Sekitei: An AI planner for Constrained Component Deployment in Wide-Area Networks

Technical report TR2004-851

Tatiana Kichkaylo, Anca Ivan, and Vijay Karamcheti
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University, New York, NY 10012
{*kichkay,ivan,vijayk*}*@cs.nyu.edu*

March 1, 2004

Abstract

Wide-area network applications are increasingly being built using component-based models, enabling integration of diverse functionality in modules distributed across the network. In such models, dynamic component selection and deployment enables an application to flexibly adapt to changing client and network characteristics, achieve load-balancing, and satisfy QoS requirements. Unfortunately, the problem of finding a valid component deployment is hard because one needs to decide on the set of components while satisfying various constraints resulting from application semantic requirements, network resource limitations, and interactions between the two.

In this paper, we describe a general model for the component placement problem and present an algorithm for it, which is based on AI planning algorithms. We validate the effectiveness of our algorithm by demonstrating its scalability with respect to network size and number of components in the context of deployments generated for two example applications – a security-sensitive mail service, and a webcast service – in a variety of network environments.

1 Introduction

The explosive growth of the Internet and the development of new networking technologies has been accompanied by a trend favoring the use of component-based models for construction of wide-area network applications. This trend, exemplified in grid frameworks such as Globus [10] and more recently OGSA [11], as well as component frameworks such as CORBA [31], J2EE [37], and .NET [29], enables the construction of applications by integrating functionality embodied in components possibly running across multiple administrative domains. Although most such frameworks have traditionally relied upon a static model of component linkages, a growing number of approaches (e.g., Active Frames [27], Eager Handlers [39], Active Streams [6], Ninja [36], CANS [15], Smock [17], Conductor [35], and recent work on Globus [12]) have advocated a more dynamic model, where the selection of components that make up the application and their location in the network ("deployment") are decisions that are deferred to run time.

Dynamic component-based frameworks allow distributed applications to flexibly and dynamically adapt to variations in both resource availability and client demand. For example, a security-sensitive application may wish to trade-off concerns of security and efficiency depending on whether or not its execution environment consists of trusted nodes and links. Similarly, an application that relies on high-bandwidth interactions between its components may wish to change the quality of service provided to the client when the available bandwidth on a link drops or the application is accessed by a resource-limited client. Dynamic frameworks enable adaptation to the above changes by deploying application-aware components that can achieve load-balancing, satisfy client QoS requirements (e.g., by transcoding), and enable higher throughput (by replicating appropriate components), in essence customizing the application to its resource and usage conditions.

The benefits of dynamic component frameworks are fully realizable only if components are automatically deployed in response to dynamic changes in network conditions. To enable this, most such approaches rely on three elements: (i) a *declarative specification* of the application, (ii) a *trigger* module, and (iii) a *planning* module. The *trigger* module monitors application behavior and network conditions and chooses the moments *when* adaptation is required. The *planning* module makes decisions on *how* to adapt, by selecting and deploying components in the network to best satisfy application requirements as dictated by the *declarative specification*. This paper focuses on the planning aspect.

In general, the planning problem in dynamic frameworks is complicated by the fact that to compute a valid deployment, one needs to (i) decide on a set of components, and (ii) place these components on network nodes in the presence of application (type) constraints (e.g., linked components should consume each other's outputs), resource constraints (e.g. node CPU capacity and link bandwidth), and interactions between the two (e.g., an insecure link might affect the security characteristics of application data). The need to simultaneously achieve both these goals makes the planning problem computationally harder than traditional mapping and optimization problems in parallel and distributed systems, which tend to focus on a subset of the concerns of requirement (ii) above. This complexity is also the reason that existing dynamic frameworks have either completely ignored the planning problem [27, 39, 6], or have addressed only a very limited case [36, 15, 17, 35, 12].

This paper addresses this shortcoming by presenting a model for the general planning problem, referred to as the Component Placement Problem (CPP), and describing an algorithm for solving it. The model aims for expressiveness: component behavior is modeled in terms of implemented and required interfaces [17], and application, resource, and their interaction constraints are all represented using arbitrary monotonic functions. Our algorithm for solving the CPP, called Sekitei, leverages several decades of research on planning techniques developed by the Artificial Intelligence (AI) community. Sekitei overcomes the scalability restrictions of state-of-the-art AI planning techniques (e.g., RIPP [24]) by exploiting the specific characteristics of CPP. The Sekitei planner has been implemented in Java as a pluggable module to allow its use in several component-based frameworks. We report on its use to generate deployments for two example applications – a security-sensitive mail service, and a webcast service – in a variety of network environments. Our results validate the scalability of the algorithm, both with respect to the network size and the number of application components.

The rest of this paper is structured as follows. Section 2 discusses existing approaches to the component placement problem and overviews AI planning techniques. In Section 3 we introduce two example applications that are used to illustrate our techniques. Section 4 describes our model of the CPP. Section 5 gives details on compilation of the CPP into a planning problem. Section 6 describes the Sekitei algorithm and its extensions. Section 7 evaluates the performance of the algorithm. We conclude with a discussion of future work.

2 Related work

2.1 Component-based frameworks

From a planning point of view, there are two classes of dynamic component-based frameworks: (i) systems that assume the existence of an external planner (Active Frames [27], Eager Handlers [39], Active Streams [6]), and (ii) systems that implement their own planner (GARA [12], Ninja [36], CANS [15], PSF [17], and Conductor [35]).

The second class can be further divided into two subclasses. The first subclass includes systems such as GARA (Globus Architecture for Reservation and Allocation) [12], the planning module in the Globus [10] architecture, which assumes a pre-established relationship between application tasks to deploy them to minimize resource consumption. GARA supports resource discovery and selection (based on attribute matches), and allows advance reservation for resources like CPU, memory, and bandwidth. However, it does not consider application specific properties, such as that some interactions need to be secure.[1]

The second subclass of planners both select and deploy a subset of components, while satisfying application and network constraints. Systems such as Ninja [36], CANS [15], and Conductor [35], all of which enable the deployment of appropriate transcoding components along the network path between weak clients and servers, simplify the assumptions of the planning problem to perform directed search. The Ninja planning module focuses on choosing already existing instances of multiple input/output components in the network so as to satisfy functional and resource requirements on component deployment. Conductor restricts itself to single input, single output components, focusing on satisfying resource constraints. CANS adopts similar component restrictions, but can handle constraints imposed by the interactions between application components and network resources, and additionally can efficiently plan for a range of optimization criteria. For example, the CANS planner [14] can ensure that node and link capacities along the path are not exceeded by deployed components, while simultaneously optimizing an application metric of interest (e.g., response time).

[1] Globus sets up secure connections between application components, thereby satisfying this particular constraint. However, there is no general mechanism to specify component properties that are affected by the environment.

More general are systems such as Partitionable Services Framework (PSF) [17], which permit network services to be constructed as a flexible assembly of smaller components, permitting customization and adaptation to network and usage situations. The PSF planner works with very general component and network descriptions: components can implement and require multiple interfaces (these define "ports" for linkages), can specify resource restrictions, and additionally impose deployment limitations based on application-dependent properties (e.g. privacy of an interface). This generality comes at a cost: the orginal PSF planning module performed exhaustive search to infer a valid deployment. The work described in this paper grew out a desire to remedy this situation.

2.2 Planning and scheduling

The component placement problem closely resembles problems studied in the AI planning and scheduling community. It requires performing dependency-driven choice, which is the focus of planning, and satisfying resource constraints, which is closely related to scheduling. This section provides an overview of the most relevant efforts.

In classic AI planning, the world is represented by a set of boolean variables, and a world state is a truth assignment to these variables. The system is described by a set of possible *operators*, i.e., atomic actions that can change the world state. Each operator has a precondition expressed by a logical formula and a set of effects (new truth assignments to variables of the world state). An operator is applicable in a world state if its precondition evaluates to true in that state. The result of an operator application is to change the world state as described by the operator's effects. A planning problem is defined by a description of the operator set, an initial state (complete truth assignment to all variables), and a goal (logical formula). The planner finds a sequence of applicable operators that, when executed from the initial state, brings the system to a state in which the goal formula evaluates to true.

Classic planners perform directed search in the space of situations or partial plans and can be divided into four classes based on their search method: regression planners (e.g., Unpop [28], HSPr [5]) search from the goals, progression planners (e.g., GraphPlan [2], IPP [25]) start from the initial state, causal-link planners (e.g., UCPOP [34]) perform means-ends analysis, and compilation-based planners (e.g., SATPLAN [18], ILP-PLAN [20], BlackBox [19], GP-CSP [9]) reduce the planning problem to a satisfiability or optimization problem, e.g. integer linear programming. Some planners, e.g. BlackBox, use a combination of the above techniques to improve performance. McDermott [28] suggests extending regression planners using progression techniques; however, we are not aware of any implementation of this idea.

Adding resource constraints to a planning or scheduling problem tremendously increases its complexity [16]. For this reason, most planning systems have restricted themselves to only simple resource expressions. Most existing resource planners (e.g., RIPP [24], LPSAT [38], ILP-PLAN [20]) limit themselves to linear expressions in preconditions and effects. Zeno [33] can accept more complicated expressions, but delays their processing until variable bindings linearize the expressions.

Scheduling solutions have accommodated more general resource expressions, given their focus on finding the best (according to some metric) sequence of actions subject to various constraints. Note that although the problems of planning and scheduling are nominally different (the planning focuses on the *choice* of actions, while scheduling focuses on *ordering*), one can extend the tradeoffs and techniques for dealing with resource constraints from one domain into the other.

The algorithms in [26] and [30] describe computation of *resource envelopes* for scheduling problems with constant changes of resource levels. Both resource envelopes and temporal networks [8] use graph-theoretic algorithms to prune the search space, and are able to produce generalized optimal schedules.

Scheduling systems that need to support complex resource functions discretize resources to decrease the search space and use heuristic search to find a *good* (suboptimal) solution. For example, the algorithm described in [13] uses forward chaining to cope with sequence-dependent (i.e. non-reversible) resource functions. Pegasus [4] relies on external modules for resource-dependent decisions.

