Construction of Component-Based Applications by Planning

by

Tatiana Kichkaylo

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

January 2005

Research Advisors:

Vijay Karamcheti ______________________

Ernest Davis ___________________________

You are never given a wish without also being

given the power to make it true.

You may have to work for it, however.

Richard Bach

Acknowledgments

First of all, I would like to thank my advisors, Vijay and Ernie, for letting me explore and invent while still keeping me on track. Striking this balance was not an easy task, but you did it. I would also like to thank all people who asked tough questions. John, Dan, and Robert, thank you for your time and help.

I am eternally grateful to my parents for making me who I am. You taught me that, no matter what happens, it is important to keep working. Reminding myself about "at least two ways out" helped me many times.

I thank Anca, my sister *for all practical purposes*, for taking good care of me and making me feel home. I thank my friends, Sveta, Dana, Igor, Oana, Gela, for accepting my not exactly angelic personality. And I thank all my teachers for telling me that no matter how far we get, there is still a road ahead.

Abstract

Many modern wide-area distributed systems are component-based. This approach provides great flexibility in adapting applications to the changing state of the environment and user requirements, but increases the complexity of configuring the applications. Because of the scale and heterogeneity of modern wide-area environments, manual configuration is hard, inefficient, suboptimal, and error-prone. Automated application configuration is desired.

Constructing distributed applications requires choosing a set of components that will constitute the application instance and assigning network resources to component executions and data transfers. Stated this way, the application configuration problem (ACP) is similar to the planning (action selection) and scheduling (resource allocation) problems studied by the Artificial Intelligence (AI) community.

This thesis investigates the problem of solving the ACP using AI planning techniques. However, the ACP poses several challenges not usually encountered and addressed by the traditional AI solutions. The problem specification for the ACP can be much larger than the solution, with the relevant portions only identified during the search. Additionally, the interactions between planning operators are numeric rather than logical. Finally, it is desirable to be able to trade off quality of the solution versus

search time.

We show that the ACP is undecidable in general. Therefore, instead of a single algorithm, we propose a set of techniques that can be used to compose an algorithm for a particular variety of the ACP that can exploit natural restrictions exhibited by that variety. These techniques address the challenges above by dynamically obtaining portions of the problem specification as necessary during the search, using envelope hierarchies based on numeric information for pruning and search guidance, and discretizing continuous variables to approximate numeric parameters without restricting the form of supported numeric functions.

We illustrate these techniques by describing their use in algorithms tailored for two specific varieties of the ACP — snapshot configurations for dynamic component-based frameworks, and scheduling of grid workflows with replica selection and explicit resource reservations. Experimental evaluation of the performance of these two algorithms shows that the techniques successfully achieve their goals, with acceptable run-time overhead.

Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction

1.1 Motivation

A growing number of distributed applications spanning various areas such as adaptive component frameworks, web services, and grid computing, are being structured as aggregations of multiple independent components communicating over a wide-area network such as the Internet. Components cooperate to realize application functionality by invoking each other's services, processing data streams, or reading and writing files.

For example, consider a distributed application that delivers a stream of image and text data from a remote server to a client. In the simplest case, this application consists of the server and client components deployed on two different Internet hosts. Depending on the properties of the client's computer, the properties of the network path between the client and the server, and the client's Quality-of-Service requirements, additional components may be injected into the data path. For example, a transcoding component may be required to convert data into the format understandable by the client; compression and decompression components may be inserted to reduce bandwidth requirements; and encryption components may be used to deal with insecure links.

Although it is possible to encapsulate all the above functionality in the two end-point components (the client and the server), creation of specialized components with well-defined interfaces has several advantages:

- Separate components can be deployed on nodes other then the end points allowing for better load balancing.
- Components can be independently created and modified, reducing the development cost of the end-point components.
- Specialized components, such as compression and encryption, can be reused across many applications.

Given such a modular approach, the notion of a “distributed application” is shifting from the traditional view of statically deployed entities into one defined by a high-level description of its components, their locations, and the linkages between them. An **application configuration** is an instance of such an application.

Irrespective of how much flexibility the components of an application offer, the usefulness of distributed component-based applications from the human user’s point of view ultimately depends on the ability of the particular application configuration to deliver the service requested by the user. Many different application configurations, defined by different sets of components and different component locations, can satisfy the client requirements. Some of such feasible configurations may be preferable, because, for example, they have smaller requirements with respect to some expensive resource. The best choice depends on the current state of the environment (resource availability and costs) and user goals.

What makes it difficult to optimize this choice in wide-area environments, where such flexibility offers the most benefit, is their scale and heterogeneity. For example, when deploying a computationally intensive grid application, one may consider multiple hosts with different properties and an ever-expanding set of reusable components. In large networks, resource availability constantly changes, which make it impossible to predict the future run-time conditions and construct an application configuration that will always work. It is also desirable to be able to seamlessly integrate new versions and types of components into existing applications. Manual approach to application configuration does not provide flexibility necessary to adapt applications to such changes. In addition, manual configuration is hard, inefficient, suboptimal, and error-prone. Automated application configuration is desired.

In this thesis we investigate the problem of automated construction of good application configurations at the deployment time of the application. We refer to this as the **Application Configuration Problem (ACP)**. Solving the ACP requires reasoning about both qualitative (logical) and quantitative (numerical) characteristics of application components and environments.

1.2 Approach and challenges

To construct an application configuration given a set of available components and a state of the environment one needs to make the following decisions:

1. Select a set of component instances;[1]

2. Configure the components, which may involve choosing linkages between component instances and selecting values for component parameters;

3. Assign components to network hosts and data transfers to network links.

The above three decisions depend on each other. For instance, in the context of our earlier example, if one chooses to use compression to cope with a link of low bandwidth, the new application configuration will have smaller bandwidth requirements and higher CPU requirements compared to the no-compression configuration. This in turn may affect feasibility of new component mappings. This interdependency suggests that all three decisions need to be made simultaneously.

To be useful in practice, an algorithm for solving the ACP needs to provide sufficiently high performance, because an application configuration should be constructed at deployment time. This high performance requirement is at odds with the need to support a flexible model capable of accurately describing interactions between components and their resource consumption behavior. Development of a model and an algorithm that can address both these requirements simultaneously is a major research challenge.

Unfortunately, the three subproblems listed above are computationally hard even when considered separately. The mapping subproblem, for example, is as an NP-hard scheduling problem. The ACP, which requires making all three decisions, is therefore even harder.

[1]It is possible to have more than one instance of a given component type in a configuration. For example, compression can be applied independently to different files.

One way to address the complexity of the problem is to restrict the expressiveness of the model. Some of the existing algorithms for solving the ACP have chosen this approach. CANS [37] supports only sequences of components aligned along a given network path. Ninja [44] works only with components already running in the network (as opposed to choosing a host for deployment), which drastically reduces the number of options the planner needs to consider. The limited models allow for creation of fast specialized algorithms. For example, the CANS planner can find a solution in polynomial time using a dynamic programming algorithm. In this thesis we describe a more general solution.

Another possibility is to represent the ACP as an optimization problem, such as an integer linear programming (ILP) or a constraint satisfaction problem (CSP). For example, one could encode the fact that a component of a given type is deployed on a given host using a Boolean variable, and then pose a constraint that the total CPU consumption of all components on a host does not exceed the available value of the resource. However, such approaches also have limitations. First, compilation of all choices involved in expressing the ACP as an optimization problem loses the problem structure, making it hard to determine relevant parts of the problem specification during the search. Using all available information from the very beginning would create optimization problems of very large size, resulting in low performance of the algorithm. Second, traditional optimization techniques, such as linear programming, often support only limited form of functions describing resource consumption behavior. For the application configuration problem, it is desirable to lift such restrictions.

Our approach is motivated by the observation that the problems of component selection and mapping (resource allocation) are similar to the planning and scheduling problems investigated by the Artificial Intelligence (AI) community. AI offers several decades of experience in knowledge representation, planning and scheduling. In recent years several major breakthroughs have been made that allow modern planning algorithms to achieve good performance on challenging benchmarks. This gives us hope that AI can provide techniques for achieving a good balance between flexibility and scalability. The techniques presented in this work are based mostly on AI planning. Therefore, we refer to the algorithm for solving the ACP as a **planner**.

In Chapter 3 we show how the ACP can be modeled as a planning problem with numeric state variables. However, existing planning algorithms are not capable of solving the ACP efficiently. To make AI planning algorithms efficiently solve the ACP, the following three issues need to be addressed.

Scale: The first problem a planner for the ACP needs to address is the scale of the problem specification. When constructing a configuration of an application, one may need to consider thousands of network nodes and hundreds of components available for deployment. Existing AI planners assume that the whole problem specification is given as an input, and rely on this fact to perform various sorts of reachability analyses. Even though the size of the constructed configuration is usually small, using a complete problem specification (and even obtaining it) is often not feasible for the ACP, and new techniques need to be developed to deal with the scale.

Resource functions: The next feature of the ACP that is not supported by existing planners is the fact that resource behavior of components may be described by arbitrary functions. Existing planners usually assume that resource expressions have a particular form, e.g. linear functions with at most one variable [62], and rely on the structure of the resource expression to perform analysis. Some of the existing planning algorithms assume reversibility of resource functions, i.e. that function arguments can be computed given the result. In general, the assumptions of linearity and reversibility do not hold for the ACP. For example, functions describing resource consumption of a component may be specified as tables of profiling results. The algorithm for the ACP needs to be able to deal with such functions.

Feasibility vs. optimality: As we show in Section 3.3.3, even the problem of finding any feasible solution for the ACP is computationally hard. In practice it is desirable to find a "good" application configuration as opposed to just any one.

In this thesis we develop techniques to address these issues, and show how these techniques can be used to implement efficient algorithms for solving several varieties of the ACP.

1.3 Contributions

This thesis develops techniques that permit *efficient* solutions to the application configuration problem with *expressive* models.

The overall contributions of this thesis include

- Modeling of the ACP as an AI planning problem with numeric state variables. The models developed provide expressiveness to capture important properties of the ACP and provide necessary information for automated application configuration.

- Complexity analysis of the ACP.

- A set of techniques for solving the ACP in expressive models. The proposed techniques allow a planner to deal efficiently with large-scale open worlds and complex numeric dependencies between configuration components, and to trade off search time and the solution quality.

- Planning algorithms for two varieties of the ACP as example applications of these techniques. In this thesis we describe a planner for the component placement problem (Sekitei) and a planner for computational grid applications with explicit resource reservations and sharing (GPRS).

- Evaluation of performance and scalability of these algorithms. We show that these algorithms efficiently prune the search space and obtain high quality solutions within reasonable time.

Specifically, the thesis introduces the following techniques:

T1 **On-demand compilation of problem specification** permits the planner to obtain necessary parts of the search space on-the-fly and hides unimportant semantic details of the problem (e.g. units of measurement) from the planning algorithm.

T2 **Regression-progression search** enables efficient cost-based pruning of the search space.

T3 **Bounded resource envelopes** enhances the pruning power of the search in the presence of numeric functions.

T4 **Envelope hierarchies** help to reduce the overhead of envelope computation in large search spaces.

T5 **Costs depending on resource consumption** make it possible to find good application configurations with respect to resource consumption and to specify preferences for resource tradeoffs.

T6 **Resource levels** improve solution quality in the presence of non-reversible numeric functions and provide the means for performance-quality tradeoffs.

T7 **Constraint-based representation of reservations** allows the planner to efficiently reason about resource reservations and sharing.

1.4 Thesis organization

The rest of this thesis is organized as follows. Chapter 2 provides background information, including an overview of related work and examples of the application configuration problem addressed in this thesis. Chapter 3 gives a formal definition of the application configuration problem. It also includes an analysis of the complexity of the ACP, sets the ACP in the context of traditional AI planning and scheduling problems, and identifies challenges that an algorithm for the ACP needs to address.

Chapters 4, 5, and 6 introduce techniques for addressing these challenges. In these three chapters we incrementally introduce Sekitei — a planner for the component placement problem, which is a special case of the ACP. Each of the chapters describes a *version* of the algorithm, which incorporates techniques introduced in that chapter, and provides empirical evaluation of the performance of that version.

Chapter 7 illustrates how the same techniques can be applied to another instance of the ACP, namely, planning for grid applications with explicit reservations.

Finally, Chapter 8 provides a *recipe* for constructing efficient planners for different varieties of the ACP and outlines possible directions for future research.

1.5 Conventions

In this thesis the following formatting conventions are used.

The defined terms are shown in **bold font**. Other important terms, including forward references, are shown in *italics*.

Left-to-right top-to-bottom direction in illustrations is from the goal (client) to the initial state (servers).

To avoid confusion, terms *operator* and *node* refer to the planning algorithm, and terms *action* and *host* to the application configuration problem being solved.

Chapter 2

Background

In this thesis we investigate application of AI planning and scheduling to the problem of construction of application configurations. In this chapter we first describe two types of component-based systems that require solving the ACP, and the current approaches for solving the ACP in those systems. We then discuss state-of-the-art AI planning techniques and their strengths and weaknesses with respect to the ACP. Finally, we describe several instances of the ACP we will use in this thesis to illustrate our solutions.

2.1 Component-based frameworks

The component-based approach is becoming increasingly popular in areas such as adaptive component frameworks, web services, and grid computing. The common model in such systems is that an application consists of communicating components distributed in a wide-area network. Application components can be reused across several applications and dynamically orchestrated to achieve various objectives, such as quality of service delivered to the client and high throughput of the system.

In this section we discuss two classes of such systems: adaptive component-based systems and computational grids.

2.1.1 Dynamic component-based frameworks

Dynamic component-based frameworks (DCBF), such as PSF [51], CANS [38], Ninja [44], and Conductor [98, 84], provide system-level support for run-time deployment and connection of application components. In such systems, all components of an application run simultaneously and communicate by invoking each other's methods using RPC such as Java RMI or by processing streams of data.

The goal of DCBFs is to ensure quality of service received by a client by creating and modifying an application configuration according to the current state of the environment, e.g. changes in network topology and resource availability.

The common mode of operation among existing DCBFs is as follows. Each of the network hosts executes a run-time environment of the DCBF. This run-time environment provides monitoring functions, as well as a component life-time management facility capable of uploading, starting, connecting, and reconfiguring components.

A client wishing to join the system sends a request to the run-time environment of the local host. The DCBF then uses a planning module (see Section 2.1.3) to design an application configuration, and deploys the chosen configuration. Some systems, e.g., CANS [38] monitor the state of the environment throughout the life time of the application configuration, and initiate *reconfiguration* when necessary.

DCBFs differ in the details of the application model they assume. For example, CANS and Conductor support stream-based applications, while PSF is designed for request-reply applications. Frameworks can support fixed (Conductor) or extendible (PSF) sets of application properties. Some systems, such as CANS and PSF, support run-time uploading of component code, while others (Ninja), require the code to be present on the destination node.

Despite these differences, DCBFs share common features.

1. The total number of components constituting an application configuration is usually small.
2. Logical compatibility of components is captured by the types of data streams/interfaces.
3. All components of an application are active simultaneously, and therefore compete for the same network resources.
4. The network considered by a DCBF is usually large and can contain up to hundreds of hosts.
5. An application configuration typically requires a small number of parallel data streams. For example, CANS and Conductor are restricted to chains.

2.1.2 Computational grids

Computational grid infrastructures, such as Globus Grid [30], are another example of component based applications. The overall objective in grids is to seamlessly orchestrate resources distributed in a wide-area network for execution of computation and data-intensive applications. The typical goal of such applications is to obtain a particular data product — a file.

An application configuration in this case consists of a (partially ordered) set of data transfers and component executions (jobs) mapped onto network resources. The decision of whether to execute jobs sequentially or in parallel depends on the producer-consumer relations between the jobs and on the resource availability. Given sufficient resources, multiple jobs can be executed concurrently on the same host. Alternatively, job executions may be divided between several hosts and necessary file transfers may be arranged.

In current grid systems, a user submits a workflow that describes the application structure to a grid scheduling service, such as DAGMan [90]. The scheduling service then discovers necessary resources [81, 89], submits jobs for execution, and monitors the progress of the application, resubmitting jobs if necessary.

The following features are typical in modern grid applications:

1. The total number of different component types constituting an application configuration is usually small. However, multiple instances of each type may be present, so that the application consists of multiple (sometimes hundreds) parallel job sequences.

2. Logical compatibility of jobs is captured by the types of input and output data files.

3. The application is distributed in both network space and physical time, i.e. jobs can be executed sequentially or in parallel on a set of hosts.

4. The network is usually small and consists of a few hosts. Each of these hosts can be a single machine (e.g. a supercomputer) or a cluster of machines organized in a pool [90].

5. Both job executions and data transfers [1] can take a long time. Therefore, it is worthwhile to invest a significant amount of time into construction of a good configuration.

2.1.3 Planners for the ACP

In component-based systems of both classes described above, in order to achieve good performance, the application configuration needs to be constructed at run-time. Some CBFs assume the existence of an external planner [69, 103, 12]; others [33, 44, 38, 51, 84] implement their own planning module. To achieve good performance, the systems of the second category usually simplify the application configuration problem by putting restrictions on the supported application model. This section provides an overview of several such solutions.

Systems such as Ninja [44], CANS [38], and Conductor [84], all of which enable the deployment of appropriate transcoding components along the network path between clients and servers, simplify the assumptions of the planning problem in order to perform directed search. The Ninja planning module focuses on choosing 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 [37] can ensure that host and link capacities along the path are not exceeded by deployed components, while simultaneously optimizing an application metric of interest (e.g., response time).

More general are systems such as Partitionable Services Framework (PSF) [51], 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 original PSF planning module performed exhaustive search to find a valid deployment. A more recent version of PSF uses the Sekitei planner described in Chapter 4 of this thesis.

Globus Architecture for Reservation and Allocation (GARA) [33], the planning module in the Globus [30] architecture, assumes that the task graph is given and deploys these tasks so as 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]

[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.

Pegasus [9] is a grid planner that uses AI techniques. Pegasus generates grid workflows (application configurations) to achieve a user objective — creation of a particular data product. Automated generation of workflows aims to achieve several goals. First, it makes the grid more user-friendly. The automated planner allows the user to specify only a high-level objective instead of a complete task graph. Second, automated generation of workflows permits run-time customization of workflows according to the current resource availability, which leads to better performance of the application. Finally, the automated planner/scheduler can simultaneously work on goals of several users, taking advantage of possible sharing of intermediate data products to improve the global resource utilization [10].

The main challenge Pegasus faces is that of coping with the scale of the problem. As mentioned in Section 2.1.2, grid applications may involve hundreds of components. To deal with such scale, Pegasus divides the application configuration problem into the logical part (selecting components and logical types) and the resource allocation part. Randomized scheduling algorithms such as GRASP [7] are used to obtain high quality schedules.

Pegasus models logical compatibility of components using types of input and output data files. However, it assumes that jobs can execute on any of the available hosts and does not support explicit resource reservations.

One can point out two reasons for the latter model restriction. First, simplicity of the model helps to design more efficient algorithms. Second, the current grid frameworks, such as Globus, do not yet provide facility for representing time-dependent resource availability. However, several ongoing efforts aim to extend the grid models to express such information [2, 70].

2.2 AI planning

In this section we describe the traditional AI planning problem and approaches for solving it. We also discuss the complexity of planning, including planning with numeric state variables.

2.2.1 The classical planning problem

The classical planning problem is to find a sequence of actions (operators) — a plan — that achieves a goal state when executed starting from the initial state. Now let us define these terms more formally.

In the classical planning problem, a **world** contains sets of typed **objects**, e.g. passengers, boxes, airplanes.

