid corners are found or all elements have been placed. A detailed description of the algorithm as well as a complexity analysis can be found in [11]. We are pleased with the performances of this algorithm, expect for such cases where there is a large number of circles (50 and above). In such cases the number of possible corner placements grows very fast, therefore drastically increasing computation time. An example of a layer containing two different types of circles is shown in figure 5.

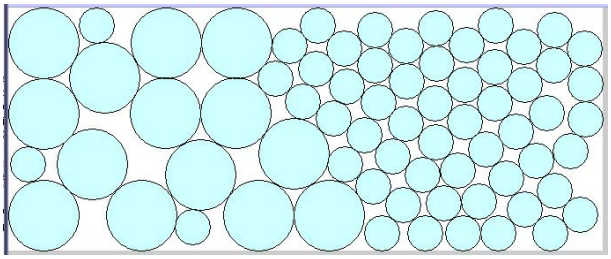


Fig. 5. placement computed by the maximum hole degree algorithm

3.5 Trolley packing

The trolley packing algorithm has to answer two key questions.

1. Which layers are mounted into which trolley?
2. Where is each layer placed within the trolley?

The first question concerning the distribution of layers onto trollies is a classical bin packing problem. This can be solved using standard algorithms like next fit or best fit.
To answer the second question, we have to compute each layer's correct mounting point. This has to be done while observing the height and weight constraints of each layer as well as the trolley's layout and the maximum height allowed during transport. Figure 6 illustrates the typical trolley layout. Layers are hooked into mounting points, which are generally found at 5 cm intervals from a base of 20 cm up to a height of 190 cm. The available height can be increased through the use of add-on modules up to a total of 225 cm. The placement of layers within a trolley follows a simple strategy:
The tallest layer is hooked into the topmost mounting point that still ensures the adherence of all other constraints, especially the maximum allowed height. All remaining layers are then sorted in ascending order by their weight and inserted top to bottom into the trolley. This strategy aims at two goals. It tries to

- maximise the usage of available space on the truck and
- lower the centre of gravity to the nethermost point for stability reasons.

Due to the use of add-on modules the layout of a trolley might change, this requires no change in the method of computation of the layer placement however.

4 The implemented prototype

The strategies and algorithms explained above have been implemented to a large extend in a prototype. As has partly been mentioned in the section 3.4, the known main limitations of the prototype are that;

- only plants potted in round pots are considered (this is the majority),
- no stacking of plants is allowed,
- and the stored packing patterns only allow for circles of equal size.

The prototype has been implemented using the Java language and makes use of the eclipse rich client platform. A screenshot is shown in figure 6. The prototype allows opening of multiple orders, trollies, and layers. It provides a tree view for each order, containing a branch for each trolley and hereunder a leaf for each layer within that trolley. To compute a new packing instruction, the user has to specify a valid order and then start the computation. The prototype is currently (May 2006) undergoing field testing and is expected to be developed into a 1.0 release afterwards.

5 Future work

Our main focus is the further improvement of the computed solutions and the generation of competitive packing instructions. A main advance in that respect - and the next step planned - will be the implementation of plant stacking. To

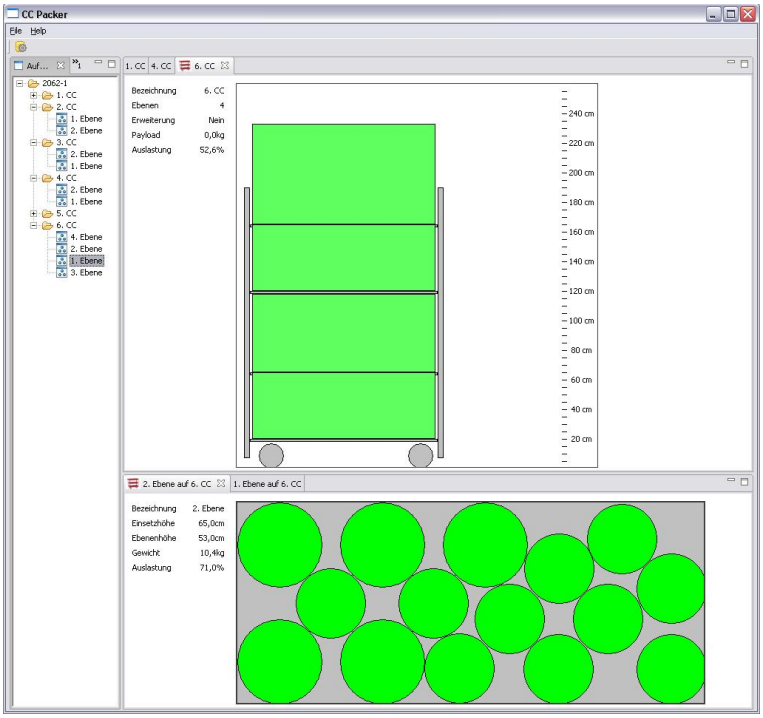


Fig. 6. Screenshot showing a packed trolley

avoid damage, this requires information on the stackability of each plant as well as its trunk diameter (see figure 7). The existing algorithms will have to be extended to permit for stacking of such plants that allow it. Packing algorithms addressing stacking of circles and respecting non-stackable areas have, to the best knowledge of the authors, not been addressed in literature so far.
Once stacking will be implemented in our packing algorithms, we are excepting to compute layers with a much higher quality from a practical perspective. We then plan to evaluate our solution. Because of the complexity of the problem an evaluation can only be done by comparison of computed packing instructions with real packing data gathered on site.