Sekitei, the algorithm described in this paper, builds upon several of the planning an scheduling techniques described above, particularly to deal with the presence of non-reversible resource functions. However, a challenge it needs to overcome is the scalability limitations of classical planning approaches. Sekitei addresses the latter issue by exploiting the structured nature of the component placement problem to introduce optimizations not possible in a general AI planner.

3 Example applications

From the perspective of Sekitei, applications are viewed as sets of components interacting with each other by sending data streams (referred to as interfaces) over network links. Components specify their logical and resource requirements. These requirements effectively represent the (possibly infinite) set of possible application configurations. The purpose of the planner is to choose a (minimal) application configuration that satisfies resource constraints.

In this paper we use two component-based applications to illustrate our algorithm.

3.1 Mail application

The first application is a component-based security-sensitive mail service, originally introduced in [17]. The mail service provides expected functionality — user accounts, folders, contact lists, and the ability to send and receive e-mail. In addition, it allows a user to associate a trust level with each message depending on its sender or recipient. A message is encrypted according to the sender's sensitivity and sent to the mail server, which transforms the ciphertext into a valid encryption corresponding to the receiver's sensitivity and saves the new ciphertext into the receiver's account. The encryption/decryption keys are generated when the user first subscribes to the service.

The mail service is constructed by flexibly assembling the following components: (i) a `MailServer` that manages e-mail accounts, (ii) `MailClient` components of differing capabilities, (iii) `ViewMailServer` components that replicate the `MailServer` as desired, and (iv) `Encryptor`/`Decryptor` components that ensure confidentiality of interactions between the other components. These components allow the mail application to be deployed in different environments. If the environment is secure and has high available bandwidth, the `MailClient` can be directly linked to the `MailServer`. The existence of insecure links and nodes triggers deployment of an `Encryptor`/`Decryptor` pair to protect message privacy. Similarly, the `ViewMailServer` can serve as a cache to overcome links with low available bandwidth.

Figure 1 shows the abstract structure of the mail application, which describes all possible configurations of the application. Rectangles correspond to component types, ovals represent interfaces (types of data streams). Since each component consumes at most one data stream, all legitimate configurations of this application are chains. However, it is possible to have more than one instance of the same component type in such a chain.

Figure 2 illustrates a simple scenario where the `MailClient` can be deployed on node 0 only if connected to a `MailServer` through a `ViewMailServer`. Directly linking the `MailClient` to the `MailServer` is not possible because the link between them does not have enough available bandwidth to satisfy the `MailClient` requirements. Satisfying the requirements implicit in this scenario automatically needs both a better specification of application requirements and a planning module to generate the deployment.

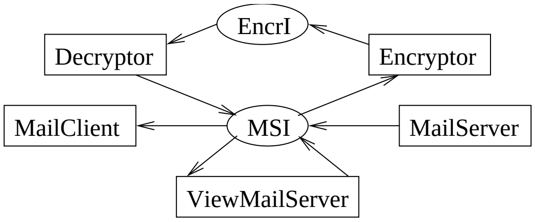


Figure 1: Abstract structure of the mail application. Rectangles represent components, and ovals represent interfaces.

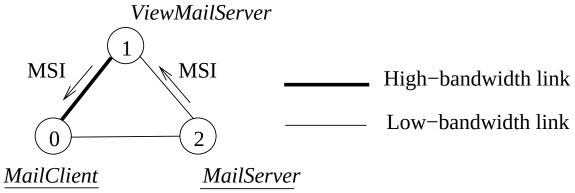


Figure 2: Component deployment of the mail application

3.2 Webcast application

The second application models a webcast scenario (Figure 3), where the server provides a combined media stream consisting of images and text, which needs to be delivered to the client. The client issues requests at a particular rate, which translates into a minimum bandwidth requirement. If the network between the client and the server has stable high bandwidth, a direct connection is made. However, in more resource-restricted situations additional components might be injected into the network: Figure 3 shows an example of such injection involving `Splitter`, `Merger`, and compression components (`Zip`and `Unzip`). Similarly, a `Filter` component may be injected to change parameters of the image stream, such as the color depth. In the example shown in the figure, the network consists of two high-bandwidth LANs with a low bandwidth link between them. The `Server` located on node 7 produces a media stream, and the `Client` on node 0 wants to consume this stream with a particular request rate. This goal is achieved by splitting the media stream (M) into text (T) and image (I) components, zipping the text portion of the stream, so that the combined I+Z bandwidth is less than that of the original M stream, sending the I and Z streams to the client LAN, and performing the reverse transformations there.

Figure 4 describes the abstract structure of the webcast application. Since the splitter component produces and the merger component requires two interfaces, some configurations of the webcast application might have a DAG

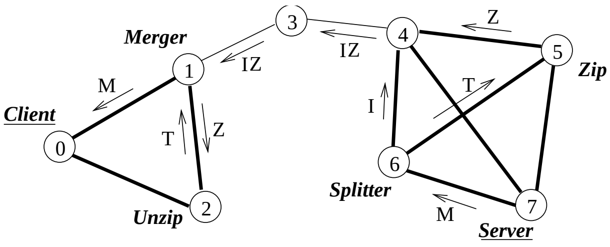


Figure 3: The webcast application

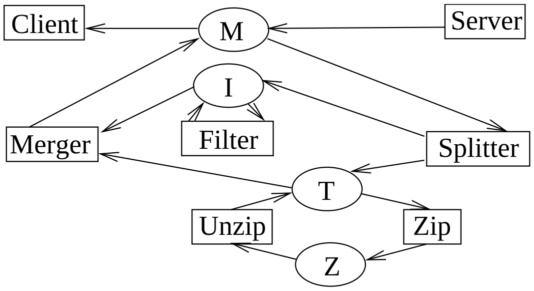


Figure 4: Abstract structure of the webcast application. Rectangles represent components, and ovals represent interfaces.

structure.

4 Model of the CPP

Many systems solve the Component Placement Problem (CPP) in one form or another. However, the specific formulation differs along one or more of the following dimensions: mobility (fixed locations in Ninja [36] vs. arbitrary deployments), arity (single input - single output components in CANS [15] vs. arbitrary arity), support for resource constraints, etc. As one of the contributions of this paper, we present a general model for the CPP that unifies different variations of this problem and enables use of the same planning algorithm in various component-based frameworks.

Formally, the CPP is defined by the following five elements: (i) the network topology, (ii) the application framework, (iii) the component deployment behavior, (iv) the link crossing behavior, and (v) the goal of the CPP.

4.1 Network topology

The network topology is described by a set of nodes and a set of links. Each node and link has tuples of static and dynamic properties associated with it. The dynamic properties are non-negative real values that can be changed, e.g. node CPU, link bandwidth. The static properties are assumed fixed during the life time of an application. Static properties might be represented by boolean values or real intervals, e.g. security of a link or trust level of a node.

4.2 Application framework

The application is defined by sets of interface types and component types, similar to the Corba Component Model [32] and object-oriented languages such as Java. Each component type specifies sets of *implemented* and *required* interfaces:[2] the former describe component functionality, while the latter indicate services needed by the component for correct execution. In addition, each interface is characterized by a set of component-specific *properties*. From the planning point of view, properties are defined as functions of other properties and have no semantics attached to them.

In general, applications can propagate properties either (i) from required to implemented interfaces – *publish-subscribe* applications, or (ii) from implemented to required interfaces – *request-reply* applications. In *publish-subscribe* applications, servers send data streams to clients. In *request-reply* applications, clients make requests to servers and servers send back replies. Although the planner can work with both types of applications, our description of the planning algorithm focuses on request-reply applications.

Figure 5 shows a partial specification of the `ViewMailServer` component of the mail application described in Section 3.1. This component implements and requires `MailServerInterface`. `MailServerInterface` is associated with both application-specific and application-independent properties. Application-specific properties include the trust level (`Trust`) and message security (`Sec`), which indicate, respectively, the maximum message sensitivity level and whether or not the interface preserves message confidentiality. Application-independent properties include the number of incoming requests (`NumReq`), the maximum response size for a request (`ReqSize`), the request reduction factor (`RRF`), the amount of CPU consumed to process each incoming request (`ReqCPU`), and the maximum number of requests that can be processed by the component (`MaxReq`). The `RRF` attribute gives the ratio of requests sent to required interfaces in response to requests on the implemented interfaces. The use of these application-independent properties is described below.

[2] The counterpart for these concepts in a statically-linked Java/RMI application is as follows: implemented interfaces are identical to their namesake, while required interfaces correspond to remote references.

<**Component** name = VMS >
<**Linkages**>
<**Implements**>
<**Interface** name = MSI^i >
<**Properties**>
$MSI^i.Trust - derived$
$MSI^i.Sec - derived$
$MSI^i.NumReq - derived$
$MSI^i.ReqSize - derived$
$MSI^i.RRF := 10$
$MSI^i.ReqCPU := 2$
$MSI^i.MaxReq := 100$
<**Requires**>
<**Interface** name = MSI^r >

<**Conditions**>
$Node.NodeCPU \geq (MSI^i.NumReq * MSI^i.ReqCPU)$
$MSI^r.NumReq \geq (MSI^i.NumReq * MSI^i.RRF)$
$MSI^i.NumReq \leq MSI^i.MaxReq$
$MSI^r.Sec = True$
$MSI^r.Trust \geq 5$

<**Effects**>
$MSI^i.Sec := True$
$MSI^i.Trust := Node.Trust$
$MSI^i.ReqSize := 1000$
$MSI^i.NumReq := MIN(MSI^r.NumReq/MSI^i.RRF,$
$MSI^i.MaxReq, Node.NodeCPU/MSI^i.ReqCPU)$
$Node.NodeCPU := Node.NodeCPU -$
$MSI^i.NumReq * MSI^i.ReqCPU$

<**Interface** name = MSI >
<**Crosslink**>
$MSI^d.Sec := MSI^o.Sec\ AND\ Link.Sec$
$Link.BW := Link.BW -$
$MIN(Link.BW, MSI^o.NumReq * MSI^o.ReqSize)$
$MSI^d.NumReq :=$
$MIN(MSI^o.NumReq, Link.BW/MSI^o.ReqSize)$
$MSI^d.ReqSize := MSI^o.ReqSize$

VMS = `ViewMailServer`,
MSI = `MailServerInterface`
Superscripts r and i indicate required and implemented interfaces, o and d correspond to interfaces at link origin and destination.