Objects can be used as arguments for predicates. A **predicate** describes a statement about the world, which can be true or false at a given time.

A **world state** is described by a set of **ground** (variable-free) instances of predicates, referred to as **propositions**, that are true in that state.

The **initial state** is a completely specified world state. Propositions that are not known to be true are considered false.[2] A **goal** is a logical formula over propositions. Any world state in which this formula evaluates to true is a goal state.

For example, in an air travel planning domain,[3] *passengerA*, *planeB*, and *cityC* are objects, *boarded(?h: human ?p:plane)*[4] is a predicate with two parameters that states that the passenger *?h* is on the plane *?p*, and *boarded(passengerA planeB)* is

[2]This is called the **closed world assumption**.

[3]In this section we adopt the example used in [62]

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

a proposition that can be true or false in different world states.

A world state can be changed by the application of an operator. An **operator** has typed parameters (like a predicate), a precondition, and effects. The set of parameters can be empty. The precondition is a conjunction of propositions and/or their negations. An operator is applicable in a world state if its precondition evaluates to true in that state. Effects of an operator are defined by a set of propositions that the operator makes true (the ADD list) and a set of propositions that it makes false (the DELETE list). For example, the following operator describes boarding of a passenger to a plane in some city. The operator can be applied in any world state, where both the passenger and the plane are in that city. As a result of the operator application, the passenger is considered to have boarded and no longer in the city. The rest of the world state is not affected. In this operator definition, the passenger, the plane, and the city are parameters, so that the operator can be *instantiated* with any objects of appropriate types in the world.

board(?h: human ?p: plane ?c: city)
pre: *in(?h ?c)* $\wedge$ *in(?p ?c)*
eff: ADD *boarded(?h ?p)*; DEL *in(?h ?c)*

The **planning domain** is a set of objects, a set of predicates, and a set of operators. A **planning problem** is defined by a planning domain, an initial state, and a goal.

A **plan** is a sequence of ground instances of operators $o_1 \cdots o_n$ such that o_1 is applicable in the initial state and o_i is applicable in the world state resulting from execution of $o_1 \cdots o_{i-1}$. The planning problem is to find a plan that, when executed starting in the initial state, brings the world to a goal state. For example, if initially both *passengerA* and *planeB* are in *cityD*, then the plan $\{$*board(passengerA planeB),* *fly(planeB cityC cityD), debark(passengerA planeB)*$\}$ achieves the goal *in(passengerA cityD)*.

Due to interactions between the operators (one operator can delete preconditions of another), the length of the shortest plan that achieves a goal can be exponential. The problem of determining if a solution exists for a given planning problem is EXPSPACE-complete [28] (the purely propositional case, when predicates have no parameters, is PSPACE-complete [13]). However, on average, planning can be quite efficient [14].

2.2.2 Planning with numeric state variables

The trend in classical planning research has been towards faster algorithms and more expressive languages. Many modern planners can support universal and existential quantifiers and disjunction in goals and preconditions [77]. However, even with this extension, the classical planning problem is purely logical, i.e. it involves only Boolean values.

Many realistic applications also require reasoning about numeric resources. Planning with numeric state variables is often referred to as *metric planning*. Metric planners extend the above definition of the planning problem in the following way.

A **resource** is a real-valued variable. In addition to the truth values for all propositions, the world state in a metric planning problem contains values for all resources. The goal expression can include (in)equalities with resources, e.g. in the earlier air travel example one might want to complete a trip in a certain amount of time.

Operators in metric planning can have resource preconditions and resource effects. The example below describes an operator for flying between two cities. In this specification, *distance* is a *database entry*, i.e. a constant that can be looked up during the operator instantiation, and *\$gas* and *\$time* are numeric state variables (resources).

fly(?p: plane ?c1 ?c2: city)

pre: *in(?p ?c1)* $\wedge$ *(\$gas > distance(?c1 ?c2)/3)*

eff: ADD *in(?p ?c2)*; DEL *in(?p ?c1);*

\$gas -= distance(?c1 ?c2)/3;

\$time += distance(?c1 ?c2)$*$*(3/20).*

The form of the expressions that might be used in resource preconditions and effects is usually limited. In most cases, only constants (like in the example above) or simple linear expressions are allowed [62, 57]. More complex expressions are either not supported at all, or ignored during the planning phase and then checked by a symbolic execution phase. Such decoupling can cause significant degradation of performance.

Addition of real-valued variables greatly increases the complexity of the planning problem. Helmert [47] analyses several classes of the metric planning problem, admitting various forms of functions in goal conditions and preconditions and effects of planning operators, and shows that in most cases metric planning is undecidable.

Note that, although operators can use continuous variables in their preconditions and effects, only discrete variables are allowed in operator parameters. Planning Domain Definition Language (PDDL) [35, 26] can be used to encode such planning problems.

2.2.3 Search in planning

Planning can be viewed as search in the space of all possible sequences of operators or all possible world states. AI planners build totally ordered or partially ordered plans. Search techniques used by planners of both categories can be divided into four categories. In this section we present these categories in the context of classical planning and describe their advantages and deficiencies with respect to the ACP.

Refinement planners

Refinement planners build a plan by performing a means-ends analysis. For example, UCPOP [79] is a partial order planner that supports universal quantification and conditional effects. UCPOP uses a notion of *causal links*, which are used to record why a step was added to a plan and to protect its purpose. UCPOP keeps a set of *flaws* (things that have to be fixed before the plan is complete) and resolves at least one flaw at each step. A flaw can be an unsatisfied precondition of some action, or a threatened causal link. Initially, the plan contains two artificial steps: the initial step that has the initial state as its postcondition, and the final step that has the goal state as its precondition. The initial set of flaws therefore contains the goal state. Flaws are resolved nondeterministically by adding new steps to the plan, adding causal links between existing steps, and ordering steps to protect causal links.

Refinement planners adopt the least commitment principle, i.e., they delay binding variables to particular values until it is really necessary. Applied to the ACP, this means that most of the possible ground instances of operators are never instantiated, and therefore the total memory requirements of the planner depend on the size of the final plan rather than the size of the ground specification of the problem. Moreover, the planner makes explicit the justification for a planning decision (e.g. an encryption component is placed *because* a particular link is insecure). This provides natural hooks for monitoring. The main disadvantage of refinement planners is their slow performance compared to other approaches.

Planning graph based (progression) planners

Planning-graph based (progression) planners are currently among the fastest ones. Such planners (GraphPlan [8], IPP [64]) first build a special data structure — the *planning graph* — that optimistically describes all world states possibly achievable in a given number of parallel steps. A *parallel step* includes one or more operators that do not conflict with one another and can be executed in any order. The second phase of the algorithm searches this graph for a solution. Planning-graph based algorithms are sound and complete. In addition, planning-graph based algorithms guarantee that the solution found has the minimum number of parallel steps.

Progression planners build the planning graph in the forward direction from the initial state. Therefore complex expressions in effects can be easily processed. The main disadvantage of this approach is that progression planners use a complete ground specification of the initial state for construction of the planning graph. This makes such planners very sensitive to the size of the search space [63].

Regression planners

Regression planners (HSPr [11], Unpop [71]) build a *regression graph* from goals towards the current planning state. The regression graph contains all operators that might possibly be useful for achieving the goals starting from the current state. Such a graph can be used to estimate the cost of achieving a goal, which serves as a heuristic to guide the search. Regression planners are sound and, under some conditions, complete [71].

Regression planners effectively determine the relevant portion of the search space. This feature is very useful for application configuration [72]. However, it is extremely hard to incorporate support for complex effect expressions in such planners.

Compilation-based planners

The fourth approach is to reduce the planning problem to a sequence of simpler (NP-hard) problems. The planner first non-deterministically guesses the number of required plan steps, and then constructs and solves a problem instance corresponding to this number of plan steps in another formalism (SATPLAN [55], ILP-PLAN [57]). A variant of this idea is to apply this technique to the plan extraction phase of the GraphPlan algorithm (BlackBox [56], GP-CSP [25]). Various target formalizations might be used. For example, SATPLAN and BlackBox use compilation into a Boolean satisfiability problem, ILP-PLAN into an integer linear program, and GP-CSP compiles into a constraint satisfaction problem. Compilation-based planners achieve relatively good performance by using fast, often randomized, algorithms for solving the problems they compile into. Moreover, such algorithms permit additional restrictions on the solution, e.g. particular operators or combinations of operators can be prohibited, cost functions introduced. On the other hand, compilations like that of SATPLAN have very high memory requirements.

Compilation-based planners that use numeric optimization problems such as ILP

Class	**Search direction**	**Advantages**	**Disadvantages**
Refinement	Means-ends analysis	Small memory requirements Natural hooks for replanning	Slow
Progression	From initial state	Fast Easy reasoning about complex effects	Sensitive to the size of the search space
Regression	From goal	Effectively cuts the relevant portion of the search space	Hard to support complex interactions
Compilation	N/A	Natural extension for resource planning Natural support for optimization	Big memory requirements Hides the problem structure

Table 2.1: Advantages and disadvantages of the four planning techniques for construction of applications.

as their target formalism provide natural support for resource planning and optimization, which is highly desirable for construction of applications. On the other hand, the compilation process completely hides the structure of the original problem from the solver, making it very hard to extract information for monitoring and replanning. It also precludes use of heuristics based on the problem structure in solving the target problems.

Table 2.1 summarizes features of the four classes of planners relevant to solving the ACP. In all four groups, the algorithms that work with ground instances of operators are faster than those working with lifted operators (i.e. those containing variables), even though the space searched by the latter is often smaller. Some researchers have suggested [99, 75] that it might be possible to make the lifted operators planners work faster by extracting information about possible variable bindings from the problem.

2.2.4 Scheduling

Scheduling focuses 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 (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 [67] and [73] describe computation of *resource envelopes* for scheduling problems with constant changes of resource levels. Both resource envelopes and temporal networks [23] use graph-theoretic algorithms to prune the search space.

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 [36] uses forward chaining to cope with sequence-dependent and non-reversible resource functions.

Contemporary schedulers are capable of solving large problem instances. The scheduling problem is NP-hard, and exact methods have limited scalability due to the combinatorial complexity of the search. Therefore, schedulers often use randomized algorithms [7, 17], which allow them to obtain high quality (although sub-optimal) solutions fast.

2.2.5 Planning with resources

Many real world planning problems require support for numeric (*metric*) resources. Metric planners can be divided into three groups.

Extensions of classical planners

Several metric planners are relatively straightforward extensions of classical planners.

Zeno [78] is a least commitment planner. It supports metric preconditions and effects, as well as deadline goals. The logical part of Zeno is similar to UCPOP. Zeno uses Gaussian elimination and the Simplex algorithm to solve linear equalities and inequalities, and delays processing of non-linear expressions until they are linearized.

RIFO [62] is an extension of IPP [64], which is a planning graph-based algorithm. In parallel with the planning graph, RIFO builds a resource time map (RTM) — a data structure that contains optimistic boundaries for resource values at each time step. RTM is similar to resource envelopes of scheduling, but it reflect logical, rather than temporal, flexibility of a solution. During the plan extraction phase, the goal resource intervals are propagated backwards along with the logical goals. Finally, to ensure correctness of the plan, symbolic execution of the found plan is performed. The main disadvantage of this approach is that to allow the algorithm to propagate the resource goals during the plan extraction phase, the form of resource effect expressions is very restricted (only constant deltas or linear equations with at most one other resource variable are allowed).

The compilation approach seems very attractive for planning with resources. However, the need to merge both logical and resource expressions in one compilation makes the compilation process complicated. LPSAT [95] combines a SAT solver with an incremental Simplex system called by the SAT solver. TM-LPSAT [86] extends that work to support external processes and durative actions. ILP-PLAN [57]) compiles the planning planning problem (both logical and metric parts) into an integer linear program. All three systems are limited to linear expressions in preconditions and effects.

Heuristic planners

Several modern high-performance planners, such as FF [48, 49] and SAPA [24] use solutions to *relaxed* planning problems to drive search. The relaxation usually involves ignoring delete effects and negative preconditions of actions. Such relaxed problems can be solved in polynomial time.

The solution to a relaxed problem is then used in the next *phase* of the algorithm to guide the search for the original problem. Bonet and Geffner [11] point out that GraphPlan-based algorithms can be viewed as a special case of heuristic planners.

Constraint-based interval planners

Several planners targeted for the aerospace domain (IxTeT [41], HSTS [54], ASPEN [80]) use the constraint-based interval model of the planning problem [87, 53]. In this model, both propositions and actions are modeled as *tokens* on *timelines* corresponding to objects.

This formalism naturally allows for specification of external events, such as change in resource availability. Numeric constraints can be implemented using resource envelope techniques. While traditional AI planners offer limited support for numeric expressions [35, 26], the CBI formalism allows for very expressive resource models [5]. The disadvantage of CBI planners that they are not well suited for open worlds (all available objects are considered by the planners) and usually rely on domain-dependent search control to achieve good performance.

2.3 Example problems

In this section we present example applications that we use throughout this thesis. The applicability of our techniques is not limited to these domains. We have chosen these applications because they exhibit properties, which, we believe, are characteristic of component-based systems in general.

The first two applications are mail and webcast applications that can be instantiated using a dynamic component-based system such as CANS or PSF. We also present a simple application that involves generation and scheduling of workflows on computational grid infrastructures such as Globus Grid.

2.3.1 Mail application

This application is a component-based security-sensitive mail service, originally introduced in [51]. 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` and `Decryptor` pair to protect message privacy. Similarly, the `ViewMailServer` can serve as a cache to overcome links with low available bandwidth.

The mail application uses the request-reply model of interaction between components. Components expose *interfaces*, whose methods can be invoked by other components. In the mail application, the `MailServer` component exposes (implements) `MailServerInterface` (MSI), against which `MailClient` makes requests.

Types of the required and implemented interfaces describe qualitative compatibility of components. To model quantitative compatibility of components and network resources, such as resource requirements and quality-of-service characteristics of the application, we define numeric interface properties, e.g. supported request rate.

Figure 2.1 shows the abstract structure of the mail application, which describes all possible configurations of the application. Rectangles correspond to component types, ovals represent interfaces. In the mail application, each component invokes and exposes at most one interface. Therefore, 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 2.2 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.

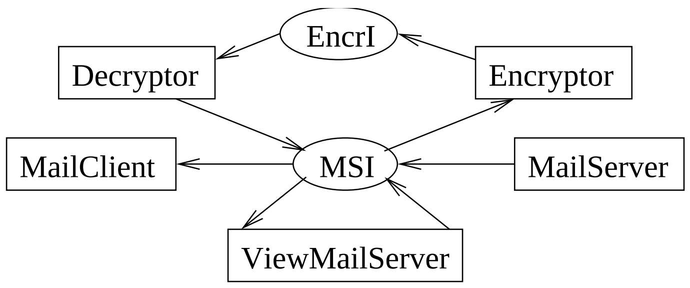


Figure 2.1: Abstract structure of the mail application. Rectangles represent components, and ovals represent interfaces.

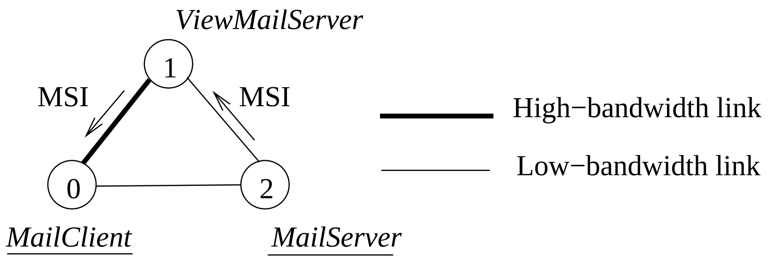


Figure 2.2: Component deployment of the mail application.

2.3.2 Webcast application

The second application models a webcast scenario (Figure 2.3), where a server provides a combined media stream consisting of images and text, which needs to be delivered to a client.

The webcast application uses the publish-subscribe model of component interaction. In the beginning of a session, the client subscribes for a data stream with particular QoS properties, such as minimum frame rate. After that, the server pushes the data stream towards the client until the session is terminated. Although this mode of communication is different from the request-reply mode used in the mail application, we use a unified scheme described in section 3.2 to model both kinds of applications.

The frame rate requested by the client 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 2.3 shows an example of such injection involving `Splitter`, `Merger`, and text 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 host 7 produces a media stream, and the `Client` on host 0 wants to consume this stream with a particular frame 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

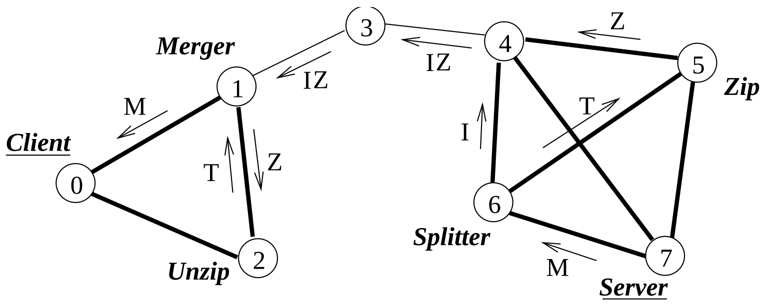


Figure 2.3: The webcast application

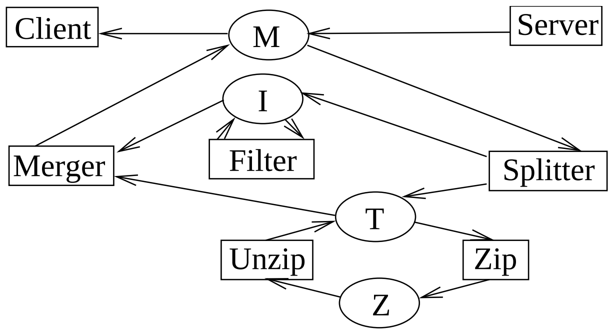


Figure 2.4: Abstract structure of the webcast application. Rectangles represent components, and ovals represent interfaces.

reverse transformations there.

Figure 2.4 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 structure.

2.3.3 Grid planning

The second variety of the application configuration problem we will consider is generation and scheduling of workflows for computational grids [20].

A grid application consists of jobs that process files. Unlike the previous ex-

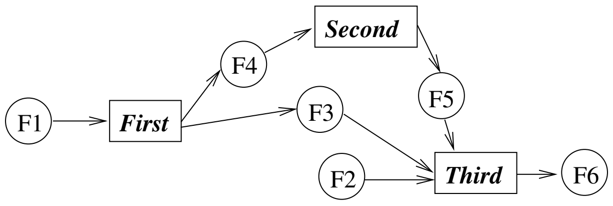


Figure 2.5: Abstract structure of a simple grid application. Boxes represent jobs, and circles represent files.

amples, some of jobs constituting a grid application may need to run sequentially. Therefore, grid planning requires a more sophisticated resource model. In particular, resource availability is represented as a (piecewise constant) function of time, and an application configuration contains explicit resource reservations.

To make this more concrete, let us first present a small synthetic example.

Simple example. The application shown in Figure 2.5 consists of three instances of different component (job) types. The *First* job requires file *F1* and produces files *F3* and *F4*. The *Second* job requires file *F4* and produces *F5*. The *Third* job requires files *F2*, *F3*, and *F5* and produces *F6*.

The network consists of four hosts (Figure 2.6). File *F1* is initially available on *host2*, and *F2* on *host1*. The files are available there from the very beginning. In principle, the availability time of files may be greater than zero, for example, when another workflow currently being executed is expected to produce the file at a known (planned) moment in the future.

The goal in grid planning is to make an instance of a given file available on a given network host as quickly as possible. In our example, an instance of *F6* should be obtained on *host3*.