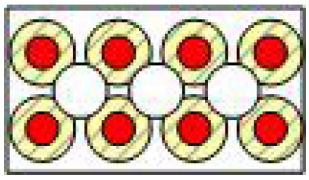


Fig. 7. stacked plants, non-stackable areas marked red

As has been mentioned, offline computed packing patterns are used to improve the computation of packing instructions. We assume that there are going to be a lot of different packing patterns of different quality, relevance, and practical usability in terms of frequency of usage. To ensure that the time needed for pattern retrieval stays in acceptable boundaries, we assume that it is necessary to cache frequent used patterns. Therefore bookkeeping information about the usage and quality of packing pattern is believed to be necessary and will also be a field of future work.
The project's aim is a software system with a high degree of integration into our customers systems. To offer him more flexibility it is desirable that he can modify stored packing patterns manually through a graphical editor.
Most of the plants relevant in our context are potted in round pots. However, there are exceptions that will ask for further extension or new implementations of the packing algorithm to enable their handling as well. A subset of this problem will also be the consideration of trays, which have a rectangular shape and can hold a fixed number of plants. Current publications mainly focuses on the packing of either rectangular or circles into a larger rectangular (see [12] for packing rectangular and e.g. [11] for packing circles). The authors know of no publication addressing an integrated approach for both, circles and rectangles.
So far we assume that the dimensions of the packed plants are static and inflexible. This is a simplification and in fact not true for live plants as their dimensions change during the season. Further research in this field could start with the work presented in [13] and then try to evaluate whether a more realistic model (including dynamic plant dimensions) would indeed improve quality of the instructions computed.

References

1. Schumann, R., Behrens, J., Siemer, J.: The potted plant packing problem. In Sauer, J., ed.: 19. Workshop Plan, Scheduling und Konfigurieren / Entwerfen (PUK). Fachberichte Informatik, Koblenz, Universität Koblenz-Landau Fachbereich Informatik (2005)
2. OFFIS: AmmLog, http://www.ammlog.de. (2006) accessable on 21.04.06.
3. Foko Lübsen und Sohn Internationale Spedition: Hompage Focko Lüpsen & Sohn GmbH, http://www.luepsen.de/seite01e.htm. (2006) accessable on 21.04.06.
4. Sauer, J.: Wissensbasiertes Lösen von Ablaufplanungsproblemen durch explizite Heuristiken. DISKI 37. Infix Verlag (1993)
5. Dyckhoff, H.: A typology of cutting and packing problems. European Journal of Operational Research (EJOR) **44** (1990) 145 – 159
6. Wäscher, G., Haußner, H., Schumann, H.: An improved typology of cutting and packing problems. Working paper no. 24, (Otto von Guericke Universität Magdeburg) Revision 2006-01-06.
7. Erich Friedman: Erich's Place, http://www.stetson.edu/ efriedma/cirinsqu/. (2006) accessable on 19.04.06.
8. George, J.A., George, J.M., Lamar, B.W.: Packing different sized circles into a rectangular container. European Journal of Operational Research (EJOR) **84** (1995) 693 – 712

9. Correia, M.H., Oliveira, J.F., Ferreira, J.S.: Cylinder packing by simulated annealing. Pesquisa Operacional **20** (2000) 269–286
10. Schöning, U., Toran, J., Thierauf, T., Messner, J., Blubeck, U.: Three algorithms for packing variable size bobbins. Report, Abt. Theoretische Informatik Universität Ulm (2002)
11. Huang, W.Q., Li, Y., Akeb, H., Li, C.M.: Greedy algorithms for packing unequal circles into a rectangular container. European Journal of Operational Research (EJOR) **56** (2005) 539 – 548
12. Scheithauer, G., Terno, J.: The G4-Heuristic for the Pallet Loading Problem. Journal of the Operational Research Society (JORS) **47** (1996) 511 – 522
13. Albrecht, A., Cheung, S.K., Hui, K.C., Leung, K.S., Wong, C.K.: Optimal Placements of Flexible Objects Part II: A Simulated Annealing Approach for the Bounded Case. IEEE Transaction on Computers **46** (1997) 905 – 929