Figure 5: Component/Interface descriptions.

4.3 Component deployment behavior

A component can be deployed on a node only if the required interfaces are present on the node, and the resource and property constraints of component deployment are satisfied on that node (e.g., that there is sufficient available memory or that the node has an appropriate version of the operating system). After deployment, the implemented interfaces become available on the node and the dynamic properties of the node are altered. The Sekitei planner can find a plan that satisfies both the application-specific and application-independent constraints. The former are expected to be supplied by the programmer. To simplify the task of writing application-independent constraints, we have introduced a small set of properties: `NumReq`, `MaxReq`, `RRF`, `ReqCPU`, described in the previous section. Figure 5 shows how these properties can be used to capture component resource consumption. The conditions associated with `ViewMailServer` specify that (i) the node should have enough capacity to serve incoming requests, (ii) the number of incoming requests should not exceed a certain maximum, and (iii) the component should be able to forward the `RRF` portion of requests to the required interfaces. The effects of deploying the `ViewMailServer` component are to decrease the node's CPU capacity and constrain the number of requests to the implemented interface.

4.4 Link crossing behavior

The link crossing behavior is described by interface specific functions. For each interface type, these functions describe how the interface properties are affected by the link properties when crossing the link, and how dynamic properties of the link are changed as a result of this operation. For example (see Figure 5), the security (used here to denote privacy attributes) of an interface after link crossing can be computed as a conjunction of the security of the interface at the source and the security of the link; the link bandwidth after the link crossing is the original bandwidth minus the consumed bandwidth, which is the smaller of the original bandwidth and the total bandwidth requirement of processed requests.

4.5 CPP goal

In the simplest case, the goal is to put a component of a given type onto a given node. For example, the goal in Figure 2 is to place `MailClient` on node 0. Other goals can include, for example, delivering a particular set of interfaces to a given node; this can be useful for repairing deployments when network resource availability changes.

The above model of the CPP is very flexible and allows the expression of a variety of application properties and requirements. In particular, most models we have found in literature can be captured in our formalism.

5 CPP as a planning problem

The CPP can be viewed as an AI planning problem with resource constraints:

- The state of the system is described by the availability of interfaces on nodes and placement of components on nodes. This information is described by a set of propositional (boolean) variables.
- Properties of nodes, links, and interfaces on nodes are described by real-valued resource variables.
- Operators correspond to placing a component on a node and sending an interface over a link.
- The CPP goal is translated into a propositional goal of having a component placed on a node.

Figure 6 describes the general structure of the system. The compiler module transforms a framework-specific representation of the CPP into an AI-style planning problem, which can be solved by the planner. The decompiler performs the reverse transformation, converting the AI-style solution into a framework specific deployment plan.

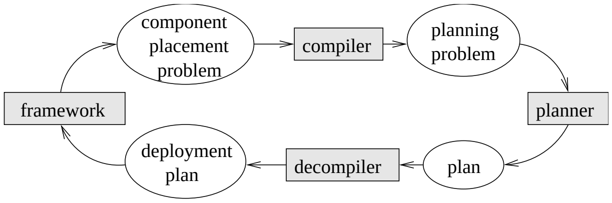


Figure 6: Process flow graph for solving CPP.

The rest of this section describes compilation and decompilation modules.

5.1 Compiling CPP into a planning problem

The state of the world in CPP is described by the network topology, the existence of interfaces on nodes, and the availability of resources. This information is mapped by the compiler into propositional and resource variables. For example, the fact that `MailServerInterface` is available on node `0` is represented by proposition `avMSI(0)`, and the amount of available CPU on node 1 by a real-valued resource variable `cpu(1)`.

Compilation of the CPP into a planning problem generates two operators: `pl`<*component*>`(?n)`[3] places a component on a node, and `cr`<*interface*>`(?n1,?n2)` sends an interface across a link.

An operator schema (parameterized operator) has the following sections (line numbers refer to the code fragment below):

- logical precondition of the operator, i.e., a set of propositions (boolean variables) that need to be true for the operator to be applicable (line 2);
- resource preconditions described by arbitrary functions that return boolean values (line 3-6);[4]
- logical effects, i.e., a set of propositions made true by an application of the operator (line 7);
- resource effects represented by a set of assignments to resource variables (lines 8-16).[4]

For example, the following schema describes the placement of the `ViewMailServer` (VMS) component on a node. The preconditions result from the conditions in Figure 5 and the fact that `MailServerInterface` (MSI) is a required interface. The effects come from the effects section of Figure 5, with `MaxReq` providing the upper bound on the `NumReq` parameter of the implemented interface.

```
1 plVMS(?n: node)
2  PRE: avMSI(?n)
3      cpu(?n) > MSIMaxReq*MSIReqCPU
4      numReq(MSI,?n)>MSIMaxReq*MSIRRF
5      sec(MSI, ?n) = True
6      trust(MSI, ?n) > 5
7  EFF: avMSI(?n), plVMS(?n)
8      numReq(MSI, ?n):=
9        MIN(numReq(MSI, ?n) / MSIRRF,
10         MSIMaxReq,
11         cpu(?n) / MSIReqCPU)
12     cpu(?n):=cpu(?n) -
13       numReq(MSI, ?n)*MSIRRF/MSIReqCPU
14     sec(MSI, ?n):=True
15     trust(MSI, ?n):=ntrust(?n)
16     reqSize(MSI, ?n):=1000
```

Given the operator definition above, the compilation of the CPP into a planning problem is straightforward. For each of the component types, the compiler generates an operator schema for a placement operator. In addition, an operator for link crossing is generated for each interface type. The initial state is created based on the properties of the network. The goal of the CPP is translated into a boolean goal of the planning problem.

5.2 Decompilation

The plan is a sequence of grounded (variable-free) instances of `pl`<*component*> (<*node*>) and `cr`<*interface*> (<*from*>, <*to*>) operators. In addition, information about logical support is easily

[3] Identifiers prefixed with a question mark denote variables.

[4] Sekitei currently does not support formulae involving parameters of implemented interfaces, and instead generates a conservative solution by using upper bounds on the values of such parameters.

extractable from the plan. For example, the fact that operator crMSI(1,0) depends on proposition avMSI(1) produced by operator plVMS(1) means that a ViewMailServer component needs to be placed on node 1 to produce the MailServerInterface before this interface is sent over the link to node 0 (Figure 2). This information can also be represented as a framework-specific deployment plan, which consists of (*component*, *node*) pairs and linkage directives, e.g. (VMS,1,MSI,MC,0) (send the MailServerInterface implemented by the ViewMailServer component located on node 1 to the MailClient component on node 0).

6 The Sekitei algorithm

Sekitei needs to deal with two problems not traditionally addressed by AI planning algorithms: the scale of the problem specification and non-reversibility of resource functions. These two problems are closely related, because the existence of resource preconditions and effects is the reason why existing preprocessing methods popular in state-of-the-art AI planning are unable to remove irrelevant operators from the problem specification.

Sekitei addresses these problems by combining regression and progression techniques and using layers of relaxed problems to prune the search space.

Section 6.1 describes the core algorithm as a sequence of layers solving relaxed versions of the original problem. Since it is easier to find a solution to a relaxed problem than to the original one, the layers of relaxed problems are used to prune the search space. Each new layer takes into account more restrictions present in the original problem, but needs to consider smaller sets of operators and variables.

Section 6.2 introduces a notion of resource maps. Resource maps are used to represent possibly achievable values of resource variables (resource envelopes). Section 6.3 discusses issues involved in implementation of resource maps, and presents a version of the algorithm which improves performance of the basic algorithm on the CPP.

6.1 The core algorithm

The first issue that an efficient planner for the CPP needs to address is the size of the problem. A problem instance can include hundreds of nodes and dozens of component types, which translate into component placement and link crossing operators. However, most of these operators will not be used in the shortest plan that achieves the goal. Standard preprocessing techniques [2], which rely on reachability analysis, do not remove these operators, because an operator can be included in some (long) sequence leading from the initial to the goal state. For example, when sending a data stream between two nodes in the same LAN, the operator for crossing a network link on the other side of the globe cannot be statically eliminated. Since we do not expect practical problems to require use of all possible operators, what distinguishes a good CPP solution is its ability to scale well in the presence of large amounts of irrelevant information. Our solution combines multiple AI planning techniques and exploits the problem structure to drastically reduce the search space.

The algorithm uses two data structures: a *regression graph* (RG) and a *progression graph* (PG). RG contains operators relevant for the goal. An operator is *relevant* if it can participate in a sequence of actions reaching the goal, and is called *possible* if it belongs to a subgraph of RG rooted in the initial state. PG describes all world states *reachable* from the initial state in a given number of steps. Only possible operators of the RG are used in construction of the PG.

The Sekitei algorithm consists of four phases shown in Figure 7 and described in detail below. Each of the phases solves a relaxed problem. A solution to the relaxed problem is an argument of a new subproblem, which is passed to the next phase of the algorithm. Thus, the regression phase of the algorithm finds a smallest set of possible operators for the original problem with all resource requirements ignored. This set of operators is then used by the progression phase to determine if the goal is reachable given this set of operators and an aggregated version of resource constraints. If it is not, the algorithm backtracks to the regression phase to obtain a bigger set of possible operators. If the goal is reachable, the PG, which contains an aggregated representation of all plans reaching the goal, is passed to the third phase of the algorithm, plan extraction. The plan extraction phase performs a search in the PG, and all candidate plans are passed to the last phase of the algorithm for symbolic execution. Success of the fourth stage guarantees that the found plan is correct.