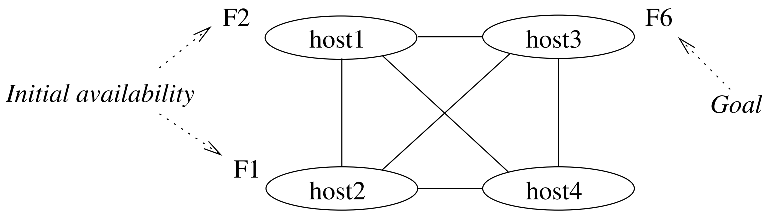


Figure 2.6: Network for the simple grid application. The initial and goal locations of files are shown next to the hosts.

As in our previous examples, the grid application consists of component executions connected by data transfers. The main difference from the dynamic component-based applications discussed in the previous sections is that in the grid applications different jobs and data transfers can be executed sequentially. Therefore, the planner needs to reason about physical time in addition to network resources.

For each job and each network host, the possible start time and duration of execution of the job on the host depends on resource availability on the host and job's resource requirements. In our example, we assume that the resource availability is all-or-nothing, so that the job duration is a constant for each host. Job *First* can run on *host1* starting at time points 2 and 12. Job *Second* can run on *host4* starting at 10, 25, or 35. Job *Third* can run on *host1* starting at 5, 25, 45, or 65. For every job-host pair, job execution takes 10 time units.

Files can be transfered over network paths consisting of links. Duration of the transfer depends on the size of the file and the current available network bandwidth. In our simplified example, any data transfer takes 10 time units to complete.

Figure 2.7 present a configuration (an execution plan) for our simple grid planning problem. The completion time of this plan, i.e. the earliest availability time of *F6* on

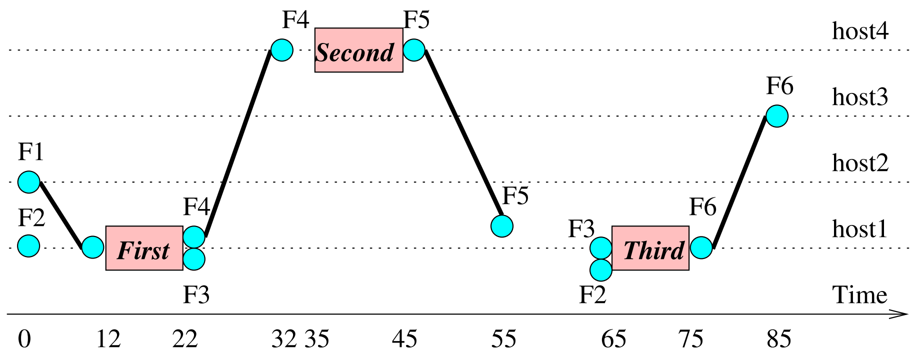


Figure 2.7: Example plan with completion time 85. Horizontal lines represent hosts. Circles show existence of a file on a node (the first time the file appears and the moment when it is required). Rectangles correspond to execution of a component. Solid lines are data transfers.

host3 is 85.

The earliest time component *First* can be executed is 12, because the required file *F1* needs to be transferred from *host2*. The data transfer takes at least 10 time units. Similarly, component *Second* can start executing on *host4* at time point 35, and component *Third* is scheduled to run on *host1* at time point 65. No data transfers are scheduled for files *F2* and *F3*, because they are already available on *host1*. The final step of the plan is the data transfer of file *F6* from *host1* to *host3*.

Realistic example. In practice, grid workflows may involve hundred of jobs and files. For example, Figure 2.8 shows an instance of a Montage workflow [6] with 57 jobs and 108 files. The goal of this application is to construct a map of a particular region of the sky out of pieces. Achieving this goal requires several stages of processing, including projecting original images on the required plane, computing differences between projected pieces, calculating background model, adjusting the images, and

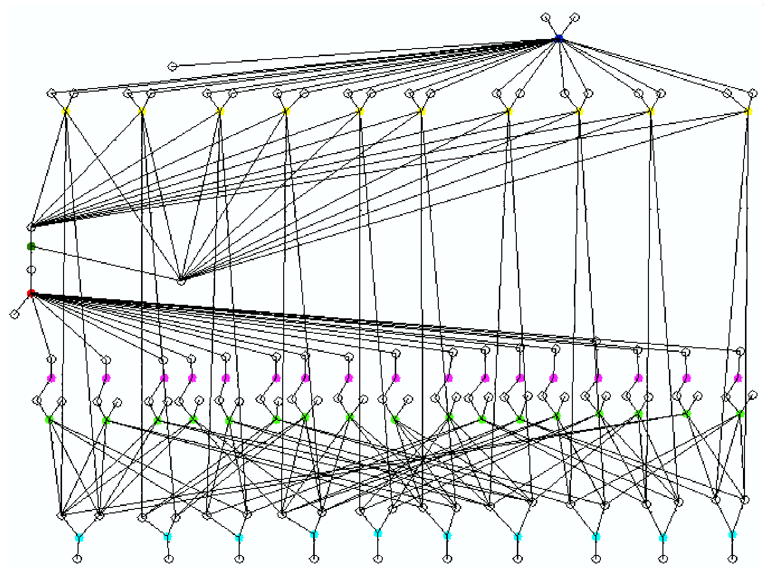


Figure 2.8: Montage workflow. Empty circles represent files, with the source files shown in the bottom of the figure. The filled circles represent jobs. The goal is to obtain the data files shown in the top of the figure.

putting together the final mosaic.

Some of the intermediate data products may be already available, or be scheduled to be produced, in the network, for example, as a by-product of another workflow. The choice of whether to recompute such data, which requires host resources, or to reuse the existing files, which requires link resources for data transfer, depends on the current resource availability.

An important distinction between the dynamic component-based applications presented earlier and grid applications is the size of the network. Grid applications often require large amount of resource, typically available only in supercomputer sites or at large cluster farms, and therefore the total number of hosts that may participate in a grid application configuration is usually small. Figure 2.9 shows a simple net-

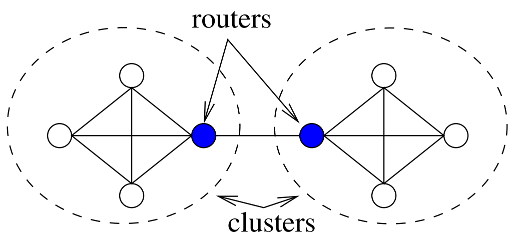


Figure 2.9: Example network for grid applications.

work consisting of 6 supercomputers. The hosts are organized in two clusters. All communication between the clusters uses a shared link, which is modeled using two specialized router hosts. Routers are artificial hosts, which can not perform computation or store data.

2.4 Summary

In this chapter we discussed example component-based systems that need to solve the ACP.

Existing systems, both dynamic component-based systems and computational grid frameworks, address the complexity of the ACP by putting additional restrictions on the problem model. In this thesis we show that it is possible to efficiently solve the ACP without sacrificing expressiveness of the model.

Section 2.2 gave an overview of AI planning. Metric AI planners can handle very expressive models. However, none of the existing planners we found was able to efficiently handle the ACP (see Appendix A).

In the next chapter we present a detailed model of the ACP, analyze complexity of this problem, and identify challenges that a planner for the ACP needs to address.

Chapter 3

The Application Configuration Problem

In this chapter we describe the general model of the ACP, the architecture used for integrating the planner with the underlying component-based framework, and the assumptions we make about the services provided by the framework. We also introduce a model for a variation of the ACP for the dynamic component-based systems called the Component Placement Problem. We show how the CPP (and the ACP in general) can be represented as a planning problem. We analyze the computational complexity of the CPP and reasons for poor performance of existing planners. We conclude this chapter by summarizing challenges that need to be addressed to design a scalable and efficient planner for the CPP/ACP.

3.1 General model of ACP

The **ACP** is a problem of constructing a configuration of a component-based application given specifications of environment, components, and user goal. This section describes these specifications.

3.1.1 Environment

The **environment** is a wide-area network consisting of hosts (computational nodes and routers) and links. Usually, they are abstractions for the physical hosts and links in the network that capture the dominant bottlenecks. These hosts/links may or may not correspond to physical resources of the network. For example, the fact that all communication between two geographical regions shares a particular network path can be captured by a topology that involves two (artificial) routers connected by a single (artificial) link.

In addition to the network topology, hosts and links may have numeric **properties** associated with them as described in Section 3.1.3.

3.1.2 Components and data

To reason automatically about valid application configurations, one needs to formalize the notion of linkage compatibility between components. To achieve a high degree of interchangeability and reusability of components, the linkage compatibility information needs to be local, as opposed to application-wide.

In our model, **components** are described as independent executable modules, whose intended functionality is completely captured by the *types* of input and out-

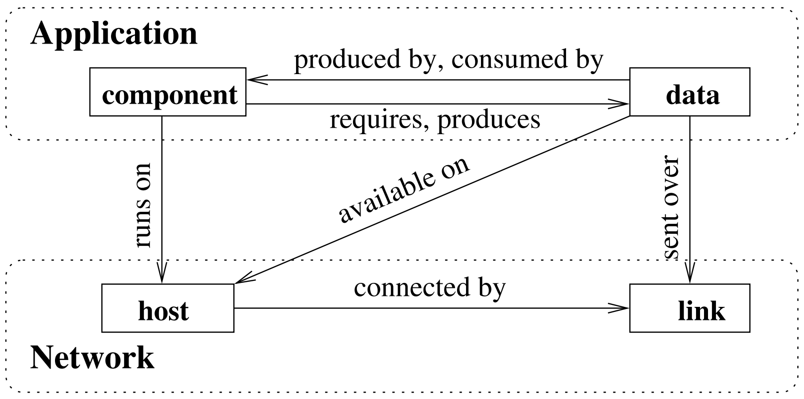


Figure 3.1: Objects of the application configuration problem.

put data (Figure 3.1). In this thesis, we consider two kinds of **data**: (i) files and (ii) interfaces. An interface can be used to make calls against it (e.g. a mail server interface for sending and receiving messages), or it can be viewed as a provider of a continuous stream of data (e.g. a video interface).

A component can be *deployed* (executed) on a network host.

Data can be *produced* and *consumed* by components and *transfered* over network links. A data item (file, interface) can be consumed by a component running on a network host if the data is *available* on that host. A data item becomes available on a host when it is produced by a component running on that host or is transfered over the network from another node where that data item is available.

Each component *requires* a set of data items to be present on a host before the component can be executed/deployed on that host. These data items are to be *consumed* by the component. Consumed data items remain on the host and can be used by other components. However, the numeric properties of the consumed data may change as a result of component execution. For example, the sharing of a mail server by several clients can be modeled by decreasing the maximum supported request rate of the server after connecting a new client to the server's interface.

The set of required data items constitutes a component's **qualitative requirements**.

3.1.3 Numeric properties

There may be **numeric** (real-valued) **properties** associated with network links, network hosts, and data items available on the hosts. For examples, one might consider link bandwidth, host memory, and request rate supported by an interface.

Properties of data on hosts are determined by the properties of the data transfer or the component execution that created that data.

Properties of links can be changed by data transfers. For example, link bandwidth can be consumed. Similarly, properties of hosts can be affected by executing components.

Numeric properties of hosts and available data can be used to specify a component's **quantitative requirements**. For example, a component may require a particular amount of available memory to process an incoming video stream of a certain bandwidth (resource consumption requirement), or a client component may require the mail interface to support a given minimum request rate (client's Quality-of-Service requirement).

3.1.4 User goal

We consider two types of user **goals**. A user might request that an instance of a given component type (user's client) be running on a given network host (user's host).

Deployment of a client component on a user's host may require satisfying the component's qualitative and quantitative (QoS) requirements, which in turn may require deployment of additional component instances and data transfers.

Alternatively, a user may request that a particular data item (file) be present on a particular network host. Achieving this goal may require performing computations (executing components) to produce this data item and transferring files over the network. This kind of user goal is often found in computational grids.

In both cases, the user may have **budget restrictions** (e.g. disk quotas, deadlines) and **metrics** (preferences) (e.g. obtain the data as fast as possible given the budget restrictions).

So far, we have described the goal as a single client placement or data product. In general, the goal may require simultaneous placement of several components on different network hosts (e.g., setting up a video-conference) or producing multiple data products. The algorithms presented in this thesis are capable of achieving such conjunctive goals. However, when discussing modeling and complexity issues, we will limit ourselves to singleton goals for simplicity.

3.1.5 Application configuration

A **feasible application configuration** for a given state of the environment and a given user goal is a set of executable component instances and data transfers such that:

1. Each component instance is mapped onto a network host and scheduled to execute at a certain time.

2. Each data transfer is mapped onto a network link and scheduled to execute at a

certain time.

3. The properties of a data item on a host, including earliest availability time of a file and maximum supported request rate of an interface, correspond to the action that created that data item.
4. Qualitative and quantitative constraints of all components are satisfied.
5. At any given moment, the total resource consumption requirements of all components running on a given host do not exceed the amount of host resources available at that moment.
6. At any given moment, the total resource consumption requirements of all data transfers over a link do not exceed the amount of resources available on that link at that moment.
7. The user goal is satisfied.

Given the complexity of the problem, discussed later in this chapter, even finding a feasible application configuration in an automated way is a hard and important task. However, in practice, one additionally desires that the configuration be a *good* one. The following section makes this statement more formal.

3.1.6 Metrics

To define the notion of *optimality* of configuration we introduce a **cost function** capturing the total resource consumption of the configuration.

To do so, we define a **cost of an action**, where the action is associated with a component execution or a data transfer, as a function of properties of the network resources and data consumed by that action, and a **cost of configuration** as a (non-decreasing) function of costs of all actions in that configuration. This specification of cost allows us to specify *preferences* over resources (see Section 6.1 for details).

The cost of a configuration corresponds to the total resource consumption of the application. Our model also supports the notion of **duration** as an alternative metric. In computational grid applications, duration of a configuration is defined as the completion time of computation, i.e. the earliest moment in time when the user goal is achieved (the required data item is produced and delivered to the client machine). This way, the duration corresponds to the delay of the final data product with respect to the beginning of computation.

In dynamic component-based systems, where all components of a configuration are running simultaneously, the duration is defined as the number of actions on the longest path through the configuration. This way, duration corresponds to the number of transformations seen by a data packet, and has no direct correspondence to the delay with respect to the server producing the data stream. To describe the latter, one may also define a *delay* property of an interface (data stream) with the semantics of accumulated latency with respect to the server. However, this property is not assigned any special significance by the planner, and is treated the same way as, for example, CPU or bandwidth.

3.1.7 Architecture

We assume that a component-based framework has a single centralized instance of a planner. The information about the current state of the environment and requirements of components is collected and stored by external services. The planner can access this information by making calls against these services. We assume that information provided by external services is complete and correct, and access to this information is free and instantaneous.

3.1.8 Correctness of information

We assume that if an external service reports that a particular amount of resource is available, the reported amount of resource can be consumed by a new application configuration. Detection and processing of failures during resource allocation is outside the scope of our model and needs to be handled by external means.

We assume that planning for different clients is done sequentially. This means that if another application is concurrently using the resources required by a configuration being designed, the resource consumption of such an application is already reflected in the numbers reported by external services. A new application configuration cannot affect resource behavior of already running applications. In Chapter 8 we discuss an extension to this model.

3.2 Component Placement Problem

As we mentioned before, the ACP in general is computationally hard, and it is necessary to choose planning techniques to take advantage of the natural restrictions of each ACP variety to achieve good performance. In this section we describe the **Component Placement Problem** (CPP), which is a variety of the ACP that is encountered in dynamic component-based systems such as the Partitionable Services Framework [51]. The goal of the CPP is to find a snapshot configuration of a component-based application, in which components process continuous streams of data. Chapter 7 discusses a variety of the ACP encountered in computational grid frameworks. The grid ACP deals with workflows that process files and requires reasoning about physical time in addition to network resources.

The CPP has the same structure as the general ACP. The problem specification describes:

- A network environment as a graph of links and hosts with numeric properties associated with them.
- A set of interface types that correspond to typed data streams. An instance of an interface on a node may have a set of numeric properties.
- A set of component types specified by sets of consumed and produced interface types and numeric formulas describing resource requirements and effects of component deployment.
- A user goal as a pair of a component type and a network host on which an instance of the given type needs to be deployed.

In general, dynamic component-based applications can realize a *publish-subscribe* or a *request-reply* model of interaction between components. In the former case, properties are propagated from the required to the implemented interfaces, e.g., the bandwidth of the incoming data stream directly affects the bandwidth of the produced data stream. In request-reply applications the dependency is reversed: the request rate of the required interface depends on the request rate of the implemented interface.

For simplicity, we use the forward propagation model to describe both types of interaction. To do so, we add a set of interface properties describing the upper bounds on the supported request rate and response size. This is an approximation, which may lead to over-reservation of resources. However, as we describe in Section 6.2.2, our algorithm still produces good results.

The major restriction of the CPP compared to the general ACP is the absence of the time aspect. The CPP is concerned with snapshot configurations, which means that the planner does not need to reason about sharing network resources in time. On the other hand, a planner for the CPP still needs to deal with the scale of the problem and with the non-reversibility of resource functions.

3.2.1 Assumptions

To design an efficient algorithm for the CPP, we make several additional assumptions, which naturally hold for this variety of the ACP.

First, we assume that once an interface becomes available on a host, it does not disappear from it, and components may require only *presence* of an interface on a host, not the *absence* of an interface. This allows us to simplify the translation of the CPP into a planning problem. Note that, in the presence of the numeric resources, this restriction does not cause loss of generality, since the absence of an interface can be modeled, for example, by setting its bandwidth to zero.

Second, we assume that instances of the same interface type on a host are separated from each other by at least one component instance. For example, it is possible to have two instances of `MailServerInterface` on a host if they are separated by a `ViewMailServer`, i.e. the `ViewMailServer` component consumes one of the instances and produces the other. On the other hand, it is not allowed to have a

`ViewMailServer` component produce (or consume) more than one data stream of type `MailServerInterface`. This assumption is necessary to distinguish between different streams at each stage of processing.

Third, we assume that all functions used in describing component and link crossing behavior are *monotonic* and *forward computable*.

Forward computability means that outputs of an action, e.g. memory requirements of a component or link bandwidth of a link crossing action, can be efficiently computed given properties of input streams, and initial values of properties of hosts and links. If a value of a forward computable function is not defined for a given set of arguments, an exception is raised, and the planning action that requested the function evaluation is pruned from the search.

Although we require all functions to be easily computable in the forward direction, their reverse computability is not required. For example, there may be no easy way to compute the required bandwidths of two input streams being merged given the value for the bandwidth of the combined output stream.

We also require functions to be **monotonic**. The purpose of this restriction is to enable computation of intervals of function values given intervals for its parameters. In case of monotonic functions, it suffices to compute only a finite set of function values to obtain the interval for the image.

$$\begin{aligned} f([m_1, M_1], ..., [m_n, M_n]) &= [\min S, \max S], \\ \text{where } S &= \{f(b_1, ..., b_n) | b_i \in \{m_i, M_i\}\} \end{aligned} \tag{3.1}$$

In practice, the monotonicity requirement can be easily extended to a weaker requirement of *interval-computability* (see Figure 3.2). In the rest of this thesis, when-

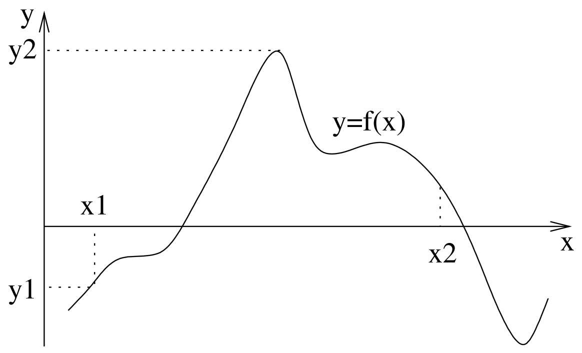


Figure 3.2: Interval-computable function. $f([x1, x2]) = [y1, y2]$.

ever a monotonic function is required, an interval-computable function can be used.