6.1.1 Regression phase

The regression phase considers only logical preconditions and effects of operators in building the RG, an optimistic representation of all operators that might be useful for achieving the goal. RG contains interleaving fact and operator levels, starting and ending with a fact level, and is constructed as follows.

- Fact level 0 is filled in with the goal.
- Operator level i contains all operators that achieve some of the facts of level $i-1$.

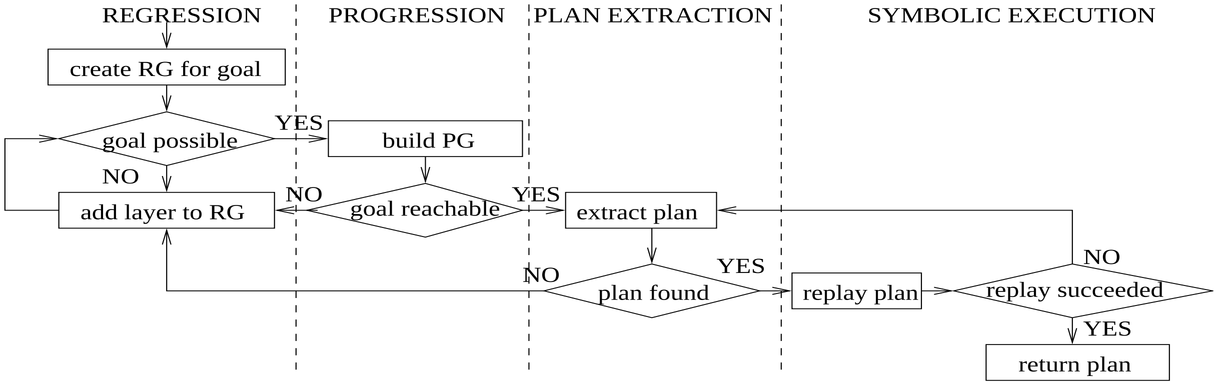


Figure 7: The algorithm. RG stands for “regression graph”, PG for “progression graph”

- Fact level i contains all logical preconditions of the operators of the operator level i.

RG is initially constructed until the goal becomes possible, but may be extended if required. Figure 8 shows the RG for the problem presented in Section 3.1. Bold, solid, and dashed lines correspond to possible subgraphs with 3, 4, and 5 steps respectively.

6.1.2 Progression phase

RG provides a basis for the second phase of the algorithm, the construction of the progression graph. PG also contains interleaving operator and fact levels, starting and ending in a fact level. In addition, this graph contains information about mutual exclusion (mutex) relations [24], e.g., that the placement of a component on a node might exclude placement of another component on the same node (because of CPU capacity restrictions). Because of this, the PG is less optimistic than the RG. Figure 8(right) shows the PG corresponding to the RG in Figure 8(left), which is constructed as described below. Straight lines show relations between propositions and operators, the dotted arc corresponds to a mutex relation.

- Fact level 0 contains facts true in the initial state.
- For each of the propositions of level $i-1$ a no-op (frame) operator is added to level i that has that fact as its precondition and effect, and consumes no resources (marked with square brackets in the figure).
- For each of the possible operators contained in the corresponding layer of the RG, an operator node is added to the PG if none of the operator's preconditions is mutex at the previous proposition level.
- The union of logical effects of the operators of the level i forms the i^{th} fact level of the graph.
- Two operators of the same level are marked as mutex if (i) some of their preconditions are mutex, (ii) one operator changes a resource variable used in an expression for preconditions or effects of the other operator, or (iii) their total resource consumption exceeds the available value.
- Two facts of the same level are marked mutex if all operators that can produce these preconditions are pairwise mutex.

In addition to purely logical structure, construction of the PG takes into account resource preconditions and effects. For each propositional layer of PG, an *optimistic resource map* is computed as described in Section 6.2. An optimistic resource map describes possible levels of resources achievable at a given stage of plan execution, and may contain false positives, but no false negatives. Given the assumption about monotonicity of resource functions, this means that, if an execution of an operator fails in the optimistic resource map for some layer of the PG, no valid plan can contain that operator at the position corresponding to the layer. However, success of an operator execution in the optimistic map does not guarantee existence of a valid plan containing that operator. Operators whose execution fails in the optimistic map of the preceding propositional layer, are not added to the PG.

Because of this resources-based pruning, it is possible that the last level of the PG does not contain the goal, or some of the goal propositions are mutually exclusive. In this case, a new step is added to the RG, and the PG is reconstructed.

6.1.3 Plan extraction phase

If the PG contains the goal and the goal is not mutex, then the plan extraction phase is started. This phase exhaustively searches the PG [2], using a memoization technique to prevent reexploration of bad sets of facts in subsequent iterations. The extracted plan is marked in bold lines in Figure 8(right).

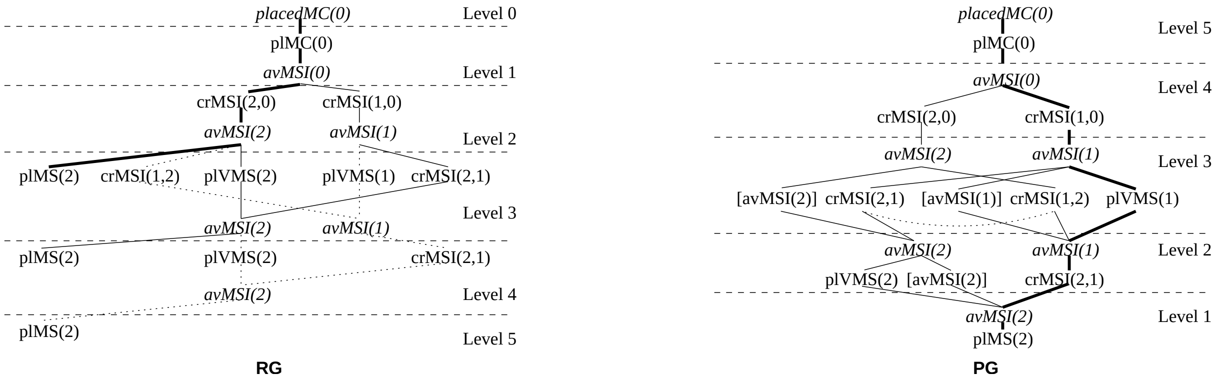


Figure 8: Regression and progression graphs.

6.1.4 Symbolic execution

As mentioned above, optimistic resource maps constructed at the second phase of the algorithm can produce false positives. It is also impossible to propagate goal intervals backwards during the plan extraction phase as done in [24] due to non-reversible nature of the resource functions. Therefore, symbolic execution is the only way to ensure soundness of a solution. It is implemented in a straightforward way: a copy of the initial state is made, and then all operators of the plan are applied in sequence, their preconditions evaluated at the current state, and the state modified according to the effect assignments. Note that correctness of the logical part of the plan is guaranteed by the previous phases; here, only resource conditions need to be checked.

6.2 Reasoning about resources

The layered structure of the Sekitei algorithm allows it to prune the search space and thus deal with the scale of the CPP. The other important feature of this problem is that the world state contains real-valued resource variables and operators have resource preconditions and effects. We assume that all resource functions are monotonic. For example, if bandwidth of a data stream at the source increases, the bandwidth at the destination will not decrease, and if a component can be deployed on a node with less resources, it still can be deployed on that node if more resources become available. These assumptions are true for the applications we are addressing. This section introduces a notion of resource maps and shows how they are used in Sekitei to reason about resources.

6.2.1 Optimistic resource maps

Execution of an operator changes values of resource variables as described by the operator's resource effects. Let $V = \{v_1, ..., v_n\}$ be the set of all resource variables. A *state* is described by a set of name-value pairs for all variables:

$$S = \{(v_1, c_i), ..., (v_n, c_n)\}, \text{where } \forall i \; c_i \in \mathbb{R}$$

Execution of an operator op in a state produces a new state where values of some variables are changed:

$$exec(op, S) = S'$$

A *resource map* is a mapping of each variable in V to a minimum and maximum value.

An *optimistic resource map* $lmap(l)$ for a given layer l of the planning graph is defined recursively as follows. $lmap(0)$ maps each variable into its minimum and maximum value in the initial state. For $l > 0$, $lmap(l)$ maps resource v to the minimum and maximum value of v over all states that result from applying any operator of layer l of the progression graph to any state consistent with $lmap(l-1)$.

According to this definition, to compute a map resulting from execution of an operator in an optimistic map map, we need to execute the operator in a (possibly infinite) set of states consistent with the map. However, since all resource functions are monotonic, it is sufficient to construct states using only boundaries of the intervals. Let $single(map)$ be a set of all such states for the map map:

$$single(map) = \{S_j\}$$

where

$$map = \{(v_1, cm_1, cM_1), ..., (v_n, cm_n, cM_n)\}$$
$$S_j = \{(v_1, c_1), ..., (v_n, c_n)\}, \forall i \; c_i \in \{cm_i, cM_i\}$$

Now the optimistic resource map can be computed as follows.

1. $lmap(0) = \{(v_i, cm_i, cM_i) | v_i \in V\}$, where cm_i and cM_i are minimum and maximum values for resource v_i in the initial state.

2. Let $ops(l)$ be the set of operators, including no-ops, of layer $l > 0$ of the planning graph. Then

$$\begin{aligned}&lmap(l) = \{(v_i, cm_i, cM_i)|\\ &\quad cm_i = min\, c, cM_i = max\, c,\\ &\quad (v_i, c) \in exec(op, S), op \in ops(l),\\ &\quad S \in single(lmap(l-1))\}\end{aligned}$$

6.2.2 Example

To illustrate how resource maps are constructed, consider the following simple example of the mail application (Figure 9). The network consists of three nodes connected in a chain. There is an instance of the `MailServer` running on node 2 able to serve up to 10 requests per second, i.e., `MailServerInterface` is available on that node with `MailServerInterface`.NumReq=10. The link between nodes 1 and 2 has low bandwidth as shown in the figure. We want to place a `MailClient` on node 0, and the client needs to be able to issue 7 requests per second with request size 10. Suppose now that we can place a `ViewMailServer` component on any of the nodes, and `ViewMailServer` reduces the number of client requests by a factor of two. Therefore, a good deployment plan would include two link crossing operations, placing `MailClient` on node 0, and placing `ViewMailServer` on node 0 or 1.