3.2.2 Metrics

In Chapters 4-6 we describe three versions of our algorithm for the CPP that use different sets of techniques. The chosen set of techniques affects the expressiveness of the model the planner can handle, including the kinds of metrics the planner can optimize.

The first two versions minimize the *parallel length* of the plan, i.e. the number of hops, including components, on the longest path from the server(s) to the client. Intuitively, this means that application configurations with the smallest number of component instances and used links are preferred. If processing of a data packet by each action has the same duration, the parallel length of the plan corresponds to the total delay with respect to the server.

The third version of the planner, presented in Chapter 6, aims to minimize the total *cost* of the plan. As we discussed in Section 3.1.6, the cost of the plan is a non-decreasing function of costs of actions in the plan (a sum in all our examples), and the cost of an action is a function of its resource consumption. As with all functions in the CPP, we require cost functions to be monotonic and forward computable.

3.2.3 Compilation into a planning problem

The CPP can be viewed as an AI planning problem with numeric resources:

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

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

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

Figure 3.3 shows a PSF-style description of the `ViewMailServer` component. This component acts as a cache for a mail server, and requires and implements `MailServerInterface`. An instance of the `ViewMailServer` can be placed on a network host if that host has sufficient number of available CPU cycles to process the incoming requests ($MSI^i.NumReq$). The number of requests forwarded by the `ViewMailServer` is proportional to the number of incoming requests. In addition, the maximum number of incoming requests is limited, and the required interface should be secure for the `ViewMailServer` to use it. The *Effects* section of the component description describes properties of the implemented interface and the resource consumption of the component.

Figure 3.5 presents the planning operator corresponding to placing a `ViewMailServer` component on a network host. The preconditions of this operator result from the resource requirements of the component and the fact that `MailServerInterface` (MSI) is a required (consumed) interface. The effects come from the effects section of Figure 3.3, with `MaxReq` providing the upper bound on the `NumReq` parameter of the implemented (produced) interface.

An operator schema (parameterized operator) has the following sections (line numbers refer to the code fragment in Figure 3.5):

- 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 monotonic forward-computable functions that return Boolean values (lines 3-6);

$<$**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**$>$

$MSI^i.Sec := True$

$MSI^i.Trust := Node.Trust$

$MSI^i.ReqSize := 1000$

$MSI^i.NumReq := MIN(MSI^r.NumReq/MSI^i.RRF,$

$MSI^i.MaxReq, Node.NodeCPU/MSI^i.ReqCPU)$

$Node.NodeCPU := Node.NodeCPU - MSI^i.NumReq * MSI^i.ReqCPU$

VMS = ViewMailServer, MSI = MailServerInterface.

Superscripts r and i indicate required and implemented interfaces.

Figure 3.3: Component description.

<**Interface** name = MSI >

<**Crosslink**>

$MSI^d.Sec := MSI^o.Sec\ AND\ Link.Sec$

$Link.BW := Link.BW - MIN(Link.BW, MSI^o.NumReq * MSI^o.ReqSize)$

$MSI^d.NumReq := MIN(MSI^o.NumReq, Link.BW/MSI^o.ReqSize)$

$MSI^d.ReqSize := MSI^o.ReqSize$

MSI = MailServerInterface.

Superscripts o and d correspond to interfaces at link origin and destination.

Figure 3.4: Interface description.

- 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 8-13).

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.

```
plVMS(?n: host)
PRE: avMSI(?n)
     cpu(?n) > MSIMaxReq*MSIReqCPU
     numReq(MSI,?n) > MSIMaxReq*MSIRRF
     sec(MSI, ?n) = True
     trust(MSI, ?n) > 5
EFF: avMSI(?n), plVMS(?n)
     numReq(MSI, ?n) := MIN(numReq(MSI, ?n)/MSIRRF,
        MSIMaxReq, cpu(?n)/MSIReqCPU)
     cpu(?n) := cpu(?n) - numReq(MSI, ?n)*MSIRRF/MSIReqCPU
     sec(MSI, ?n) := True
     trust(MSI, ?n) := ntrust(?n)
     reqSize(MSI, ?n) := 1000
```

Figure 3.5: Planning operator for placing a `ViewMailServer` component on a host.

3.3 CPP and AI problems

As we have shown in the previous section, by introducing operators for component placement and link crossing, the CPP can be compiled into a planning problem without negative logical preconditions and effects, but with very general resource functions.

The computational complexity of planning is high [28]. Moreover, planning with numeric state variables is undecidable [47]. To justify the use of AI planning for a seemingly simpler problem (CPP), we will show that (PSPACE-complete) propositional STRIPS planning can be reduced to the CPP. We will also prove that the CPP, like metric planning, is undecidable in general.

A secondary role of the STRIPS-to-CPP reduction discussed below is to demonstrate the relationship between the network space and physical time, which makes it possible to use AI planning for solving the CPP.

3.3.1 CPP and STRIPS planning

Intuitively, a state of the world in traditional AI planning corresponds to a state of an interface in the CPP. The following reduction makes this statement more formal.

A propositional STRIPS planning domain is described by a set of Boolean variables $P = \{p_i\}$, a set of operators $A = \{a_j\}$, where each operator is defined by a list of preconditions (a conjunction of a subset of the variables or their negations), an add list (a set of variables that become true as the result of the action execution), and a delete list (the set of variables that become false). A STRIPS planning problem is defined by a planning domain, an initial state (a complete truth assignment to all variables), and a goal state (a partial truth assignment). The goal of the planner is to find a (totally ordered) sequence of operators that when executed in the initial state brings the system to a goal state.

A propositional STRIPS problem is identical to the following CPP problem. The network contains only one host, and no resources are associated with this host. There is only one interface type with $|P|$ properties. Initially, this interface is available on the host. For each variable p_i, the value of the corresponding property is 1 iff p_i is true in the initial state. For each operator a_j, the CPP contains a component that consumes and produces the interface. The placement precondition of this component is a conjunction of equalities corresponding to the operator's preconditions. The effects of the component placement are assignments of 0 or 1 to some properties of the interface (in accordance to the add and delete lists) and identity assignments to the rest of the properties. The identity assignments correspond to the frame assumption in STRIPS that properties not in the add or delete list remain unchanged. An additional compo-

operator	**precondition**	**add**	**delete**
opA	none	p	none
opB	none	q	p
opC	$p \wedge q$	g	none

Figure 3.6: A propositional STRIPS problem.

nent C_{goal} is constructed with placement preconditions generated from the goal state description. The goal of the CPP is to place C_{goal} on the host.

To illustrate this reduction, consider the following problem (Figure 3.6). The world state is described by three Boolean variables p, q, and g, all of which are initially false. There are three actions, whose preconditions and effects are shown in the figure. The goal is to achieve g.

This problem maps to the following CPP (Figure 3.7). Initially, there is interface Int available on the host 0 with three properties set to 0: $Int.p(0) = 0, Int.q(0) = 0, Int.g(0) = 0$. There are four component types, each of which consumes and produces interface of type Int. The resource requirements and effects of the components are given in Figure 3.7. Values on the left side of the effect assignments refer to the produced data stream, and those on the right side of the assignments to the consumed stream. The goal is to place C_{Goal} on host 0.

The solution for this CPP is to place a chain of components $compB$-$compA$-$compC$-C_{Goal} on the host. This CPP solution corresponds to the sequence of operators opB, opA, opC, which is a solution for the original STRIPS problem.

The above reduction shows how a propositional planning problem can be cast as a CPP. This reduction requires only a small part of the expressive power of the CPP. Only constants are used in requirements and effects of components, while the CPP allows for more general functions. The presence of general numeric functions in-

component	**requirements**	**effects**
compA	none	$Int.p(n) := 1$; $Int.q(n) := Int.q(n)$; $Int.g(n) := Int.g(n)$
$compB$	none	$Int.p(n) := 0$; $Int.q(n) := 1$; $Int.g(n) := Int.g(n)$
$compC$	$Int.p(n) = 1$, $Int.q(n) = 1$	$Int.p(n) := Int.p(n)$; $Int.q(n) := Int.q(n)$; $Int.g(n) := 1$
C_{Goal}	$Int.g(n) = 1$	$Int.p(n) := Int.p(n)$; $Int.q(n) := Int.q(n)$; $Int.g(n) := Int.g(n)$

Figure 3.7: Compilation of propositional STRIPS problem into a CPP.

creases the computational complexity of the problem. Propositional STRIPS planning is PSPACE-complete [13], and the CPP is undecidable (see Section 3.3.3).

3.3.2 Network space and physical time

The STRIPS-to-CPP reduction presented in the previous section illustrates the main insight that allows us to use AI planning for solving the time-free component placement problem, namely, the correspondence between physical time and network space.

The planning problem resulting from compiling a CPP into a planning problem can be viewed as a "normal" planning problem with numeric resources. This problem has discrete time steps corresponding to evolution of a data packet through a data path (e.g. processed by a component, *then* transmitted to another node over a network link). In the model of the CPP described in Section 3.2, the exact delay of a packet from its origin is viewed as one of the properties of the interface (similar to bandwidth), and the problem specification can define any function for manipulating delays. However, the definition of the CPP does not require existence of the delay, therefore the core CPP problem is *timeless* [58].

Note, however, that the transformation of a data stream by components and links along a data path is similar to transformation of a world state by operators along the time line. Different data paths can affect each other via shared network resources just as operators in classic planning interact via shared variables. For example, in the CPP it is possible to put two components processing different data streams onto the same network host. This situation corresponds to parallel execution of operators in traditional planning. In both cases, if the sum of resource requirements of components/operators exceeds the available amount of resources, a conflict occurs. Techniques for resolving conflicts in classic planning, such as promotion and demotion [79], correspond to changes in a data path in the CPP, e.g. by sending a data stream to another host. Even though the CPP does not have an explicit time component, it can be considered *"planning in space"* similar to the traditional *"planning in time"*.

There are, however, important differences. The time considered by traditional planning is homogeneous. A logical state enables the same set of operators regardless of at what point along the time line this state occurs. In the CPP, different hosts and links have different properties. Therefore, different sets of components may be deployed on different hosts even if exactly the same data streams are available on these hosts.

Moreover, time in classical planning is linear. For each step there is exactly one step immediately preceding it and one step immediately following it. On the other hand, the network in the CPP is a graph. There are several paths leading to and from each host.

Finally, the network space is always discrete. Physical time, depending on the planning model, can be discrete or continuous.

3.3.3 CPP and metric planning

In Section 3.2.3 we showed how the CPP can be compiled into a STRIPS planning problem with numeric state variables. In [47] Helmert shows several undecidable classes of such planning problems. In particular, the plan existence problem is undecidable if operator preconditions contain comparisons of numeric variables to zero and numeric effects of operators contain increments and decrements (i.e. $v = v + 1$ and $v = v - 1$).[1]

This planning problem can be reduced to the CPP using the same technique we used in Section 3.3.1: by creating a CPP component type for each planning operator and a separate component type for the goal. Translation of numeric preconditions and effects is trivial. This reduction proves that, in general, **the CPP is undecidable**. Since the CPP is a *simplified* special case of the ACP, this result also implies undecidability of the ACP.

Note that this undecidability result holds because of the unbounded size of the search space. The reason for this is that, although the network topology and the set of component types are finite, the CPP does not specify an upper bound on the number of components that can be placed on a host or the number of streams that can be sent over the same network link.

However, in reality, every action usually consumes at least one non-replenishable

[1]In [47] this problem is referred to as PLANEX-$(\mathcal{C}_\emptyset, \mathcal{C}_0, \mathcal{E}_{\pm 1})$.

resource. For example, sending a data stream over a network link consumes link bandwidth, and a component placed on a network host consumes CPU cycles. Because of this consumption of non-replenishable resources, in reality, the total number of actions that can constitute a plan is limited.[2] This means that, in practice, it may be possible to determine existence or non-existence of a solution for the CPP in finite time.

3.3.4 CPP and scheduling

While AI planning is usually concerned with finding a feasible sequence of operators that achieve the goal, the problem of scheduling is to find an optimal ordering of a given partially ordered set of tasks. Since the CPP is also concerned with optimality, we would like to discuss the relationship between the CPP and scheduling.

In the CPP, a data stream may be forced to "jump" to a new host because of limited resources at the source host. Hosts and links in the CPP correspond to resources in scheduling terminology.

In scheduling, the problem is to allocate a given set of resources to a given set of tasks. In the CPP the set of components constituting an application configuration is not fixed. A planner may decide to use a caching component to deal with a low-bandwidth path, a pair of zip/unzip components, or choose a longer network path. This need for choosing an action and the fact that the plan length cannot be computed in advance prevent us from directly using scheduling techniques for solving the CPP.

In scheduling, it is often trivial to find a feasible solution by stretching the schedule

[2]Assuming that there is some minimum bound on the resource consumption, so that the system does not experience Zeno's paradox.

over time. In the CPP many resources, such as network bandwidth, are consumable. Stretching the component DAG over the network simply replaces one resource conflict with another, and often does not lead to a solution. Instead, one needs to change the DAG itself, i.e. the set of components participating in the deployment. The problem of finding a feasible solution for a CPP requires solving the plan existence problem in the presence of resource constraints, which is undecidable in general. Therefore, the CPP is computationally harder than traditional scheduling.

3.4 Challenges

Although the CPP can be presented as a STRIPS planning problem with numeric resources, existing algorithms supporting the STRIPS formalism are not well suited for solving the CPP. Appendix A gives evaluations of the performance of several state-of-the-art planners on the CPP and the grid planning variety of the ACP, which shows that the existing planners are not well-suited for solving the ACP. We have analyzed the reasons for poor performance of the existing planners and identified the following three challenges.

First, most of the algorithms supporting numeric resources employ forward chaining. This approach requires complete knowledge of the initial state before the search can start. Satisfying this requirement would mean feeding, among other things, the complete state of the network to the planner, because it is generally impossible to predict which parts of the network will be relevant to the solution. Regression-based algorithms are better equipped for identifying relevant operators and data (e.g., portions of the network), but require reversibility of resource functions. Some combination of techniques is required to address the two issues simultaneously.

Second, in the absence of negative Boolean preconditions and effects, the only negative interaction between operators is via numeric resources. We are not aware of an existing planning algorithm capable of deriving significant search guidance from numeric conflicts. Numeric planners described in literature tend to pose to tight restrictions on the syntactic form of resource functions [49], or use some sort of an envelope over a set of plans for pruning [62]. As our experiments show, resource envelope over a planning graph quickly loses informedness and does not provide significant pruning. New techniques need to be developed for resource-guided search.

Finally, in the CPP, not just a feasible, but an (approximately) optimal solution is desired. Finding an approximately optimal solution in the presence of arbitrary non-reversible monotonic functions also requires development of new techniques.

3.5 Summary

In this chapter we described the general model of the ACP and its specialization for dynamic component-based frameworks — the component placement problem (CPP). We have shown that CPP, and therefore also the more general ACP, are undecidable.

We discussed similarities between the ACP and traditional planning and scheduling problems that permit casting of the ACP as an AI problem. Section 3.2.3 shows how the CPP can be represented as a STRIPS planning problem with numeric resources. In a similar way, a more general ACP, such as the grid ACP discussed in Chapter 7, can be compiled into a planning problem with numeric resources and durative operators [26] (see Appendix A.3 for an example). We also discussed the novel challenges of the ACP that prevent existing planners from scaling well on this problem.

A number of previous projects have considered models for declarative descriptions of application components with the purpose of automated discovery and composition [76, 97, 43, 52, 22, 100, 32]. Some of these models also include specification of execution preconditions and resource requirements [34, 92, 51, 82, 3, 89].

The kind of information a component description needs to provide depends on the use of that information. In this thesis, we consider the problem of automated deployment-time construction of applications so as to satisfy user requirements given the current state of the environment. Achieving this objective justifies the model described in this chapter.

Our model assumes that it is possible to measure the state of the network and to allocate to the application the amount of resources prescribed by the plan. Although the modern Internet does not yet fully provide this kind of facility, we are confident that future research will enable such functionality [102, 96, 31, 33].

Chapter 4

Dealing with large-scale open worlds

In Section 1.3 we gave a list of techniques for construction of efficient planners supporting expressive problem models. In this and the two following chapters we show how these techniques are used in Sekitei, a planner for the component placement problem. Chapter 7 describes application of the same techniques for construction of GPRS, a planner for computational grids.

To better illustrate how each of these techniques works, we introduce them in groups corresponding to the three challenges identified in Section 3.4, which these techniques address. The three groups of techniques correspond to three *versions* of the Sekitei algorithm, referred to as Sekitei 1, Sekitei 2, and Sekitei 3. Each of the versions extends the previous one and improves the performance of the algorithm and/or expressiveness of the supported model.

In this chapter we describe Sekitei 1. This version of the planner introduces the techniques for dealing with large-scale open worlds: on-demand compilation of the problem specification (**T1** in Section 1.3) and regression-progression search (**T2**). We begin by describing these techniques. Then we present the Sekitei 1 algorithm and its evaluation on the mail and webcast applications described in Section 2.3. We conclude this chapter by discussing the benefits provided by the two techniques and reviewing the challenges that are not addressed by them.

4.1 Techniques

Sekitei uses two techniques — on-demand compilation (**T1**) and regression-progression search (**T2**) — to efficiently identify relevant portions of the problem specification during the search. This allows the planner to support complex dependencies between actions, such as numeric dependencies via shared resources, which are hard to deal with using static pruning techniques.

As mentioned in Section 3.1.7, the planner obtains information about the problem and the current state of the world by querying external services. On-demand compilation of the problem specification hides the interactions of the planner with the external services from the search algorithm, making the planner framework-independent.

Regression-progression search provides the means for identifying the relevant portions of the world state in the presence of non-reversible functions. This technique guides the search towards the cheapest solution and relies on an on-demand compiler for the actual construction of operators and data.

4.1.1 On-demand compilation

The first problem a planner for the CPP needs to address is the scale of the problem specification. When constructing a configuration of an application, one may need to

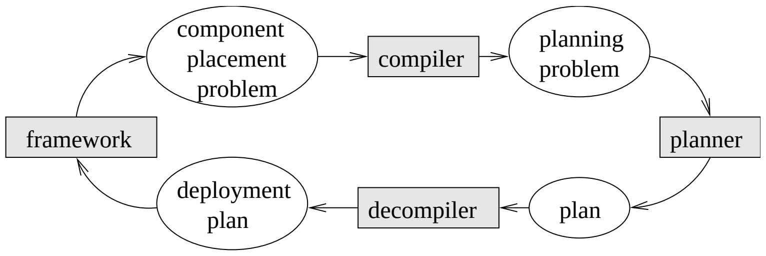


Figure 4.1: Process flow graph for solving CPP

consider thousands of network hosts and hundreds of component types available for deployment even if the final configuration involves only a dozen network hops and a few component instances.

Most existing AI planners assume that the whole problem specification is given as an input, and rely on this fact to perform various sorts of reachability analyses.

Even for small application configurations, obtaining a complete problem specification is often not feasible for the CPP. The planner needs to discover portions of the initial state as they become relevant during the search. Moreover, in the presence of arbitrary numeric functions, reachability analysis does not provide significant pruning.

It is desirable to make the planner module independent of a particular set of services provided by the framework. To achieve this, the planner is cushioned from the framework by framework specific **compiler** and **decompiler** modules (Figure 4.1). The planner communicates with these modules using ground variables, whose semantics is hidden from the planner.

Compiler

The compiler produces two main types of information. First, it returns ground instances of operators that can achieve a given ground logical variable. The preconditions and effects of the returned operators are represented as functions of ground variables and constants. Second, for a given ground variable, the compiler can return its value in the initial state of the world and an optimistic value. (See Appendix B for the complete specification of the Compiler interface required by Sekitei.)

For example, given a request for operators that achieve availability of mail interface on host 0, the compiler may return a ground operator for placing a `ViewMailServer` component on that host — `plVMS(0)`. The precondition formula of that operator uses several variables, including the amount of available CPU on host 0 — `cpu(0)`. The initial value of this variable is the interval $[v, v]$, where v is the actual value of the available CPU on host 0, obtained by querying external services. The optimistic value of the same variable is the interval $[0, v]$, which means that the available CPU can be decreased by application of planning operators, but not increased.

Note that the planner does not need to understand the meaning of the ground variables or ways to access framework services. The compiler is responsible for resolving all semantic issues, including conversion and normalizing of units of measurement.

Decompiler

The plans produced by the planner are sequences of ground operators with dependencies between them, but with no semantics attached. The purpose of the decompiler is to convert these ground plans into a format understandable by the framework.

In case of DCBFs, a plan can be represented by a sequence of ground instances of `pl`<*component*> (<*host*>) and `cr`<*interface*> (<*from*>, <*to*>) operators. Information about logical support is easily 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 host 1 to produce the `MailServerInterface` before this interface is sent over the link to host 0. This information can be represented as a framework-specific deployment plan, which consists of (*component, host*) pairs and linkage directives, e.g. `(VMS,1,MSI,MC,0)` (place a `MailClient` component on host 0 and make it invoke `MailServerInterface` of the `ViewMailServer` component running on host 1).

It is expected that the compiler and decompiler for a given framework share the assumptions about semantics of ground variables and operators. In fact, a single object can implement both interfaces. This way, the decompiler can restore the dependencies between the operators constituting the plan and the actions and objects of the original problem.

4.1.2 Combining regression and progression

The next question is how to organize the search. Section 2.2 gave an overview of four kinds of search techniques traditionally used in planning. Let us briefly review the advantages and deficiencies of each of them with respect to the component placement problem.

Compilation-based techniques naturally support optimization. However, performance of such techniques depends on the performance of the underlying optimization engine, and therefore algorithms based on such techniques are usually limited to linear functions [57, 95, 86]. The CPP requires supporting arbitrary numeric functions, which prevents us from using compilation-based techniques.

Pure means-end analysis planners such as Zeno search for a feasible solution and are hard to extend to support optimization.

The two remaining classes — regression and planning graph based planners — both have properties required for solving the CPP.

The CPP is typically characterized by a small goal and a large initial state specification. Therefore, regression, i.e. search backwards from the goal state, presents a natural approach to dealing with the open nature of the world. The planner repeatedly queries the compiler module for ways to achieve the current state, thus discovering relevant portions of the problem specification during the search. The only information that needs to be provided to the planner from the very beginning is the goal specification.

However, regression techniques are not well suited for numeric planning. In particular, they require that all functions used for describing application behavior be reversible, which does not hold for the CPP.

An alternative strategy is to perform forward search in a way similar to GraphPlan-based algorithms [8, 64]. This approach provides several benefits. First, the main data structure used in such planners — the planning graph — is an *envelope* over all possibly executable sequences of operators. Incremental expansion of this envelope guarantees that the first found solution has the optimal length. Second, since all functions used in the CPP are required to be monotonic and forward computable, it is possible to construct a resource envelope parallel to the planning graph [62]. This helps to identify resource conflicts at earlier stages of search.

The main problem with using the planning graph techniques for solving the CPP is that they use the complete specification of the initial state to construct the planning graph. Even if we assume that it is possible to obtain a complete specification of the initial state, the planning graph becomes prohibitively large and expensive to compute.

Our approach is to combine the regression and progression techniques in such a way as to compensate for their deficiencies while taking advantage of the useful properties of both.

4.2 Algorithm

In this section we describe Sekitei 1 — the version of the Sekitei algorithm that uses on-demand compilation (**T1**) and regression-progression (**T2**) to deal with large scale open worlds and non-reversible functions.

4.2.1 The core algorithm

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 1 algorithm consists of four phases — regression, progression, plan extraction, and symbolic execution. Communication between the phases is shown in Figure 4.2 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. 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 phase guarantees that the found plan is correct.

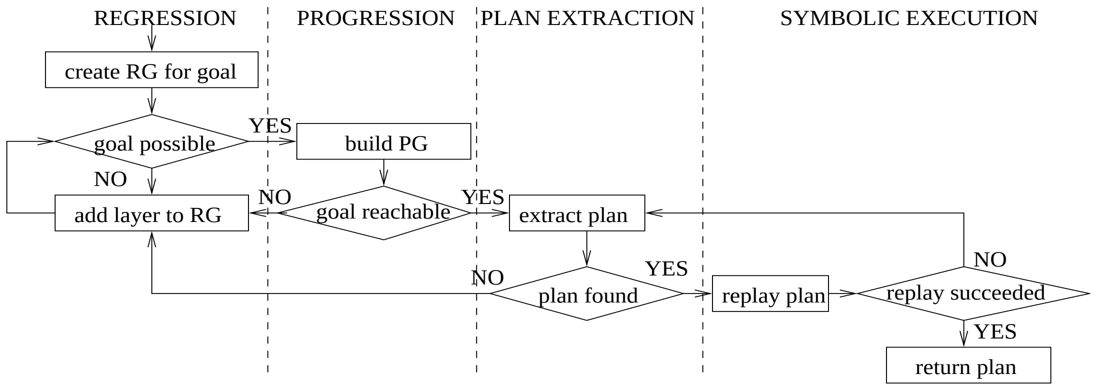


Figure 4.2: The Sekitei 1 algorithm

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 proposition and operator levels, starting and ending with a proposition level, and is constructed as follows.

- Proposition level 0 is filled in with the goal specification.
- Operator level i contains all operators that achieve some of the propositions of level $i - 1$.
- Proposition 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 4.3 shows the RG for the problem presented in Section 2.3.1. In this problem, the only host on which the server can be executed is host 2. The goal is to place client on host 0. Logically, the shortest plan to achieve the goal consists of three operators (bold lines in the figure) — place the server on host 2 (`plMS(2)`), transfer the mail stream to host 0 (`crMSI(2,0)`), and place the client (`plMC(0)`). Therefore, initially the RG is extended only to level 3. Construction of the progression graph using the propositions and operators of the path shown in bold detects a conflict (the client's QoS requirements are not satisfied). Therefore, the regression graph is expanded to level 4, and later to level 5. Corresponding possible subgraphs are shown in thin solid and dotted lines respectively.

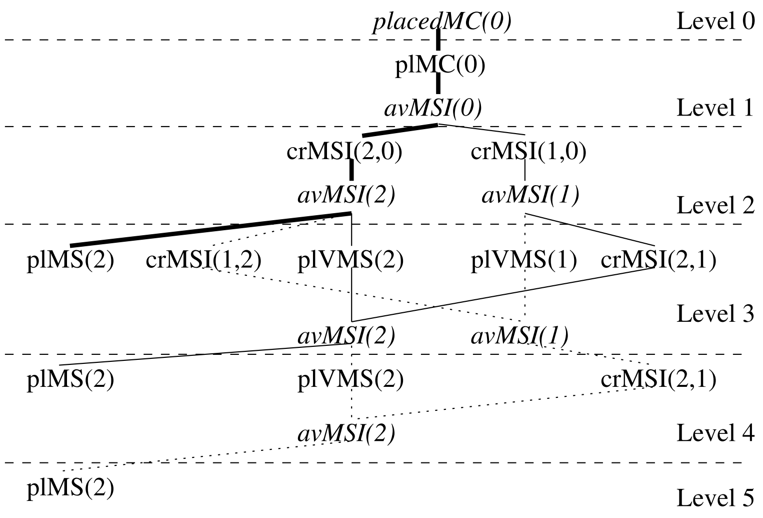


Figure 4.3: Regression graph

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 proposition levels, starting and ending in a proposition level. In addition, this graph contains information about mutual exclusion (mutex) relations [62], e.g., that the placement of a component on a host might exclude placement of another component on the same host (because of CPU capacity restrictions). Because of the propagation of mutex relations, the PG is less optimistic than the RG. Figure 4.4 shows the PG corresponding to the RG in Figure 4.3, which is constructed as described below. Straight lines show relations between propositions and operators, the dotted arc corresponds to a mutex relation.

- Proposition level 0 contains propositions true in the initial state.
- For each proposition of level $i - 1$, a no-op (frame) operator is added to level

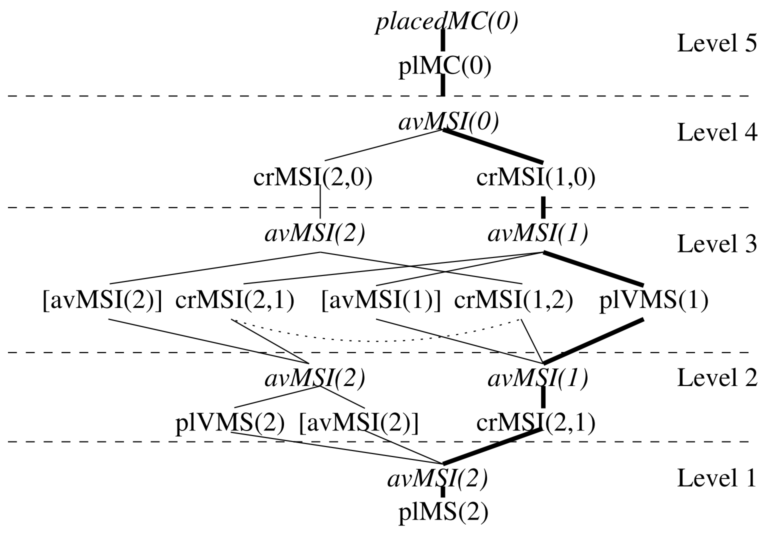


Figure 4.4: Progression graph.

i that has that proposition as its precondition and effect, and consumes no resources (marked with square brackets in the figure). The no-op operators are necessary to model situations when the given proposition is not affected by any operator of the current step.

- For each possible operator 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} proposition 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 propositions 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 the PG, an *optimistic resource map* (ORM) is computed as described in Section 4.2.2. An optimistic resource map describes possible levels of resources achievable at a given stage of plan execution in the form of *intervals*. ORM 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 level is added to the RG, and the PG is reconstructed.

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 [8], using a memoization technique to prevent reexploration of bad sets of propositions in subsequent iterations as described below.

The plan extraction phase performs backward search in the progression graph. For each level of the PG, the planner constructs a set of propositions of that level to participate in the plan. For a subset of propositions P_i at level i, a set of operators O_i of the same level is selected non-deterministically, such that the operators are not mutually exclusive and the union of effects of O_i includes P_i. P_{i-1} is the union of preconditions of O_i. The process is repeated until the initial state (level 0 of the PG) is reached.

Originally, planning graphs were designed to support only Boolean variables in the world state [8]. In the Boolean-only setting, when a set of propositions P_i is considered during the plan extraction, two outcomes of the search from P_i towards the earlier levels of the PG are possible. If the set is achievable, then a plan including P_i will be immediately returned by the algorithm. If the set is not achievable, i.e. P_i is pairwise consistent but contains a mutex of size greater than 2, then the search backtracks past level i. In the latter case, the same set P_i may be revisited along some other search path. The idea of the memoization technique is to save (create a *memo* for) the set P_i as non-achievable so that to avoid repetitive exploration of the portion of the PG below it.

The plan extracted from the PG shown in Figure 4.4 is marked in bold lines.

Symbolic execution

Optimistic resource maps constructed during the second phase of the algorithm constitute a *resource envelope* over all possible plans similar to the *logical envelope* represented by the PG. The plan extraction phase described in the previous section performs the search in the PG by propagating backwards sets of propositions. The RIPP planner [62] adopts a similar mechanism, propagating backwards through an ORM-like structure (which it calls *Resource Time Maps*) numeric intervals for each of the resource variables. To make such propagation possible, RIPP requires all numeric functions to be linear with at most one additional variable.

The functions used to specify the CPP may be non-linear and non-reversible. 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.

4.2.2 Reasoning about resources

The layered structure of the Sekitei 1 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. This section describes how our algorithm reasons about resources.

We assume that all resource functions are monotonic (see Section 3.2.1 for the formal definition). This assumption is true for the applications we are addressing. 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 host with less resources, it still can be deployed on that host if more resources become available.

Let us now introduce several definitions.

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:

$$S = \{(v_1, c_i), ..., (v_n, c_n)\}, \text{where } \forall i \; c_i \in \mathbb{R} \tag{4.1}$$

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

$$exec(op, S) = S' \tag{4.2}$$

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 consider execution of the operator only in the (finitely many) states, in which every variable is equal to one of the boundaries of the corresponding interval.[1]

[1]In practice, only two evaluations of each function are usually required to produce the image intervals.

Let $single(map)$ be a set of all such states for the map map:

$$\begin{aligned} map &= \{(v_1, cm_1, cM_1), ..., (v_n, cm_n, cM_n)\} \\ single(map) &= \{\{(v_1, c_1), ..., (v_n, c_n)\} | \forall i \; c_i \in \{cm_i, cM_i\}\} \end{aligned} \tag{4.3}$$

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

$$\begin{aligned} lmap(l) = \{(v_i, cm_i, cM_i) \;\; | \;\; & cm_i = min\ c, cM_i = max\ c, \\ & (v_i, c) \in exec(op, S), op \in ops(l), \\ & S \in single(lmap(l-1))\} \end{aligned} \tag{4.4}$$

Sekitei uses optimistic resource maps to check numeric preconditions of operators. This allows to identify some resource conflicts during the progression phase of the algorithm (as opposed to the symbolic execution phase), which in turn significantly improves performance of the planner.

4.2.3 Example

To illustrate how Sekitei 1 works, consider the following simple example of the mail application (Figure 4.5). The network consists of three hosts connected in a chain. There is an instance of the `MailServer` running on host 2 able to serve up to 10 requests per second, i.e., `MailServerInterface` (MSI) is available on that host with

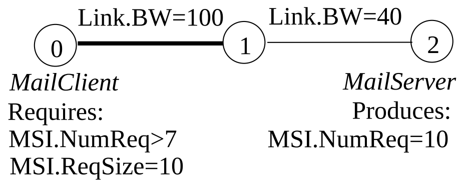


Figure 4.5: A simple example of a mail application.

`MailServerInterface.NumReq`=10. The link between hosts 1 and 2 has low bandwidth as shown in the figure. We want to place a `MailClient` on host 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 hosts, 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 host 0, and placing `ViewMailServer` on host 0 or 1.

Figure 4.6 shows the regression graph for this problem, extended to the level that contains a solution. Similar to the example described in Section 4.2.1, the first time the RG contained a possible subgraph is one level earlier. However, the corresponding progression graph has a resource conflict. We illustrate the construction of the progression graph and the resource envelope using the possible subgraph of the RG shown in Figure 4.6. This subgraph describes execution sequences involving injecting an instance of `ViewMailServer` into the path between the client and the server (one sequence for each of the three possible locations of the `ViewMailServer`).

Figure 4.7 shows the PG that contains a solution 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

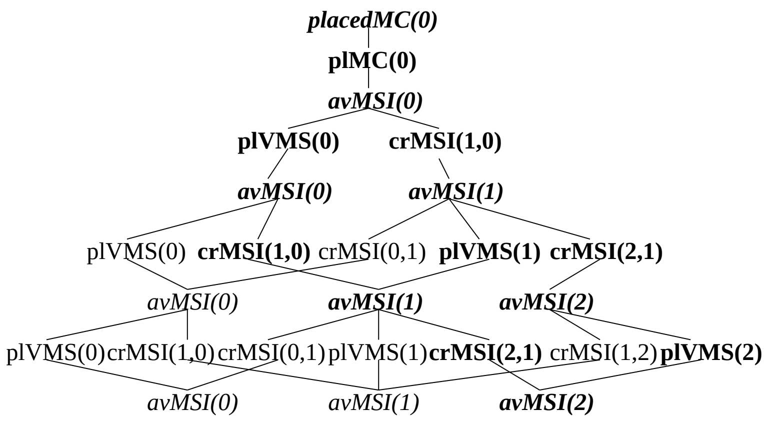
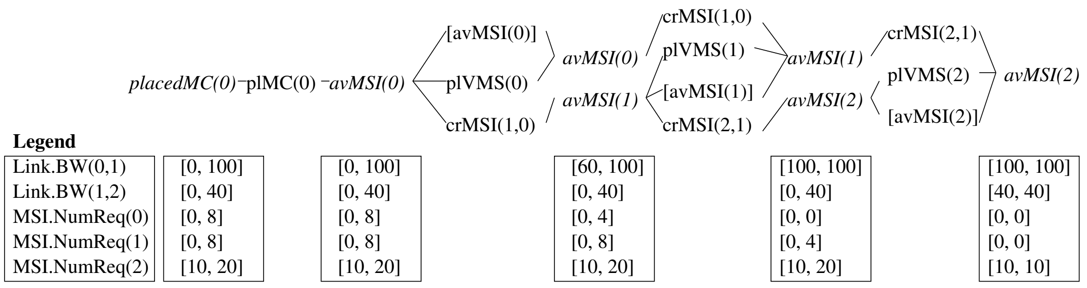


Figure 4.6: The regression graph for the problem shown in Figure 4.5. Possible subgraph is shown in bold font.

a union of maps resulting from execution of each of the three operators of the first operator layer (`crMSI(2,1)`, `plVMS(2)`, and `[avMSI(2)]`) in the initial resource map.

For example, the number of requests supported by `MailServerInterface` on host 1 (`MSI.NumReq(1)`) can be between 0 and 4. The value 0 is obtained if `plVMS(2)` or `[avMSI(2)]` are executed. The value 4 results from the execution of `crMSI(2,1)`. As can be seen from the graph, even though the logical precondition of placement of the `Client` on host 0 (*avMSI(0)*) can be achieved in two link crossing operations, at least three plan steps are required to satisfy its resource precondition `MSI.NumReq(0)`$>$`7`.

The envelope graph formed by the resource maps achieves pruning based on the numeric part of the problem specification by identifying situations when the resource preconditions of operators cannot be satisfied, as illustrated by the above example. However, the envelope graph is optimistic and can contain values, which are not actually achievable (*false positives*). For example, the graph shown in Figure 4.7 reports that the bandwidth of the link between hosts 0 and 1 can be in the interval $[0, 100]$. The



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

Figure 4.7: The progression graph with per-layer resource maps for the problem shown in Figure 4.5. `[avMSI(2)]` is a no-op operator for proposition `avMSI(2)`.

only way to achieve value 0 is to execute both `crMSI(1,0)` operators, which is not allowed on any path encoded in the PG. The value 0 appears in the graph only because the resource maps produced by execution of operators of each level are unioned, and therefore the dependency information encoded in the PG is lost.

One way to decrease the number of such false positives and thus increase the pruning power of the envelope graph is to propagate the resource maps at finer granularity, e.g., per each path as opposed to for the whole graph. However, this would make the envelope graph prohibitively expensive to compute. We will discuss the problem of choosing an appropriate granularity of envelope graphs in Chapter 5.

4.3 Evaluation

In this section we present experimental results illustrating performance of the Sekitei 1 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. 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 a Java implementation of the Sekitei algorithm.