Figure 10 shows the regression graph for this problem. Figure 11 shows the corresponding progression graph with resource maps built for each proposition layer. The initial map contains intervals for each resource variable corresponding to values of those variables in the initial state. The second map is a union of maps resulting from execution of each of the three operators of the first operator layer in the initial resource map. For example, the number of requests supported by `MailServerInterface` on node 1 (`MSI.NumReq(1)`) can be between 0 (if `plVMS(2)` or `[avMSI(2)]` are executed) and 4 (if `crMSI(2,1)` is executed). As can be seen from the graph, even though logical precondition of placement of the `Client` on node 0 (availability of `MailServerInterface`) can be achieved in two link crossing operations, at least three plan steps are required to satisfy its resource precondition `MSI.NumReq(0)`>`7`.

6.3 Improving performance of resource reasoning

The fact that symbolic execution is performed after plan extraction leads to poor performance of the planner in scenarios where steps are added to the plan solely because of resource restrictions (see Section 7.3.1 for experimental results). In such cases, many resource conflicts are detected very late. If the operator that fails during the symbolic execution is close to the end of the plan, then the same plan prefixes are evaluated many times. For example, in both of our applications, all plan prefixes succeed up to the placement of the client. The version of the algorithm presented so far evaluates these plan prefixes for all possible plan tails, which leads to a worst case exponential time spent in the third phase of the algorithm before the problem can be detected.

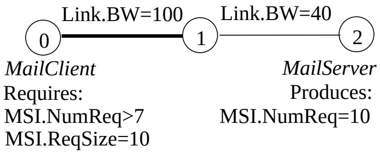


Figure 9: A simple example of a mail application.

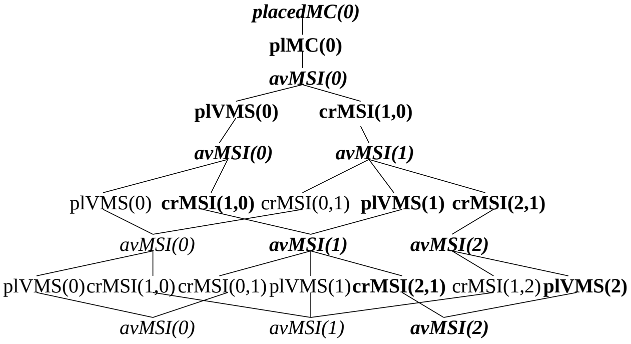


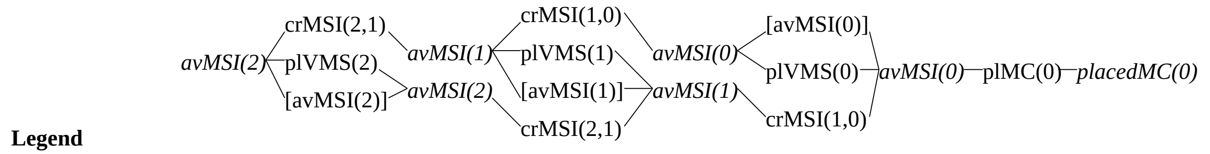
Figure 10: The regression graph for the problem shown in Figure 9. Possible subgraph is shown in bold font.

In this section we present two modifications to the resource processing algorithm that drastically improve performance of Sekitei in such scenarios. The first technique, referred to as positive memoization, takes advantage of saving intermediate results during the search. It does not add any restrictions to the model of the problem, but has high memory requirements. The second modification shows good performance without memory explosion, but assumes absence of negative logical preconditions. The latter, however, is true for all instances of the CPP we have encountered.

6.3.1 Positive memoization

One solution to the late resource conflict detection problem is to save intermediate results. GraphPlan-based algorithms use a technique called memoization: for each of the layers of the planning graph, sets of propositions not achievable together are memoized, so that they do not get checked more than once. Similar to this, we use *positive memoization* to save good sets of propositions along with corresponding resource maps.

The high-level goal of positive memoization is to detect resource conflicts earlier during the plan extraction phase by executing plan tails in the optimistic resource maps. In



Legend					
Link.BW(0,1)	(100, 100)	(100, 100)	(60, 100)	(0, 100)	(0, 100)
Link.BW(1,2)	(40, 40)	(0, 40)	(0, 40)	(0, 40)	(0, 40)
MSI.NumReq(0)	(0, 0)	(0, 0)	(0, 4)	(0, 8)	(0, 8)
MSI.NumReq(1)	(0, 0)	(0, 4)	(0, 8)	(0, 8)	(0, 8)
MSI.NumReq(2)	(10, 10)	(10, 20)	(10, 20)	(10, 20)	(10, 20)

Figure 11: The progression graph with per-layer resource maps for the problem shown in Figure 9. `[avMSI(2)]` is a no-op operator for proposition `avMSI(2)`.

the Sekitei algorithm described above, the maps are built per layer. To make resource conflict detection more effective, we need to calculate resource maps at finer granularity.

Similar to the optimistic resource map for the whole layer, we define an optimistic resource map $smap(q, l)$ for a subset of propositions q at layer l of a planning graph:

1. $smap(q, 0) = lmap(0)$ for all q.

2. Let $ops(q, l)$ be a set of smallest subsets of operators, including no-ops, at layer l that together achieve q.

   Let $precs(o, l)$ be a set of preconditions (propositions at level $l - 1$) of the set of operators o at level l.

   Then the optimistic resource map $smap(q, l)$ for $l > 0$ is defined as follows:

   $$\begin{aligned} &smap(q, l) = \{(v_i, cm_i, cM_i)| \\ &\quad cm_i = min\ c, cM_i = max\ c, \\ &\quad (v_i, c) \in exec(op, S), op \in O, O \in ops(q, l), \\ &\quad S \in single(smap(precs(O, l), l - 1))\} \end{aligned}$$

   In words, each subset of operators achieving q is executed in the optimistic resource map for the union of preconditions of these operators, and then the map for q is computed as a union of the resulting maps.

After the optimistic map is computed for the goal state, the plan extraction phase proceeds as usual, except every time a subset of operators o is chosen at some level l, the plan tail including o is replayed in the optimistic map of o's preconditions $smap(precs(o, l), l - 1)$. For example, three operators of the second layer of the PG shown on Figure 11, `crMSI(1,0)`, `plVMS(1)`, and `[avMSI(1)]`, have the same logical precondition *avMSI(1)*. An optimistic resource map for the singleton set containing this precondition is computed only once, and then reused when other two operators are considered by the plan extraction procedure.

Intuitively, the use of positive memoization in planning with resources is similar to the use of binary mutex relations in planning graph-based algorithms. Whenever a set of propositional preconditions is constructed during the plan extraction phase of the algorithm, it is first looked up in a table. The table contains information about whether the set is not achievable (the standard memoization) or, if the set is achievable, then what is the optimistic map for this set (positive memoization). The recursive call is performed only when the table contains no information. A new table entry is created upon exit from the recursive call.

Adding positive memoization to Sekitei resulted in huge (orders of magnitude) speedup on some instances of the webcast problem and a small increase of running time on simple problems (see Section 7.3.1 for results). Note that the use of positive memoization does not put any additional restrictions on the form of resource functions; only monotonicity is required. Unfortunately, positive memoization has high memory requirements. Having resource maps for all sets of propositions (essentially, most of the subsets of sets of propositions for each layer of the planning graph) leads to a worst case exponential memory explosion.

6.3.2 Per-proposition resource maps

One way to improve memory behavior of positive memoization is to save resource maps per proposition rather than per set of propositions. We implemented this idea in a version of Sekitei referred to as SekiteiNG.

In the presence of arbitrary resource functions mutex relations based on resource interference between operators do not provide sufficient pruning, and therefore can be omitted. Note that, since the CPP does not have negative logical preconditions or effects (component placement does not require or result in *absence* of an interface on a node), resource interference is the only source of mutex relations.

Recall that the purpose of the PG in the Sekitei algo-

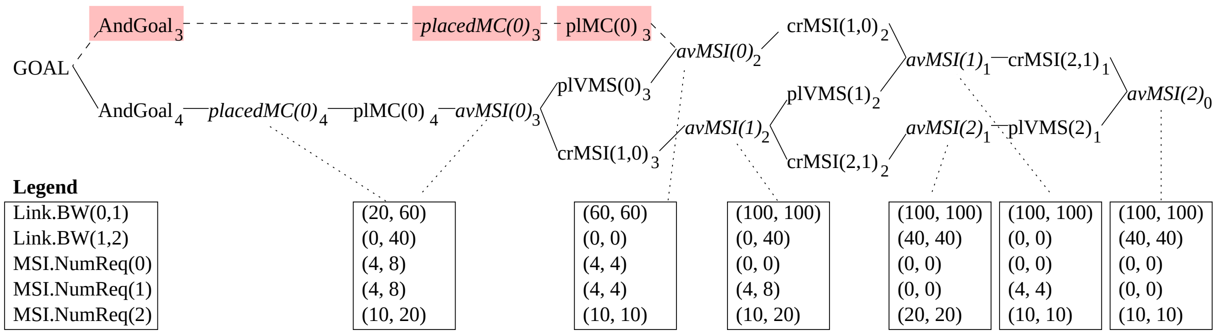


Figure 12: Regression graph of SekiteiNG. Proposition nodes are shown in italics, operator nodes in normal font. Subscripts correspond to the cost of a node. Execution of operator plMC(0) fails in the resource map for $avMSI(0)_2$. Therefore nodes $plMC(0)_3$, $placedMC(0)_3$, and $AndGOAL_3$ are dead.

rithm is to compute mutex relations and to provide basis for computation of the memoization table. Without mutex relations in SekiteiNG there is no need to explicitly store the PG. All information contained in the PG can be merged into the RG. The regression graph still needs to be constructed, because the resource maps built using the memoization table are optimistic, and exhaustive search and symbolic execution still need to be performed. The version of the core Sekitei algorithm used in SekiteiNG is the following:

1. Build a regression graph RG until the initial state is reached.
2. Propagate resource maps in the RG starting from the initial state. This step has the same purpose as the PG construction of the original version, and is described in detail below.
3. Extract a plan from the RG (see below).
4. Perform symbolic execution.