To model different wide-area network topologies, we used the GT-ITM tool [16, 101] to generate eight different networks N_k (for different $k \in \{22, 33, \ldots, 99\}$ hosts). Each topology simulates a WAN formed by high speed and secure stubs (LANs) connected by slow and insecure links. The initial topology configuration files (.alt) were augmented with link and host properties using the Network EDitor tool [59]. The numeric values were chosen so as to make direct connection between stubs infeasible (see Appendix A.2 for an example).

The performance of the planner was evaluated using applications described in Sections 2.3.1 and 2.3.2. The goal in both applications is to deploy the client components on specific hosts. The "best" deployment is defined as the one with the fewest number of components.

4.3.1 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 host n in the network N_k, the goal is to deploy a `MailClient` component on the host n given that the `MailServer` is running on some host. The algorithm indeed finds a solution when it exists.

The data points in Figure 4.8 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 hosts, which takes a very short time.[2] Placement of a client in a different stub requires inserting additional components into the path, and therefore takes longer.

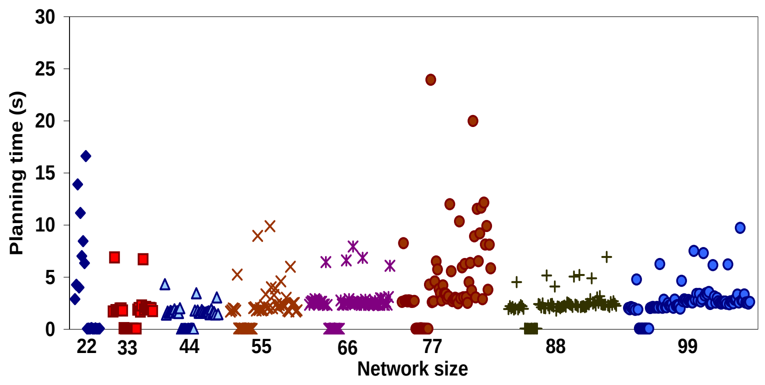


Figure 4.8: Planning under various conditions.

4.3.2 Scalability with respect to 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 4.9) as our reference and starting with a small network with only two stubs, we added one stub at a time until the original 9-stub configuration was achieved. For each of the obtained networks we ran the planner with the goal of placing `MailClient` on a fixed host. Figure 4.10 shows the planning time as a function of the network size.

As shown in the figure, 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 hosts

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

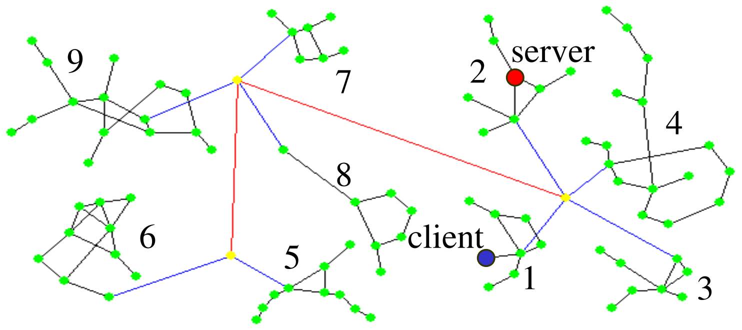


Figure 4.9: 9-stub network N_{99}

reachable in the number of steps bounded by the length of the final plan. Even this set is further pruned at the progression stage. These results show that our algorithm is capable of identifying the part of the network relevant to the solution, without additional preprocessing.

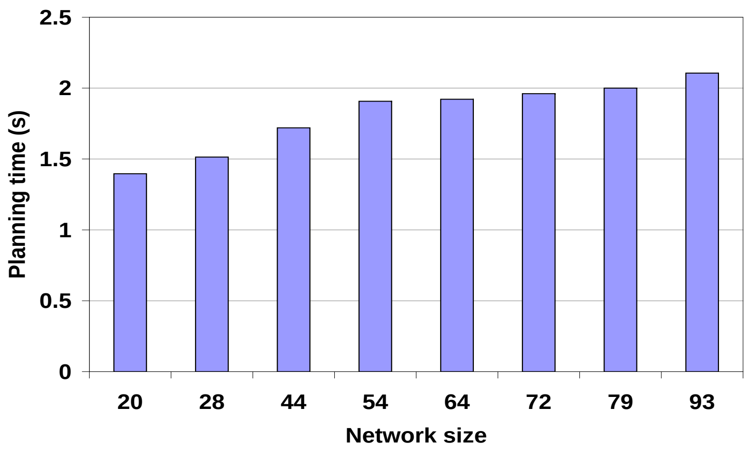


Figure 4.10: Scalability of Sekitei 1 w.r.t. network size for the mail application.

4.3.3 Scalability with respect to 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 configuration of the application; (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 are either already present or can be provided by other useful components.

Figure 4.11 shows the effect of irrelevant components in the problem specification on the planning time. 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 (produce) interfaces meaningful to the application, but require (consume) 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.[3] 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 5.4.

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

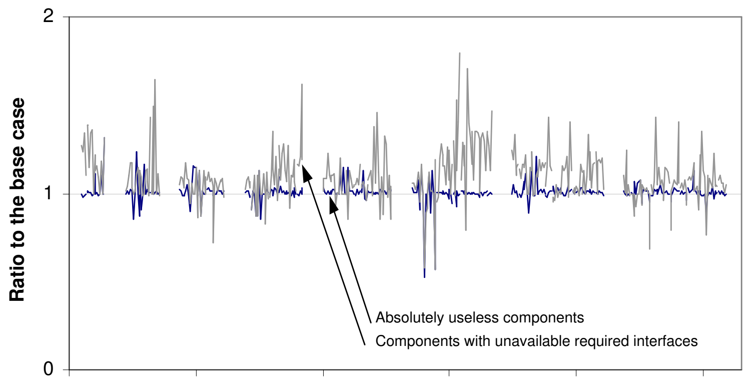


Figure 4.11: Scalability of Sekitei 1 w.r.t. increasing number of irrelevant components.

4.3.4 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 host, we analyzed the planning costs for the goal of putting the `Client` on each of the network hosts in turn. The X-axis in Figure 4.12 represents the order in which the hosts 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 4.12), because most of the path can be reused. The problem in this case is effectively reduced to finding the closest host where the required interfaces are available.

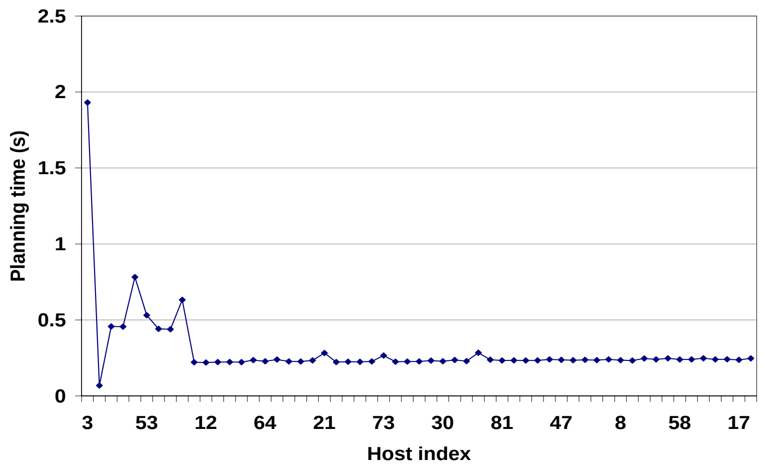


Figure 4.12: Reuse of existing deployments.

4.4 Summary

In this chapter we described the techniques used in the Sekitei planner to cope with the large specifications of the initial state of the world, when not all of the information is necessary to construct a good plan.

Sekitei uses combined regression-progression search (**T2**) to identify relevant portions of the problem specification in the presence of non-reversible functions. The regression part uses only the (easily reversible) logical part of operator specifications, and ensures that the search focuses only on the relevant operators and variables. The progression part is used to construct envelopes that may use non-reversible functions. Envelopes can detect some resource conflicts and thus achieve additional pruning of the search space.

During construction of the graphs using the regression-progression technique, the planner queries the on-demand compiler module for necessary information. Only some parts of the problem specification are obtained from the external services and compiled into planning operators. This way, the on-demand compilation technique (**T1**) allows the planner to work with large scale open worlds.

Several planners [29, 42] can explicitly plan for obtaining necessary information about the world state. However, we are not aware of a planning algorithm capable of acquiring *operators* during search. Bacchus and Teh [4] investigate the dynamic relevance of operators. However, their planner uses a greedy approach based only on the logical part of operator specifications.

As we showed in Section 4.3, Sekitei 1 scales well with the size of the network and the number of component types not used in the application configuration. The on-demand compilation and regression-progression search techniques (**T1** and **T2**) allow the planner to simply ignore such irrelevant information based on logical part of operator specifications. However, these techniques alone are insufficient for dealing with complex numeric interactions between operators. We address this issue in the next chapter.

Chapter 5

Dealing with complex resource functions

In Chapter 4 we described the Sekitei 1 algorithm and the techniques it uses to prune irrelevant information in large-scale open worlds. However, as we discuss in Section 5.1, such logic-driven pruning is insufficient to achieve good performance of the algorithm on the CPP. In Section 5.2 we introduce techniques that permit Sekitei to perform efficient pruning of the search space based on the numeric part of the search specification. We then present Sekitei 2 — the extension of the Sekitei 1 algorithm with these technique — and evaluate its performance.

5.1 Ramifications of numeric interactions between operators

One of the main features of the CPP that distinguishes it from traditional AI planning problems with metric resources is the numeric character of dependencies between the operators. Considering only logical constraints, most of the component placement problems have a simple solution. For example, in stream delivering applications such as webcast (Section 2.3.2) the client can often be directly connected to the server along the shortest path through the network. Configurations involving additional components, such as compression and encryption, are necessary only due to resource and QoS constraints.

The first version of the Sekitei algorithm described in the previous chapter builds an optimistic resource envelope for the whole planning graph, but does not propagate intervals backwards during the plan extraction phase because of non-reversibility of resource functions. The resource conditions are checked only when the plan extraction phase completes plan construction, i.e. during the symbolic execution phase.

The envelope allows the planner to identify some resource conflicts, which helps to significantly improve performance of the planner. However, for applications with more complex structure (e.g., DAGs as opposed to chains), a single envelope for the whole search space is not enough to identify resource conflicts between different branches of the application. For example, non-trivial configurations of the webcast application require transferring two data streams over the network. Data transfers and components processing these streams may compete for the same network resources. In such cases, if a single envelope is used, 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, which leads to a worst case exponential time spent in the third phase of the algorithm before the conflict can be detected.

5.2 Techniques

This section introduces techniques used in Sekitei 2 in addition to the techniques used in Sekitei 1. We start by providing intuition for how additional envelopes can improve the pruning power of the planning algorithm. Then, we describe construction of *bounded envelopes* (**T3**). Finally, we combine these ideas to construct *envelope hierarchies* (**T4**), which help to increase the pruning power of the algorithm while incurring a small computational overhead.

5.2.1 Adding a new envelope

In the previous chapter we described Sekitei 1 as consisting of four phases. Sekitei 1's algorithm can also be viewed as a hierarchy of two levels of the regression-progression search. The first level of the hierarchy, which consists of the regression and progression phases, builds an envelope graph by aggregating information for all executable sequences of operators. The second level, consisting of the plan extraction and symbolic execution phases, also uses the regression-progression search technique, but considers each executable sequence separately. This second level of the hierarchy essentially performs exhaustive search in the progression graph.

Sekitei 1 uses a memoization technique [8] to avoid reexploration of sets of propositions not achievable together. In the presence of numeric state variables, the same set of propositions can be revisited even if it was successfully achieved, because the failure of numeric constraints can be detected later (closer to the goal). For this reason, Sekitei 1 does not scale well with the number of additional components in the plan (see Section 5.4).

One solution to the late resource conflict detection problem is to save intermediate results. Similar to the (negative) memoization used in GraphPlan-based planning algorithms [8], we use **positive memoization** to save good sets of propositions *along with corresponding resource maps*.

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:

$$\begin{aligned} smap(q,l) = \{(v_i, cm_i, cM_i) \quad | \quad & cm_i = \min\ c, cM_i = \max\ c, \\ & (v_i, c) \in exec(op, S), op \in O, O \in ops(q,l), \\ & S \in single(smap(precs(O,l), l-1))\} \end{aligned} \tag{5.1}$$

   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 (symbolically executed) in the optimistic map of

o's preconditions $smap(precs(o,l),l-1)$. This helps to identify resource conflicts close to the goal early, and greatly improves planning time.

For example, consider the second layer of the PG shown in Figure 4.7. This layer contains three operators, `crMSI(1,0)`, `plVMS(1)`, and `[avMSI(1)]`, which have the same logical precondition *avMSI(1)*. *avMSI(1)* is logically achievable. However, two of the operators lead to a resource conflict. Therefore, without the positive memoization, the planner will try to achieve proposition *avMSI(1)* up to three times. Using the positive memoization technique, an optimistic resource map for the singleton set containing this proposition is computed only once, and then reused when the other two operators are considered by the plan extraction procedure.

The memoized good sets and their resource maps form the *second envelope graph*. Note that this new envelope graph has *finer granularity* than the first graph. The envelope at the first level of hierarchy is constructed for the whole progression graph, while the new envelope is constructed per set of propositions.

Adding positive memoization to Sekitei 1 results in orders of magnitude speedup on some instances of the webcast problem and a small increase of running time on simple problems (see Section 5.4 for results). Note that the use of positive memoization does not put any additional restrictions on the form of resource functions. 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.

This tradeoff between the pruning power of an envelope graph and the computational cost of constructing it is the main obstacle in using envelope-based pruning. In

Section 5.2.3 we will show how to resolve this trade-off.

5.2.2 Bounded envelope graphs

The envelope graphs of Sekitei are constructed from the goal using the regression-progression technique (**T2**). The envelopes grow as more information becomes necessary during the search.

In Sekitei, the purpose of the envelope graphs is to prune the search space and to produce *lower bound estimates* of costs of achieving intermediate goals, such as propositions or sets of propositions.

Note that the bounds produced by an envelope are used only to choose between options during the search. Therefore, the exact values of the lower bounds are irrelevant, and the expansion of the graph needs to be performed only until the values are accurate enough to make the correct choice (**T3**). In the next section we describe how Sekitei 2 computes bounds for envelope expansion.

5.2.3 Hierarchy of envelopes

In Section 5.2.1 we described a two-level hierarchy of envelopes based on the positive memoization technique. However, positive memoization has high memory (and computational) overhead. Sekitei 2 addresses this issue using a **hierarchy of bounded envelope graphs** (**T4**) to combine the high pruning power of fine-granularity envelopes with the low computational overhead of coarse-granularity envelopes.

The communication between the levels of the hierarchy is bidirectional. The coarse envelopes perform some pruning of the search space and provide finer envelopes with the sets of relevant propositions and operators (compare to the pruning

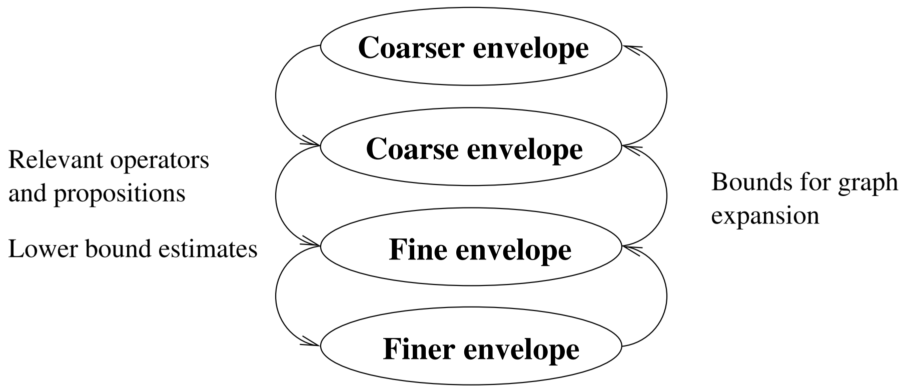


Figure 5.1: Envelope hierarchy

achieved by the RG and PG for the plan extraction and symbolic execution phases of Sekitei 1). The lower bounds on costs of achieving intermediate goals (e.g., propositions or sets of propositions) are used to choose between options during expansion of finer envelopes. In the opposite direction, the fine envelopes provide the coarser envelopes with the bounds for graph expansion (Figure 5.1). Sekitei 2's algorithm described in Section 5.3 uses a three-level hierarchy.

5.2.4 Other modifications of the algorithm

When replacing the per-layer graphs of Sekitei 1 with an envelope hierarchy of Sekitei 2, several points are worth mentioning.

First, 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 the *absence* of an interface on a host), resource interference is the only source of mutex relations.

Second, recall that the purpose of the PG in the Sekitei 1 algorithm is to compute mutex relations and to provide basis for computation of the memoization table. Without mutex relations, there is no need to explicitly store the PG. All information contained in the PG can be merged into the RG. Such a combined regression-progression graph constitutes a single envelope graph.

Finally, the regression part of the envelope construction algorithm uses only positive logical values. Negations in preconditions and effects are not supported by Sekitei 2. However, this restriction does not limit the expressive power of the algorithm, because logical expressions involving negations can be represented as numeric functions (see Section 3.3).

5.3 Algorithm

Sekitei 2 uses a hierarchy of three envelope graphs. The first graph computes the cheapest way to achieve a single proposition based only on the logical part of the specification of operators. The second graph takes into account optimistic resource maps and sets of propositions. Finally, the last and the most computationally expensive graph works with plan tails similar to the plan extraction phase of Sekitei 1.

5.3.1 First level graph

The first graph of the hierarchy used in Sekitei 2 (RPG1, for *regression-progression graph*), is a simplified version of the regression graph of Sekitei 1. RPG1 uses more aggregation than the original RG, and therefore is even cheaper to compute.

Each of the graph nodes maintains a lower bound of the cost of achieving that node from the initial state. The cost of each proposition is taken as the minimum cost of all operators that can logically achieve it. The cost of an operator is taken as the cost of its most expensive precondition plus the lower bound on the operator cost. Sekitei 2 assumes integer costs.

RPG1, as all graphs in Sekitei, is expanded using the regression-progression technique (**T2**). During the graph expansion, the cost of a leaf, i.e., a not yet expanded node, is 0.[1] After each node expansion, the updated (increased) cost estimates are propagated forward.

To start with, the graph is expanded until the initial state is reached. Further expansion of the graph may be required when the second level envelope graph extends the bound.

5.3.2 Second level graph

The second-level graph of Sekitei 2 (RPG2) 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 maintains a *cost* of reaching the current node from the initial state. In the case of unit costs of operators, this node cost corresponds to the layer number of the PG of Sekitei 1.

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.

[1]See Section 6.4.4 for a better lower bound estimate.

The nodes of the RPG2 are expanded as follows. 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 all 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 obtained from RPG1.
- 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.

5.3.3 Third level graph

Since resource maps are unioned at various points during construction of the RPG2, 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 of Sekitei 1 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, and 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 among the 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.

Initially, each of the envelope graphs is expanded until it reaches the initial state. The coarser is an envelope, the more optimistic it is. Therefore, coarse envelopes reach the initial state relatively fast, which limits their expansion. The finer, less

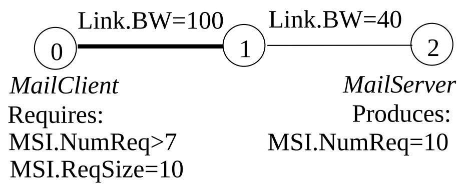


Figure 5.2: A simple example of a mail application.

optimistic envelopes can detect conflicts that coarser envelopes missed. For example, the third level envelope can detect failure during the plan tail execution (step 7) even though it uses the conflict-free second level envelope.