As before, each next step of the algorithm is invoked only if the previous step succeeds, and uses the results of the previous step as its input.

The regression graph of SekiteiNG contains three types of nodes: AND nodes correspond to operators, OR nodes to propositions, and aggregate nodes to collections of propositions. Each of the nodes also has a *cost*, which is the number of operators performed to reach the node from the initial state. The cost of the node in the RG of SekiteiNG corresponds to the layer number of the PG of the original algorithm. A node is considered *dead* if it cannot be achieved in the given number of steps (its operator/proposition does not belong to the corresponding layer of the PG). Otherwise the node is considered alive and has an optimistic resource map associated with it. A goal node is a special kind of an AND node with all goal propositions being its preconditions.

The nodes of the RG are expanded as follows (the first step of the algorithm). An OR (proposition) node with cost $n > 0$ has a child AND node with cost n for each operator that can achieve this proposition. An OR node with cost 0 is achieved by a special INIT node if the proposition is true in the initial state. The map of such a node is equal to the initial map. An OR node is dead if all of its children are dead. Otherwise the map of the node is computed as a union of the maps of its alive children.

An AND node with cost n and a set of preconditions S is expanded as follows. A set of aggregate nodes is created such that

- An aggregate node has $|S|$ child nodes, one for each of the propositions in S.
- The cost of each proposition node is between $n - 1$ and minimum cost of that proposition (see below).
- At least one of the children of an aggregate node has cost $n - 1$.
- An aggregate node is dead if at least one of its children is dead. Otherwise the map of the node is computed as a union of maps of its children.

An AND node is dead if all of its children are dead, or if the operator fails in the map computed as a union of maps of the node's alive children. The map resulting from a successful execution is taken as a map of the AND node.

Figure 12 shows the regression graph for the example from Section 6.2.2. In this example all operators have exactly one precondition. Therefore, all AND nodes have exactly one aggregate node, which are not shown in the figure.

The above algorithm can work even if the minimum cost of each proposition is assumed to be 0. However, additional pruning can be achieved if the minimum cost of a proposition is computed using a regression graph for the

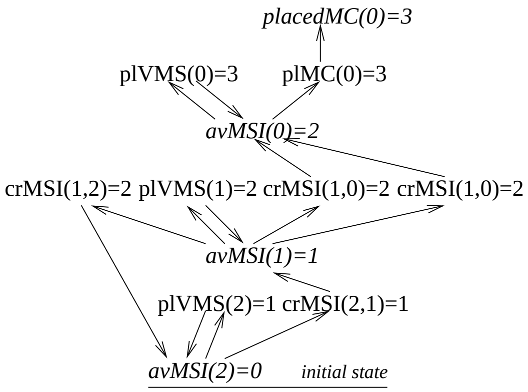


Figure 13: Relaxed graph of SekiteiNG

relaxed (without resources) problem. Initially, this graph is constructed for the minimum number of steps necessary to reach the initial state, and then extended when necessary. Figure 13 shows the relaxed graph for the example from Section 6.2.2. The arrows go from preconditions to operators and from operators to their effects. The cost of an operator node is computed as the maximum cost of its preconditions plus 1. The cost of a proposition node is the minimum cost of operator achieving it.

Since resource maps are unioned at various points during construction of the regression graph, the graph is optimistic. This means that even if the goal node is not dead, the corresponding graph may not contain a solution. To extract a solution (or prove its absence), a search is performed in the regression graph. The basic idea is similar to that of the plan extraction step with positive memoization. A totally ordered plan tail is grown starting from the goal state. After selection of a new operator, the plan tail is replayed in the corresponding resource map. The following describes plan construction for a given aggregate node of the goal node.

1. Create a Queue, initialize it with OR nodes of the aggregate node.
2. Create an empty plan tail.
3. Select the most expensive OR node OrN from the Queue. If the cost is 0, return the plan tail.
4. Nondeterministically choose an AND node $AndN$ from children of OrN. Add the corresponding operator to the plan tail.
5. Nondeterministically select an aggregate node AgN of $AndN$.
6. Compute a working resource map as a union of the maps of OrR nodes from the queue and the map of AgN.
7. Execute the plan tail in the working map. If the execution fails, backtrack.
8. Add children of AgN to the Queue.
9. Go to step 3.

As experiments presented in Section 7.3.2 show, SekiteiNG performs similar to Sekitei with positive memoization, but without the memory explosion side effect. The main restriction of SekiteiNG is that it supports only positive logical preconditions and effects (which is sufficient for the CPP), while the original version of Sekitei (with or without positive memoization) is also capable of supporting negative preconditions and effects.

7 Evaluation

In this section we present experimental results illustrating performance of different versions the Sekitei algorithm. First, we illustrate scalability of the algorithm with respect to the problem size. Second, we show how Sekitei can take advantage of existing component deployments. Finally, we demonstrate effect of our optimizations on the planning time and memory requirements of the algorithm. The measurements reported in this section were taken on a 700MHz Pentium III machine running Windows 2000 and the 1.3.1 Java HotSpot(TM) Client VM using our Java implementation of Sekitei.

To model different wide-area network topologies, we used the GT-ITM tool [7] to generate eight different networks N_k (for different $k \in \{22, 33, \ldots, 99\}$ nodes). Each topology simulates a WAN formed by high speed and secure stubs connected by slow and insecure links. The initial topology configuration files (.alt) were augmented with link and network properties using the Network EDitor tool [22].

The performance of the planner was evaluated using the two applications described in Section 3. The goal in both applications is to deploy the client components on specific nodes. In both cases, the “best” deployment is defined as the one with the fewest number of components.

7.1 Scalability evaluation

We tested scalability of Sekitei (the original version presented in Section 6.1) by running several experiments. This subsection presents in more detail the goal, the description, and the results of each experiment.

7.1.1 Planning under various conditions

The purpose of the first experiment is to show that the planner finds a valid component deployment plan even in hard cases, and usually does so in a small amount of time. The experiment, involving the mail service application, is conducted as follows. For each network topology N_k, where $k \in 22, 33, ..., 99$, and for each node n in the network N_k, the goal is to deploy a `MailClient` component on the node n given that the `MailServer` is running on some node. The algorithm indeed finds a solution when it exists.

The data points in Figure 14 represent the time needed to find a valid plan for each of the different networks, and correspond to the following cases. When the client and the server are located in the same stub, the algorithm essentially finds the shortest path between two nodes, which takes a very short time.[5] Placement of a client in a different stub requires inserting some components into the path, and therefore takes longer.

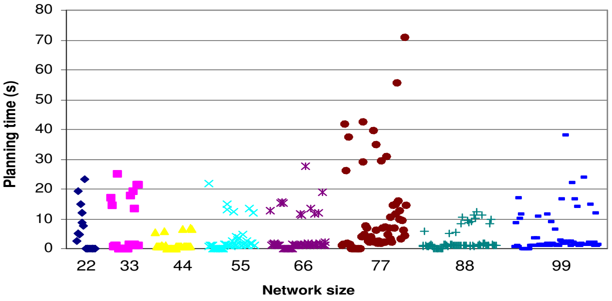


Figure 14: Planning under various conditions.

7.1.2 Scalability w.r.t. network size

To see how the performance of the algorithm is affected by the size of the network, we ran the following experiment. Taking the N_{99} network topology (Figure 15) as our reference and starting with a small network with only two stubs, we added one stub at a time until the original 99-node configuration was achieved. For each of the obtained networks we ran the planner with the goal of placing `MailClient` on a fixed node. Figure 16 shows the planning time as a function of the network size.

As shown in Figure 16, the running time of the planner increases very little with the size of the network. Moreover, the graph tends to flatten. Such behavior can be explained by the fact that the regression phase of the algorithm considers only stubs reachable in the number of steps bounded by the length of the final plan. Even this set is further pruned at the progression stage. Therefore, our algorithm is capable of identifying the part of the network relevant to the solution, without additional preprocessing.

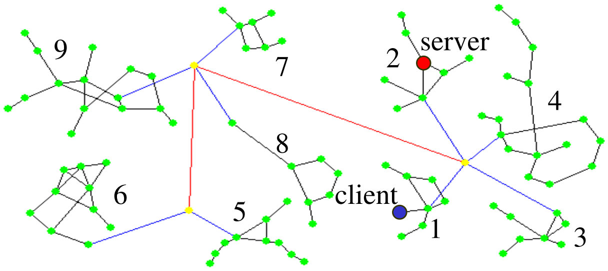


Figure 15: 9-stub network N_{99}

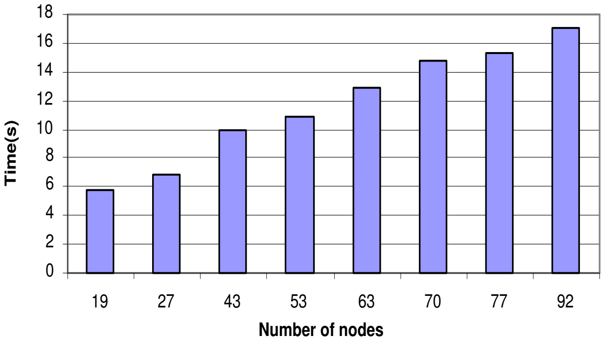


Figure 16: Scalability w.r.t. network size for the mail application.

7.1.3 Complex application structure

The mail application used in the above experiments requires only a chain of components. An important feature of our algorithm is that it can support more complicated application structures, i.e., DAGs and even loops. To verify that planner behavior is not negatively affected by DAG-like structures, we generated deployments for the webcast service (the DAG structure arises because of splitting and merging the image and text streams). The goal for the planner was deployment of the `Client` component on a specific node, given that the `Server` was separated from it by links with low available bandwidth. Figure 17 illustrates the running time of the algorithm as a function of the network size and validates our assertion.

7.1.4 Scalability w.r.t. irrelevant components

To analyze the scalability of the planner when the application framework consists of a large number of components, we classify components into three categories: (i) absolutely useless components that can never be used in any application configuration; (ii) components useless given availability of interfaces in the network, and (iii) useful components, i.e., those that implement an interface relevant for achieving the goal and whose required interfaces

[5] The algorithm does not distinguish any special cases. “The shortest path” is only a characterization of the result.

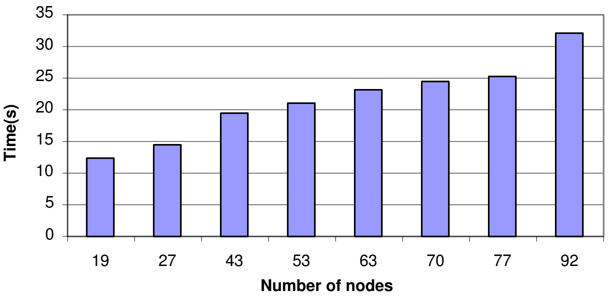


Figure 17: Scalability w.r.t. network size for webcast application.

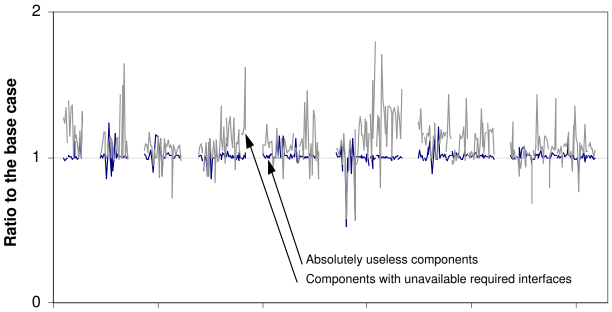


Figure 18: Scalability w.r.t. increasing number of irrelevant components.

are either already present or can be provided by other useful components.

Figure 18 shows the performance of the planner in the presence of irrelevant components. The two plots correspond to two situations: the mail service application augmented first with ten absolutely useless components, and then with ten components that implement interfaces meaningful to the application, but require interfaces that cannot be provided. The absolutely useless components are rejected by the regression phase of the algorithm and do not affect its performance at all.[6] Components whose implemented interfaces are useful, but required interfaces cannot be provided can be pruned out only during the second phase, which also takes into account the initial state of the network (the required interfaces might be available somewhere from the very beginning). The running time increases as a result of processing these components in the first phase (polynomial in the number of components).

Scalability with respect to relevant components is discussed in Section 7.3.

7.2 Reusability of existing deployments.

In practical scenarios, by the time a new client requests a service, the network may already contain some of the required components. To see how the planning time is affected by reuse of existing deployments, we ran the following experiment. Starting with the webcast application and the N_{99} topology where the `Server` was present on a fixed node, we analyzed the planning costs for the goal of putting the `Client` on each of the network nodes in turn. The x-axis in Figure 19 represents the order in which the nodes were chosen. The network state is saved between the runs, so that clients can join existing paths. We assume that clients are using exactly the same data stream, and there is no overhead for adding a new client to a server.

As expected, it is very cheap to add a new client to a stub that already has a client of the same type deployed (this corresponds to the majority of the points in Figure 19), because most of the path can be reused. The problem in this case is effectively reduced to finding the closest node where the required interfaces are available.

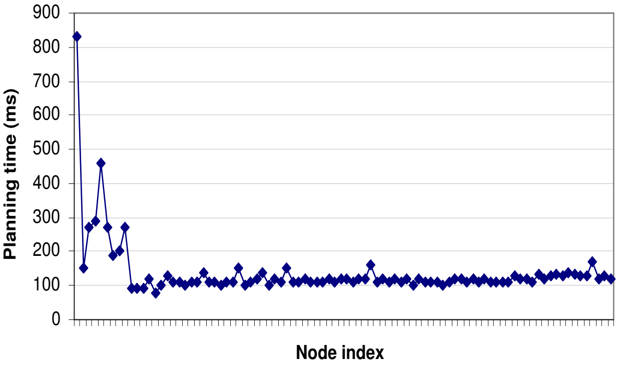


Figure 19: Reuse of existing deployments.

7.3 Benefits from optimizations

7.3.1 Planning time

Figure 20 shows the performance of the original planner (without optimizations) with increasing number of useful components.

In this experiment, the webcast client is placed in turn on each of the nodes of the N_{99} network given a fixed location of the server. The graph shows average planning time per client per stub. The four bars correspond to four different network conditions and application configurations.

Cfg 1. In the first case the transit links have high bandwidth, so that the `Client` can be directly connected to the `Server`.

[6] Slight fluctuations are a result of artifacts such as garbage collection.

Cfg 2. In the second case, the bandwidth of transit links is slightly lower, so that the `Client`'s quality requirements, which are originally specified in terms of the request rate, cannot be satisfied by a direct connection. However, it is sufficient to reduce the color depth of the image portion of the stream to resolve the problem. Therefore, the planner decides to insert `Splitter`, `Merger`, and `Filter` components into the data path.

Cfg 3. In the third case the bandwidth of transit links is even lower, but using compression of the text portion of the stream solves the problem. The planner decides to add `Splitter`, `Merger`, `Zip`, and `Unzip` components into the data path. Note that this plan requires more components then the previous scenario. Therefore, even though the quality of the resulting stream is better in this case than when the `Filter` is used, the planner's decision to use `Filter` in the previous scenario instead of `Zip` and `Unzip` is correct.

Cfg 4. Finally, the fourth configuration includes five additional components (`Splitter`, `Merger`, `Zip`, `Unzip`, and `Filter`).

The choice of whether a useful component is actually used in the final plan is made during the third phase of the algorithm, which in the worst case takes time exponential in the length of the plan. Larger numbers of useful components increase the branching factor of PG, and therefore the base of the exponent. This means that in hard cases (very strict resource constraints, multiple component types implementing the same interface, highly connected networks) the planning can take a long time.

Figure 21 shows the planning time for the same experiment presented above for the planner with the positive memoization technique discussed in Section 6.3.1. The modified version of the planner takes about the same time on simple problems (Configuration 1), and scales much better on harder instances. Figure 22 shows the additional improvements of SekiteiNG discussed below.

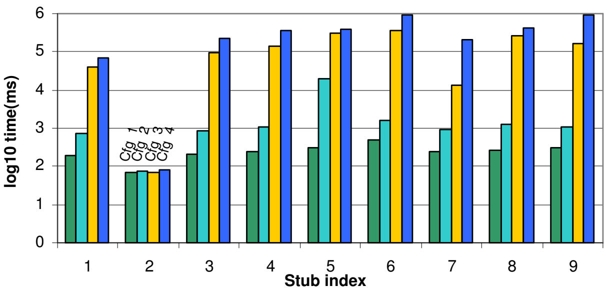


Figure 20: Scalability of the original Sekitei algorithm w.r.t. increasing number of relevant components. The highest peaks correspond to about 15 min.

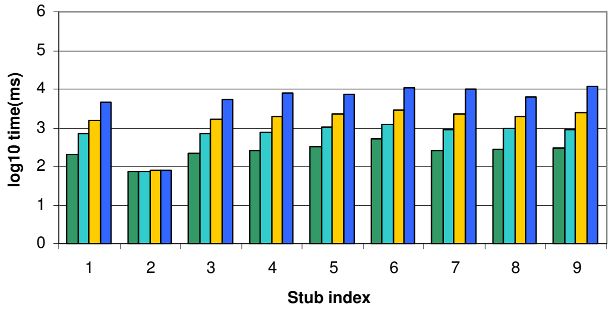


Figure 21: Scalability w.r.t. increasing number of relevant components with positive memoization. The highest peaks correspond to about 10 seconds

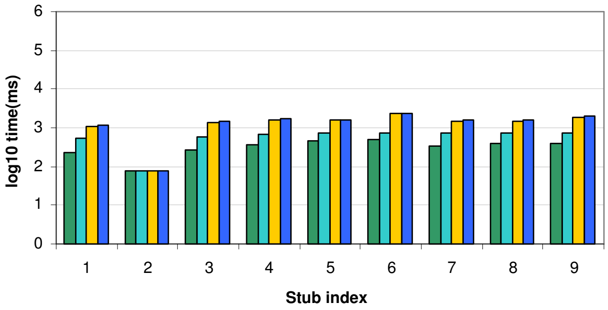


Figure 22: Scalability of SekiteiNG w.r.t. increasing number of relevant components. The highest peaks correspond to about 2.3 seconds

7.3.2 Memory consumption

The main source of memory consumption in all versions of Sekitei is the values of resource intervals stored in resource maps. To evaluate the memory behavior of Sekitei, we recorded the maximum number of such intervals present in memory at any given moment during planning. Although this number is affected by the garbage collection behavior of a Java VM, it is a reasonable estimate of the memory consumption of the algorithm.

Figure 23 shows the average number of constants generated by the three versions of Sekitei on four configurations of the webcast application discussed above. SekiteiNG scales much better with respect to memory consumption as compared to the positive memoization version of Sekitei. In fact, as shown in Figure 24, memory consumption of SekiteiNG is comparable to that of the original version of the algorithm. The number of constants generated by SekiteiNG on configurations 3 and 4 is less than three times bigger than that of the original algorithm.