Every time a failure is detected during construction of an envelope graph, a new, more expansive node is fetched from the graph's queue. At this moment, the new expansion bounds are imposed on the next coarser envelope in the hierarchy to ensure correct ordering of the nodes resulting from expansion of the newly fetched node.

5.3.4 Example

Figure 5.2 repeats the example problem from Section 4.2.3. As earlier, the goal is to deliver a mail stream with high supported request rate over a link with low bandwidth. Analysis of numeric restrictions shows that the direct connection does not satisfy client requirement, but a configuration involving a caching component on host 1 does.

Figure 5.3 shows the first-level graph for this problem expanded so as to include operators for placing the caching component (`plVMS(0)`, `plVMS(1)`, `plVMS(2)`). The arrows go from preconditions to operators and from operators to their effects.

Figure 5.4 shows the second level graph for the same problem. In this example all operators have exactly one precondition. Therefore, all AND nodes have exactly one

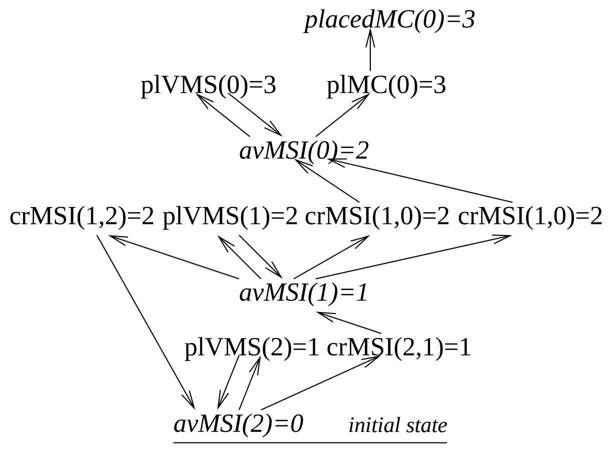


Figure 5.3: RPG1 of Sekitei 2.

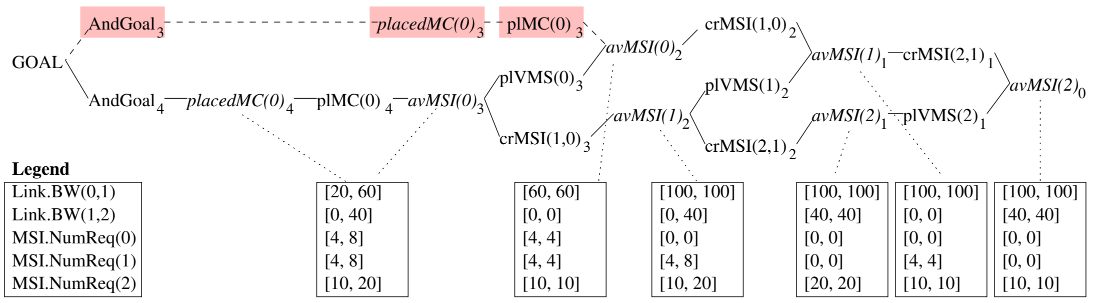


Figure 5.4: RPG2 of Sekitei 2. The shaded boxes show the nodes of the graph that are declared dead during the progression phase of the graph construction.

aggregate node, which are not shown in the figure. Proposition nodes are shown in italics, operator nodes in normal font. Subscripts correspond to the cost of a node. The 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 marked dead.

RPG2 takes unions of resource maps for preconditions of operators, and thus is optimistic. The third-level graph (Figure 5.5) considers sequential plan tails and does not perform such aggregation. This way RPG3 guarantees correctness of the solution.

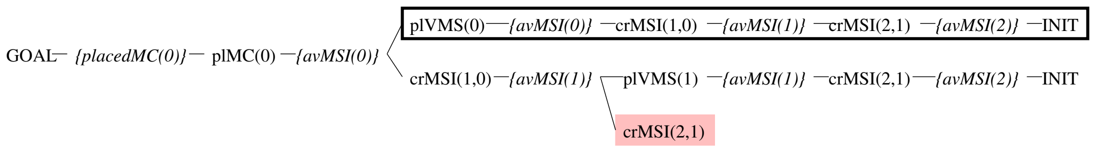


Figure 5.5: RPG3 of Sekitei 2 does not perform aggregation, which allows it to detect conflicts missed by RPG2. The shaded box marks a node where one such conflict is detected. The bold box marks an alternative solution. In this example both solutions have the same cost, and either one can be found by the planner.

5.4 Evaluation

In this section we evaluate the effect of the hierarchy of bounded envelopes (techniques **T3** and **T4**) on the performance of the planner. We use Java implementations of Sekitei 1, Sekitei 1 with positive memoization (a two-level envelope hierarchy), and Sekitei 2 (a three-level envelope hierarchy). All measurements are taken on a 700MHz Pentium III running Windows 2000 with up to 200MB of memory available for the Java VM.

5.4.1 Experimental setup

In the experiments described below we used the webcast application from Section 2.3.2. Appendix C provides the specifications of components, including the functions describing preconditions and effects.

The client requires the incoming data stream to have bandwidth greater than 80 units and to have low resolution. Depending on the resolution of the stream produced by the server and the available bandwidth between the server and the client, different sets of additional components may be required to satisfy client's preconditions.

Logically, all components may be used. To choose the best configuration, the planner needs to efficiently reason about the numeric conditions.

To evaluate the effect of relevant components with numeric dependencies on the performance of the planner, we used the N_{99} network and four different scenarios described below.

Cfg 1. In the first case the transit links have high bandwidth (100 units), and the server produces a low resolution stream. In this situation, the `Client` can be directly connected to the `Server`.

Cfg 2. In the second case, the server produces the stream with high resolution. 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 low (70 units), but the server produces the stream of the required resolution. To satisfy client's requirements the text portion of the stream needs to be compressed. The planner decides to add `Splitter`, `Merger`, `Zip`, and `Unzip` components into the data path.

Cfg 4. Finally, in the fourth configuration, the links have low bandwidth (70 units) and the server produces a high-resolution stream, so that five additional components are required to satisfy client's requirements: `Splitter`, `Merger`, `Zip`, `Unzip`, and `Filter`.

LAN hosts have 10 units of available CPU. This limits the number of components that can be placed together on the same host, increasing the number of resource conflicts encountered during the search and thus making the problem even harder.

5.4.2 Planning time

Figure 5.6 shows the performance of Sekitei 1 (without optimizations) for an 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. Note that the Y-axis is shown using log scale.

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 the 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 process can take a long time.

Figure 5.7 shows the planning time for the same experiment presented above for the planner with the positive memoization technique discussed in Section 5.2.1. The modified version of the planner takes about the same time on simple problems (Cfg 1), and scales much better on harder instances. Figure 5.8 shows the additional improvements of Sekitei 2. These results demonstrate that a hierarchy of envelopes (**T4**) provides orders of magnitude performance improvement on problems involving tight resource constraints.

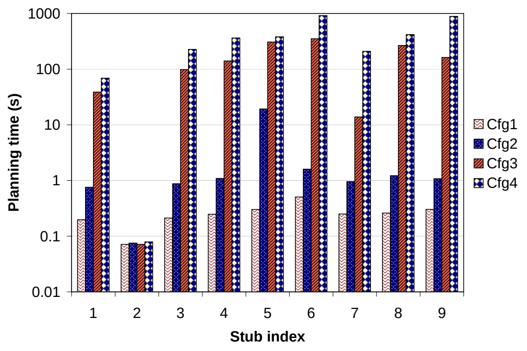


Figure 5.6: Scalability of the original Sekitei 1 algorithm w.r.t. increasing number of relevant components. The highest peaks correspond to about 15 minutes.

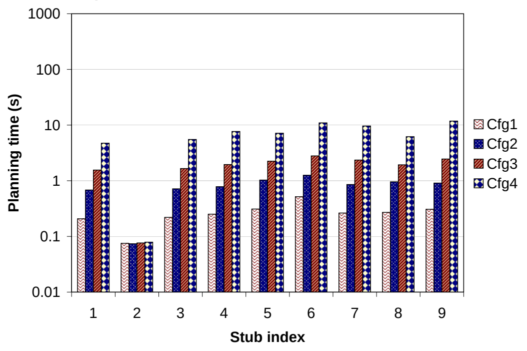


Figure 5.7: Scalability of Sekitei 1 with positive memoization w.r.t. increasing number of relevant components. The highest peaks correspond to about 10 seconds.

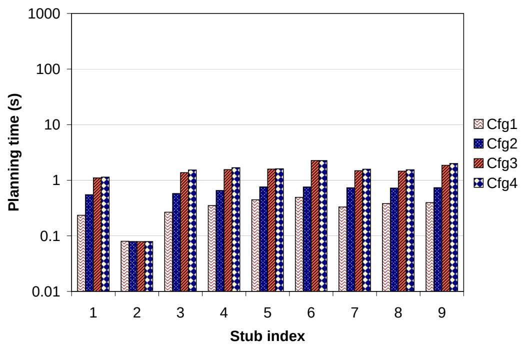


Figure 5.8: Scalability of Sekitei 2 w.r.t. increasing number of relevant components. The highest peaks correspond to about 2.3 seconds.

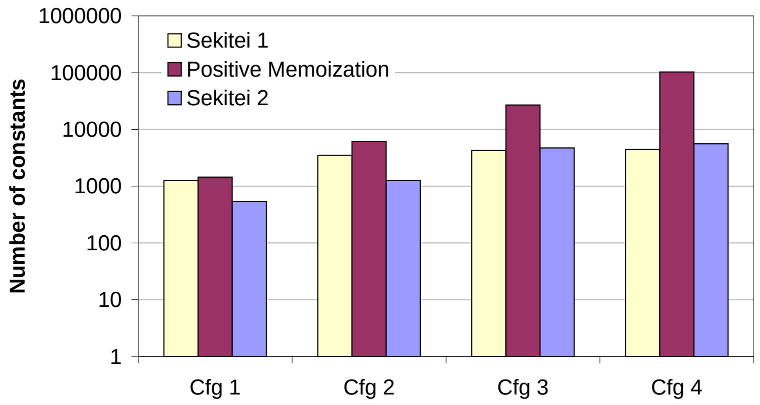


Figure 5.9: The average number of generated constants on four configuration of the webcast application.

5.4.3 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 the algorithm, 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 5.9 shows (on log scale) the average number of constants generated by the three versions of Sekitei on four configurations of the webcast application discussed above. Sekitei 2 scales much better with respect to memory consumption as compared to the positive memoization version of Sekitei 1. In fact, memory consumption of Sekitei 2 is comparable to that of the original version of the algorithm.

We also tested the scalability of Sekitei 2 with respect to irrelevant operators. The behavior of the planner is similar to that reported in Section 4.3, and we do not present the detailed results here.

5.5 Summary

In this chapter we showed how a hierarchy of bounded envelopes (**T3** and **T4**) helps Sekitei to cope with numeric dependencies between operators.

The idea of an envelope as a cheap data structure to identify conflicts has been used before in the context of planning and scheduling [73, 67, 62]. GRT-R [83] uses a finer granularity map to achieve better pruning by saving more intermediate results in memory. TP4 [46] propagates achievable resource values for constant effects in a regression-based planner. The technique of using a hierarchy of bounded envelopes allows our planner to support more expressive functions than those of TP4 while avoiding the memory explosion problem of GRT-R.

The techniques described in this and the previous chapters allow the Sekitei planner to efficiently prune large search spaces and cope with complex interactions between operators. However, Sekitei 2 still uses greedy propagation of numeric values and assumes unit action cost. In the next chapter we address these issues.

Chapter 6

Optimizing resource consumption

The two versions of Sekitei described so far minimize the number of steps in the plan and employ a greedy strategy for resource assignment.

In this chapter we provide examples that illustrate shortcomings of this approach, and introduce two techniques (**T5** and **T6**) to address them. We also present Sekitei 3 — the version of the algorithm that uses techniques **T1–T6** to address all three challenges listed in Section 3.4.

6.1 Limitations of the greedy approach

Because of the non-reversibility of resource functions, the numeric part of the envelope graphs can be propagated only in the forward direction. This forces the planner to adopt a greedy approach with respect to resource allocation: as much data as possible is processed and pushed through the network, incurring high resource requirements.

Because of this greediness, the versions of the planner described so far guarantee

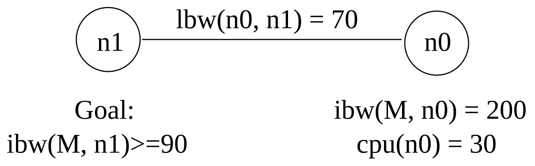


Figure 6.1: Resource optimization is required to find a plan.

feasibility of a solution, but cannot minimize resource consumption. Although it is possible to add a post-processing step to achieve this latter goal, this is not enough as the following examples demonstrate:

Scenario 1. Consider the example in Figure 6.1, where we want to deliver at least 90 units of bandwidth of the M stream (the requirement of the client component) over the link with bandwidth 70. The source node has 200 units of M available, but only 30 units of CPU. Suppose, transformation of 200 units of M by the splitter requires 40 units of CPU. Sending the M stream directly to the client does not satisfy the client's bandwidth requirements, and the amount of CPU available on node `n0` is less than that required for processing all available bandwidth of the M stream, as would be required by the greedy approach. Consequently, a greedy planner will not find a solution to this CPP even though one exists. If we allow the splitter to transform only 90 units of bandwidth of the available M stream (the amount required by the client), then the total CPU requirements of the Splitter and Zip components may be less than 30 units, and the solution shown on Figure 6.2 can be found, which involves splitting the M stream and compressing its text component on node `n0` and performing the reverse transformations on node `n1`.[1]

[1]We assume that the target node has sufficient CPU resources for the Unzip and Merger components.

place Splitter on node n0,
place Zip on node n0,
cross with Z stream from n0 to n1,
cross with I stream from n0 to n1,
place Unzip on node n1,
place Merger on node n1.

Figure 6.2: Plan for the problem presented in Figure 6.1.

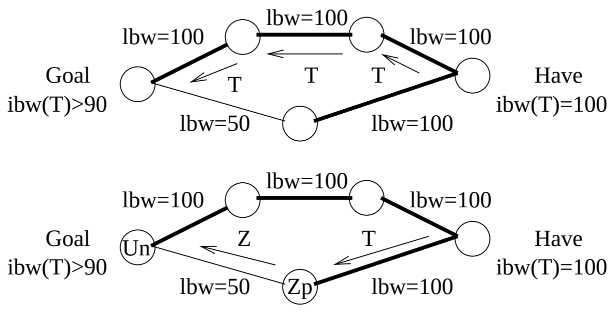


Figure 6.3: Effect of cost functions on the choice of plan.

Scenario 2. Another desirable feature not provided by the greedy model is the ability to specify preferences over the space of generated plans. For example, consider the problem shown in Figure 6.3. Here the goal is to deliver a text stream from the server to the client, which requires the incoming stream to have bandwidth greater than 90 units. In this scenario, there are two possible application configurations: one involving a crossing of three links, and another that would require two link crossings and the use of Zip and Unzip components. Which plan would perform better in a given situation depends on the relative cost of link bandwidth and host resources. Such tradeoffs can be performed by introducing a cost function that depends on resource consumption, which an ideal planner can then optimize. Note that, in general, the cheapest plan is not necessarily the one with the smallest number of steps.

6.2 Techniques

The scenarios described in the previous section illustrate two shortcomings of Sekitei 2: inability to optimize resource consumption or other user-supplied cost and performance metrics, and inability to find plans in resource-constrained situations.

The reason for the former is that the version of the algorithm presented so far assumes unit operator cost. Making the operator cost a function of consumed resources allows the planner to reason about relative quality of different solutions. However, just introducing such cost functions does not address the problems above.

The reason for the inability of Sekitei 2 to optimize resource consumption in an application configuration is the non-reversible nature of the resource functions, which forces the planner to adopt a greedy approach. The simplest way to address this issue is to assume reversibility of functions, in which case the required values of resources can be propagated during the regression phase [62]. However, this is at odds with what one finds in practice. Functions describing component behavior are often represented by tables obtained by application profiling. It is not always possible to derive an analytical representation of such functions, and even less reasonable to assume reversibility of such functions. Consequently, we adopt a different approach, which approximates optimality while still being practically usable.

In this section we introduce two techniques that embody the above observations. First, we extend the action specification with a cost function depending on the values of numeric state variables (**T5**), which in the case of the CPP describe consumed resources and parameters of incoming data streams. Second, we add discrete parameters corresponding to intervals of values of real-valued resources (**T6**). Sekitei 3 incorpo-

```
(:action plZip
 :parameters   (?n - host)
 :precondition (>= (cpu ?n) (/ (ibw T ?n) 10))
 :effect       (and (placed Zip ?n)
                    (assign (ibw Z ?n) (/ (ibw T ?n) 2))
                    (decrease (cpu ?n) (/ (ibw T ?n) 10)))
 :cost (+ 1 (/ (ibw T ?n) 10))
)
```

Figure 6.4: plZip operator before leveling. The extension to PDDL is shown in bold font.

rates these extensions into the algorithm based on a hierarchy of bounded envelopes described in the previous chapter.

6.2.1 Cost functions

Most of the classical planning algorithms minimize the length of the plan. In the CPP (and the ACP in general), the shortest plan is not always the best. It is desirable to be able to express relative costs of different resources and find a configuration that minimizes the total resource consumption of the application.

To achieve this goal, we extend specification of operators with cost functions that use numeric state variables. Such functions can be attached to component types (for deployment actions) and interface types (for data transfer actions), or directly to PDDL operators. Figure 6.4 gives an example of a PDDL-style operator with a cost function. This operator describes placing of a Zip component on a host. The component requires that the host has sufficient CPU to process the incoming stream. The bandwidth of the produced compressed stream is half of that of the consumed text stream.

As defined in Section 3.2.2, every action of the ACP has a cost described by a function of the consumed resources and the properties of the data being processed. This action cost function is directly mapped onto the cost of the corresponding planning operator. The total cost of an application configuration is a function of costs of individual actions constituting it. The cost functions can describe the overhead of the initial deployment of the application, the run-time cost of the configuration (e.g., the total amount of resources required to sustain a data stream), or a combination of both.

For the cost functions of operators to be used with Sekitei's resource envelope technique, they need to be monotonic and forward computable as defined in Section 3.2.1. In addition, a function that computes the total cost of a configuration using costs of individual operators needs to be defined. Our current implementation performs summation, but any non-decreasing function can be used instead.

Note that the cost functions describe a metric. This metric is defined *in addition* to the qualitative and quantitative constraints, both local and global, imposed by resource availability and execution preconditions of actions. The planner searches for a feasible configuration that minimizes the total cost.

6.2.2 Resource levels

To permit the planner to reason about numeric resources during the regression phase of the search, we introduce discrete counterparts for continuous variables. Every interface property or network resource, which appears as a real-valued variable in a specification formula, is assumed to have one or more **levels** associated with it. The levels specify disjoint intervals of values of the resource and are defined by the interval bounds. Resources for which no intervals are specified are assumed to have one interval $[0, \infty)$. For example, the specification of the M stream shown in Figure 6.5

```
<interface name=M>
 <cross_effects>
  M.ibw'    := min( M.ibw, Link.lbw )
  Link.lbw' -= min( M.ibw, Link.lbw )
 <levels>
  <cutpoint value=30>
  <cutpoint value=70>
  <cutpoint value=90>
  <cutpoint value=100>
```

Figure 6.5: Specification of an interface with resource levels. The tick mark in the specifications serves to distinguish the value of a resource after the link crossing operation.

interface	**level 1**	**level 2**	**level 3**	**level 4**	**level 5**
M	[0,30]	[30,70]	[70,90]	[90,100]	[100,∞)
I	[0,9]	[9,21]	[21,27]	[27,30]	[30,∞)
T	[0,21]	[21,49]	[49,63]	[63,70]	[70,∞)
Z	[0,10]	[10,25]	[25,31]	[31,35]	[35,∞)

Table 6.1: Resource levels for the four interface types.

defines five intervals for the bandwidth property: $[0, 30]$, $[30, 70]$, $[70, 90]$, $[90, 100]$, and $[100, \infty)$. Table 6.1 shows levels for bandwidth of other streams.