The fact that SekiteiNG considers much fewer resource values also affects the planning time. Figure 22 shows the

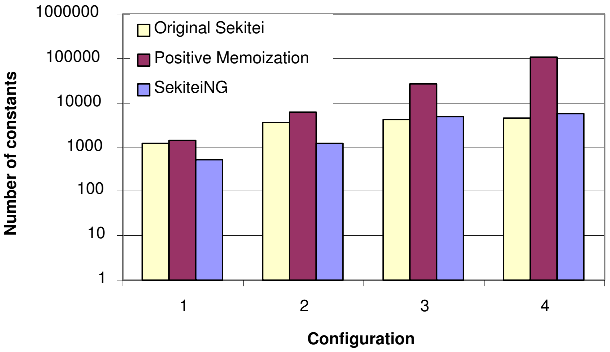


Figure 23: The average number of constants generated by the three versions of Sekitei on four configuration of the webcast application

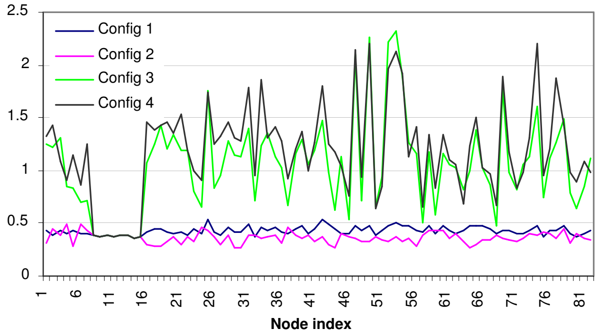


Figure 24: Ratio of constants generated by SekiteiNG w.r.t. the original algorithm.

planning time of SekiteiNG for the four configurations of the webcast application (compare to Figures 20 and 21).

We also tested the scalability of SekiteiNG with respect to irrelevant operators. The behavior of the planner is similar to that reported in Section 7.1, and we do not present the detailed results here.

8 Discussion and future work

Sekitei achieves good scalability on the CPP in presence of non-reversible resource functions by using a combination of regression and progression techniques. Such a combination is beneficial when it is impossible to prune the set of operators using standard preprocessing techniques. On classic planning problems (without resources) this technique does not give any speedup, even though the size of the progression graph is smaller than that of the standard GraphPlan approach.

The examples presented in this paper describe placement of a single component on a given node. The same algorithm can be used for placing multiple components, or for fixing existing deployments after a failure due to change in resource availability.

Extensions to Sekitei need to focus on two principal directions. First, we believe that the performance of the algorithm can be further improved. In particular, it seems reasonable to explore only the most promising paths. One possible way to identify such paths in the component placement problem is to start by building a direct connection between the client and the server along the shortest path in the network (the cheapest path in the relaxed regression graph), and then deviate from this path and add components only in case of a resource conflict.

Another way to improve performance of Sekitei is to use some properties of resources to prune search. It is often possible to distinguish between monotonic and general resources. A resource is *monotonic* if application of any operator changes its value in the same direction. If some operators can increase and others can decrease the value of a resource, we refer to such a resource as *general*.[7] For example, available CPU is always a decreasing resource in the CPP, but the bandwidth of a data stream may be general if a caching component can be injected into the data path. We are currently investigating use of resource monotonicity information for early resource conflict detection.

The second direction is improving expressiveness of the model of the CPP. This includes a better model for publish-subscribe applications and support for global preconditions. The latter may be used, for example, for parallel applications where all copies of the same component need to be deployed with the same parameters.

The current Sekitei implementation does not take into consideration the actual load on components, e.g. the number of clients connected to a server. One way of capturing such incremental resource consumption in our current model is by introducing artificial components that can support a limited number of additional clients. A more general scheme may include changing the formulae describing component placement to consider parameters of implemented interfaces (as opposed to their upper bounds).

The current version of our planner, as many other AI planners, minimizes the total number of parallel steps. In real world problems, such as the CPP, application of an operator usually involves some cost. It is more desirable to minimize the total cost of a plan rather than its parallel length. Supporting a notion of resource-dependent operator cost may help realize this objective.

Another interesting research direction is allowing uncertainty in the resource values of the initial state and producing sensitivity information for a plan.

We are working on adding distributed planning capa-

[7] [1] proposes a classification of resource variables.

bilities to Sekitei. The reason for this is that it is desirable for each administrative domain to have its own planner, which plans for nodes in its domain collaborating only when necessary.

In addition to improving the presented algorithms, we also plan to evaluate the effectiveness of other approaches for solving the CPP. For example, the progression phase of the four-phase algorithm can be replaced with compilation into an optimization problem. Such an approach will require putting tighter restrictions on the form of expressions used in preconditions and effects. The right balance between the expressiveness of the expressions and the performance of the algorithm is an interesting long-term research question.

9 Summary

In this paper, we have presented the Sekitei algorithm for solving the component placement problem and possible ways to improve its performance. The CPP is a real-world problem, whose compilation into a planning problem is characterized by simple logical structure and arbitrary non-reversible monotonic resource functions. In addition, a planner for the CPP needs to cope with large number of irrelevant operators that cannot be removed by static preprocessing techniques.

Sekitei addresses the scaling problem by using a combination of regression and progression techniques to limit the search space. The positive memoization technique significantly increases performance of Sekitei by allowing early detection of resource conflicts. The main drawback of positive memoization is its high memory requirements. We presented a refined version of the algorithm that addresses this problem.

Sekitei is designed and optimized specifically for the component placement problem. However, techniques developed for the CPP may be useful for other problems as well. We plan to extend our resource planner to support more general planning problems, namely, those containing operators with negative logical preconditions and effects.

10 Acknowledgements

This research was sponsored by DARPA agreements N66001-00-1-8920 and N66001-01-1-8929; by NSF grants CAREER:CCR-9876128, CCR-9988176, and CCR-0312956; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Labs, SPAWAR SYSCEN, or the U.S. Government.

References

[1] T. Bedrax-Weiss, C. McGann, and S. Ramakrishnan. Formalizing resources for planning. In *Proc. of ICAPS'03 Workshop on PDDL*, Trento, Italy, June 2003.

[2] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[3] J. Blythe, E. Deelman, and Y. Gil. Planning for Workflow Construction and Maintenance on the Grid. In *Proc. of ICAPS'03 Workshop on Planning for Web Services*, 2003.

[4] J. Blythe, E. Deelman, Y. Gil, C. Kesselman, A. Agarwal, G. Mehta, and K. Vahi. The role of planning in grid computing. In *ICAPS*, 2003.

[5] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *ECP*, 1999.

[6] F. Bustamante and K. Schwan. Active Streams: An approach to adaptive distributed systems. In *HotOS-8*, 2001.

[7] K. Calvert, M. Doar, and E. Zegura. Modeling Internet topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997.

[8] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[9] M. B. Do and S. Kambhampati. Solving planning-graph by compiling it into CSP. In *AIPS*, pages 82–91, 2000.

[10] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. J. of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[11] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An Open Grid Services Architecture for distributed systems integration. Open Grid Service Infrastructure WG, Global Grid Forum, 2002.

[12] I. Foster, A. Roy, and V. Sander. A quality of service architecture that combines resource reservation and application adaptation. In *IWQOS*, 2000.

[13] J. Frank and E. Kurklu. SOFIA's choice: Scheduling observations for an airborne observatory. In *ICAPS*, 2003.

[14] X. Fu and V. Karamcheti. Planning for network-aware paths. In *Proc. of DAIS*, 2003.

[15] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS: Composable, Adaptive Network Services infrastructure. *USITS-3*, 2001.

[16] M. Helmert. Decidability and undecidability results for planning with numerical state variables. In *AIPS*, 2002.

[17] A. Ivan, J. Harman, M. Allen, and V. Karamcheti. Partitionable Services: A framework for seamlessly adapting distributed applications to heterogenous environments. In *HPDC-11*, 2002.

[18] H. Kautz and B. Selman. Planning as satisfiability. In *ECAI*, 1992.

[19] H. Kautz and B. Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *AIPS*, 1998.

[20] H. Kautz and J. Walser. Integer optimization models of AI planning problems. *Knowledge Engineering Review*, 15(1):101–117, 2000.

[21] T. Kichkaylo. Planning with Arbitrary Monotonic Resource Functions. In *Printed Notes of ICAPS'03 Doctoral Consortium*, 2003.

[22] T. Kichkaylo and A. Ivan. Network EDitor. http://www.cs.nyu.edu/pdsg/projects/partitionable-services/ned/ned.htm, 2002.

[23] T. Kichkaylo, A. Ivan, and V. Karamcheti. Constrained component deployment in wide-area networks using AI planning techniques. In *IPDPS*, 2003.

[24] J. Koehler. Planning under resource constraints. In *ECAI*, 1998.

[25] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In *ECP*, 1997.

[26] P. Laborie. Algorithms for propagating resource constraints in ai planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, 143(2):151–188, 2003.

[27] J. Lopez and D. O'Hallaron. Support for interactive heavy-weight services. In *HPDC-10*, 2001.

[28] D. McDermott. Using regression-match graphs to control search in planning. *Artificial Intelligence*, 109(1-2):111–159, 1999.

[29] Microsoft Corporation. Microsoft .NET. http://www.microsoft.com/net/default.asp.

[30] N. Muscettola. Computing the envelope for stepwise-constant resource allocations. In *Proc. of Principles and Practice of Constraint Programming (CP)*, pages 139–154, 2002.

[31] Object Management Group. Corba. http://www.corba.org.

[32] Object Management Group. CORBA Component Model. *http://www.omg.org/*, 2003.

[33] J. Penberthy and D. Weld. Temporal planning with continous change. In *AAAI*, 1994.

[34] J. S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *KR*, 1992.

[35] P. Reiher, R. Guy, M. Yarvis, and A. Rudenko. Automated planning for open architectures. *OPENARCH*, 2000.

[36] S. Gribble et al. The Ninja architecture for robust Internet-scale systems and services. *Computer Networks*, 35(4):473–497, 2001.

[37] Sun Microsystems, Inc. Java(TM) 2 platform, Enterprise Edition.

[38] S. Wolfman and D. Weld. Combining linear programming and satisfiability solving for resource planning. *Knowledge Engineering Review*, 2000.

[39] D. Zhou and K. Schwan. Eager Handlers - communication optimization in Java-based distributed applications with reconfigurable fine-grained code migration. In *3rd Intl. Workshop on Java for Parallel and Distributed Computing*, 2001.