Additionally, a property of a resource can be marked as being *degradable*, *upgradable*, or neither. A degradable resource tag indicates that the availability of a resource at a higher value indicates its availability at a lower value as well. For example, link bandwidth is a degradable resource. Similarly, an upgradable resource is assumed available at a higher value when a lower value is present. Information about degradability (upgradability), which can be obtained automatically by syntactic analysis of the problem specification or provided manually, helps the planner to find plans in resource-constrained situations as described below.

The key insight underlying the resource discretization technique is that more than an exact understanding of the resource consumption effects of a planning operator, what we care about is the ability to identify operators that come close to the right (optimal) decision. The latter is somewhat easier and more reasonable for a domain expert to provide information on. In particular, experts are already used to thinking of different operational regimes for components as also qualitatively different regions of values for network resources.

6.2.3 Leveled operators

The main benefit from identifying resource levels is that we can incorporate that information when defining operators for the AI-style planning problem compiled from the CPP specification. Specifically, levels for all resources mentioned in the operator specification are added as parameters to the operator.

The set of levels is discrete, which means that search techniques that use data structures such as progression and regression graphs are still applicable to the problem.

On the other hand, the resource levels correspond to intervals of real values for resources. Therefore, adding resource levels to operator parameters is an approximation for using real-valued parameters.

Note that the resource levels are used in addition to, rather than instead of, real-valued resources. The effect the resource levels have on the planning problem is that of replacing metric operators with sets of operators with additional resource preconditions restricting values of numeric state variables. Such modification of the operators essentially pushes part of the resource restrictions into the discrete (logical) specification of operators, thus allowing the planner to reason about numeric resources during the regression phase of the search.

For example, given two resource levels for bandwidth of text and zip streams, the `plZip` operator (Figure 6.4) can be replaced with four operators as shown in Figure 6.6. Each of these new operators has additional preconditions based on the resource intervals, which limit applicability of these operators, and different lower bounds on the cost. During the envelope construction, the planner uses the cheapest operators that do not cause conflicts, thus optimizing the cost of the solution and its resource consumption.

6.2.4 Constructing leveled operators

Extension of operators with parameters for resource levels can be done automatically (ideally), or manually. The manual approach may produce a smaller set of operators, and thus improve the performance of the planner. Let us first consider the automatic approach.

The planner tries to instantiate each operator from the program description with all possible values for its parameters. Ground operators for which the resource requirements are not satisfied are immediately pruned from further consideration. The following algorithm checks feasibility of a ground leveled operator and computes its cost.

1. Construct an optimistic resource map for intervals corresponding to the chosen levels of required resource variables.

2. Check if resource preconditions hold in the optimistic map. If not, ignore the ground operator.

```
(:action plZip00
 :parameters   (?n - node)
 :precondition (and (>= (cpu ?n) (/ (ibw T ?n) 10))
                    (< (ibw T ?n) 50)  (< (/ (ibw T ?n) 2) 30))
 :effect       (and (placed Zip ?n)
                     (assign (ibw Z ?n) (/ (ibw T ?n) 2))
                     (decrease (cpu ?n) (/ (ibw T ?n) 10)))
 :cost (+ 1 (/ (ibw T ?n) 10)))

(:action plZip01
 :parameters   (?n - node)
 :precondition (and (>= (cpu ?n) (/ (ibw T ?n) 10))
                    (< (ibw T ?n) 50)  (>= (/ (ibw T ?n) 2) 30))
 :effect       (and (placed Zip ?n)
                     (assign (ibw Z ?n) (/ (ibw T ?n) 2))
                     (decrease (cpu ?n) (/ (ibw T ?n) 10)))
 :cost (+ 1 (/ (ibw T ?n) 10)))

(:action plZip10
 :parameters (?n - node)
 :precondition (and (>= (cpu ?n) (/ (ibw T ?n) 10))
                    (>= (ibw T ?n) 50)  (< (/ (ibw T ?n) 2) 30))
 :effect       (and (placed Zip ?n)
                     (assign (ibw Z ?n) (/ (ibw T ?n) 2))
                     (decrease (cpu ?n) (/ (ibw T ?n) 10)))
 :cost (+ 1 (/ (ibw T ?n) 10)))

(:action plZip11
 :parameters (?n - node)
 :precondition (and (>= (cpu ?n) (/ (ibw T ?n) 10))
                    (>= (ibw T ?n) 50)  (>= (/ (ibw T ?n) 2) 30))
 :effect       (and (placed Zip ?n)
                     (assign (ibw Z ?n) (/ (ibw T ?n) 2))
                     (decrease (cpu ?n) (/ (ibw T ?n) 10)))
 :cost (+ 1 (/ (ibw T ?n) 10)))
```

Figure 6.6: Expansion of plZip operator using one cutpoint for T stream (50) and Z stream (30).

3. Evaluate the cost function in the optimistic map. Take the minimum cost (the lower bound of the interval) as the cost of the operator.

4. Compute resource effects in the optimistic map. If the resulting intervals do not intersect with those required by the chosen levels, ignore the ground operator.

For example, if the amount of CPU on host `n0` in the initial state is 10 units, then the `placeZip` operator (Figure 6.4) for host `n0`, $[0, 21]$ resource interval for the incoming T stream, and $[10, 25]$ interval for the produced Z stream is generated as follows:

- The optimistic map is $\{$`(cpu n0)=[0,10], (ibw T n0)=[0,21]`$\}$.
- The precondition `(`$>=$ `(cpu n0) (/ (ibw T n0) 10))` is satisfied.
- The cost formula `(+ 1 (/ (ibw T n0) 10))` evaluates to `[1,3]`. Therefore, the cost of the ground operator is 1.
- The resulting value of bandwidth of the Z stream is `[0, 10.5]`, which is consistent with the second level interval for Z `[10, 25]`.

The described algorithm does not assume anything about how the intervals are defined. It is still able to prune some operators (e.g. `placeZip` with the $[0, 21]$ interval for T and the $[25, 31]$ interval for Z will be pruned), but lots of other operators will pass the tests. If the levels in operators are introduced manually, it is possible to further reduce the number of operators. For example, a human can see that, given the way the intervals are defined, `placeZip` produces the same level of stream Z as the incoming stream T, allowing the definition shown in Figure 6.7.

```
(:action placeZip
 :parameters   (?n - node)
 :precondition (>= (cpu ?n) (/ (ibw T ?n) 10))
 :levelprecondition (= (IN ibw Z ?n) (OUT ibw T ?n))
 :effect       (and (placed Zip ?n)
                    (assign (ibw Z ?n) (/ (ibw T ?n) 2))
                    (decrease (cpu ?n) (/ (ibw T ?n) 10)))
 :cost (+ 1 (/ (ibw Z ?n) 5))
)
```

Figure 6.7: Leveled operator with additional mark-up.

6.3 Algorithm

The third version of our algorithm, Sekitei 3, uses the techniques introduced in the previous chapters for dealing with large-scale open worlds and non-reversible resource functions. Moreover, it takes these techniques one step further to take advantage of resource levels and support optimization.

Sekitei 3 uses a three-level hierarchy of envelopes with different granularity (**T3** and **T4**). An on-demand compiler module (**T1**) is used to hide semantics of the problem domain from the planner, and regression-progression search (**T2**) is used for expansion at all levels of the envelope hierarchy.

For more unified treatment of different kinds of dependencies, Sekitei 3 exploits the idea of compilation of a logical problem into a numeric one (Section 3.3.1). All discrete variables used in search correspond to levels of numeric variables. Boolean variables are treated as numeric variables with two levels.

The first graph of Sekitei 3, the per-proposition logical regression graph, estimates the minimum cost of achieving a proposition from the initial state. Both resource restrictions and interactions between actions are ignored in this case.[2] Given the minimum proposition cost, the second graph computes the minimum logical cost of achieving a *set* of propositions. This phase takes into account logical interactions between actions, but ignores resource restrictions. Finally, during the last phase of the algorithm, the search for a plan is performed that uses all types of restrictions and estimates the remaining cost using the logical cost of achieving a set of propositions.

6.3.1 Envelope construction

Similar to Sekitei 1 and 2, Sekitei 3 builds envelope graphs using regression-progression search (**T2**). To support non-unit operator cost, Sekitei 3 uses A*-like search instead of per-layer expansion.[3]

The classical A* search [45, 65, 85, 66] finds the shortest path from a particular state (the initial state) to some state that satisfies the goal condition. For each intermediate state, the distance from the initial state to the current state is known exactly, and the distance from the current state to the goal is estimated using an admissible heuristic. The term *admissible* means that the heuristic does not overestimate the true value.

The envelope construction algorithm of Sekitei 3 differs from classical A* in several aspects. First, an envelope is constructed using bidirectional regression-progression (**T2**) search as opposed to a single pass of A*. This is necessary, because during the regression phase both parts of the cost of a graph node (to the goal and to the initial state) are lower bounds, and more accurate values are obtained during the forward

[2]Except for the resource restrictions reflected by the leveling of actions.

[3]Note that per-layer expansion can be considered a special case of the more general A*-like algorithm.

propagation of numeric expressions.

Second, the purpose of A* search is to construct an actual *path* from one state to another. The purpose of the envelope graph is to prune unreachable states and to provide lower bounds on the cost of achieving states. The envelopes are organized into a hierarchy (**T4**), which allows Sekitei to use bounded (incomplete) expansion of the graphs (**T3**).

The same bounded bidirectional A*-like expansion algorithm is used for all three levels of the envelope hierarchy. Below we describe the purpose of each envelope. Then, we illustrate construction of all three graphs using an example.

6.3.2 Cost of propositions

The first-level envelope — per-Proposition Logical Regression Graph (PLRG) estimates the minimum logical cost of achieving a proposition from the initial state and identifies the set of relevant operators [60]. Compared to RPG1 of Sekitei 2, PLRG is extended to support non-unit operator cost. Since the PLRG only considers logical preconditions and effects, its cost estimates are a lower bound on the actual cost of achieving a proposition, and therefore can be used as an admissible heuristic in the later stages of the algorithm.

The PLRG is expanded from the goal state until a solution is obtained, a bound is reached, or no further expansion is possible. The latter implies that the goal is logically unreachable from the initial state, and the problem has no solution.

6.3.3 Cost of sets of propositions

The second-level envelope — per-Set LRG (SLRG) — estimates the minimum logical cost of a set of propositions. The nodes of the SLRG correspond to sets of propositions. New nodes are generated by regressing over operators. Sekitei 2's RPG2 is also built for sets of propositions. The difference between RPG2 and SLRG is that RPG2 can contain several nodes for a given set of propositions that differ *in cost* of achieving them, while SLRG can contain several nodes for a given set that differ *in resource levels achieved*.

During the leveling of operators, the level propositions are created for all resource variables mentioned in operators. However, only levels of interfaces need to be *achieved*, and the rest are only checked. Only operators that achieve such **important** propositions are used for branching.

The SLRG computes set costs for important propositions only. For each important proposition, the best achievable levels of unimportant resources are computed in the PLRG. This information is then used to improve estimates of the cost of achieving sets of important propositions.

6.3.4 Main regression graph

The final level of the envelope hierarchy of Sekitei 3 is the main regression graph (MRG). The MRG contains totally ordered plan tails. The logical cost of achieving a set of propositions is used as an estimate of the remaining cost.

Each MRG node has an operator and a set of propositions describing the state in which the operator is to be executed. The operator of a node needs to achieve at least one proposition of the parent node.

Whenever a new node is created by regressing the current cheapest node over an operator, the plan tail including this operator is replayed in the optimistic map of this operator. If the execution fails, the new node is pruned from the search. Such partial execution allows early detection of violations of quality-of-service requirements, and, for example, discarding of partial plans whose total latency exceeds a given limit.

The optimistic map contains intervals for all resource variables required by the operator as specified by its leveled resource preconditions. Before execution of each successive operator in the plan tail, the interval produced by execution of the previous operator is intersected with the optimistic interval of the current operator, and new optimistic intervals are added if necessary.

The main difference between SLRG and MRG is propagation of resource maps in the MRG. Since resource failures depend on the plan tail, it is not possible to reuse nodes in the MRG. The MRG is a tree, while the PLRG and SLRG are general graphs.

The search in the MRG ends when all propositions, both important and unimportant, are present in the initial state, and the plan tail successfully executes in the resource map of the initial state.

6.3.5 Example

Figure 6.8 repeats the example from the beginning of this chapter. Due to limited CPU and bandwidth resources, a greedy approach fails to find a solution in this scenario. In this section we assume the resource discretization shown in Table 6.1.

Figure 6.9 shows a portion of the PLRG for the problem in Figure 6.8. Recall that during leveling of operators, each operator is executed in the optimistic map formed

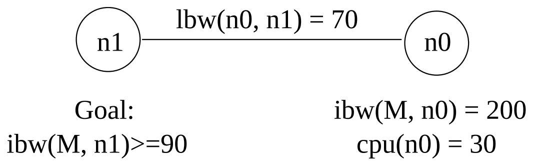


Figure 6.8: Resource optimization is required to find a plan.

by levels of required variables, such as bandwidths of a link and of an incoming stream for the link crossing operator. The resulting map is then compared with the expected values prescribed by the level intervals for the output data. Operators for crossing the link with the M stream with levels above the first one fail this test, because it is not possible to deliver more than 70 units of interface bandwidth over a link of bandwidth 70 regardless of the bandwidth of the incoming stream.

Without these link crossing operators, the cheapest way to achieve the proposition `L(ibw(M,n1))=2`, which states that the M stream bandwidth on host `n1` is in the second level interval [70,90], is to use Splitter and Merger components.

The PLRG consists of operator and proposition nodes, and thus contains information about logical support. When estimating the cost of a proposition, the cost of a proposition node is taken as the minimum of the costs of supporting operators, and the cost of an operator node as the maximum cost of its preconditions plus the cost of the operator. For example, the logical cost of achieving the proposition `placed(Cl,n1)` in Figure 6.9 is 18. Obtaining this cost requires sending both image and uncompressed text streams over the link (`cross(T,n0,n1)` and `cross(I,n0,n1)`). This would lead to violation of client's bandwidth requirements, but this fact cannot be detected in the PLRG.

The estimate of the cost of a set of propositions by the SLRG is more accurate

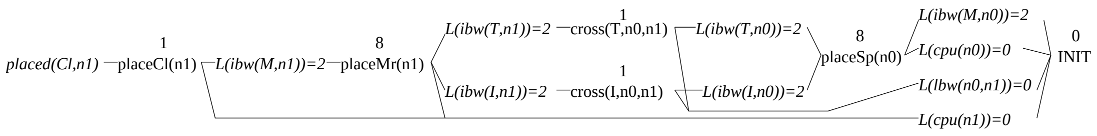


Figure 6.9: A part of the PLRG for the problem shown on Figure 6.8. The notation *L(v)=n* means that the resource variable *v* has level *n*. Numbers above operator nodes show costs of those operators given the resource levels. `Cl` stands for Client, `Sp` for Splitter, and `Mr` for Merger.

than that obtained directly from the PLRG. For example, the cost of achieving a singleton set {`placed(Cl,n1)`} is 19, because the two link crossing operators `cross(T,n0,n1)` and `cross(I,n0,n1)` are now considered in sequence rather than in parallel.

SLRG can also detect some conflicts that PLRG misses. For example, if both interface bandwidth and link bandwidth are leveled, it may be possible to detect in the SLRG the fact that sufficient levels of text and image streams cannot be delivered by using `cross(T,n0,n1)` and `cross(I,n0,n1)` operators, because both of them also decrease the level of available link bandwidth.

Figure 6.10 shows the third-level graph for our problem. In this figure, arrows connect operators to propositions they achieve. Propositions of a node include preconditions of the node's operator (underlined) and unsatisfied preconditions of subsequent nodes. In the resource maps, dashed lines mark newly added optimistic intervals, and solid lines show values added as a result of operator execution. For example, at the node corresponding to execution of the `cross(I,n0,n1)` operator, the value of `ibw(I,n0)` is added to the map to evaluate preconditions of the operator, the value of `ibw(I,n1)` is obtained as an effect of the operator execution, and the value of

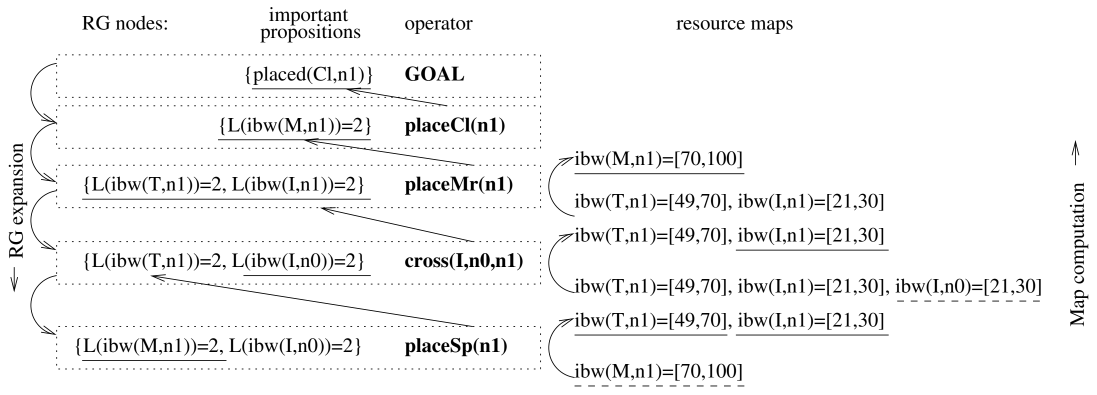

RG nodes:
important propositions
operator
resource maps
{placed(Cl,n1)}
GOAL
{L(ibw(M,n1))=2}
placeCl(n1)
{L(ibw(T,n1))=2, L(ibw(I,n1))=2}
placeMr(n1)
{L(ibw(T,n1))=2, L(ibw(I,n0))=2}
cross(I,n0,n1)
{L(ibw(M,n1))=2, L(ibw(I,n0))=2}
placeSp(n1)
ibw(M,n1)=[70,100]
ibw(T,n1)=[49,70], ibw(I,n1)=[21,30]
ibw(T,n1)=[49,70], ibw(I,n1)=[21,30]
ibw(T,n1)=[49,70], ibw(I,n1)=[21,30], ibw(I,n0)=[21,30]
ibw(T,n1)=[49,70], ibw(I,n1)=[21,30]
ibw(M,n1)=[70,100]
RG expansion
Map computation

Figure 6.10: Propagation of resource maps in the MRG. The shown portion of the graph does not contain a solution. Section 6.3.6 describes how such dead paths are pruned by Sekitei.

`ibw(T,n1)`, which is produced by the previous operator and consumed later in the plan, is unmodified.

6.3.6 Additional pruning

Several additional solution-preserving heuristics are used to prune search based on mutual exclusion relations between resource levels and repetitions of the logical state.

Important and unimportant propositions. Resource discretization leads to explosion of the number of propositions in the planning problem. Only part of them needs to be used in branching.

For example, the operator for placing an Unzip component can use values for available CPU and bandwidth of the incoming zipped stream. The level of bandwidth of the zipped stream needs to be achieved, meaning that if the zipped stream is not already present on the required node with sufficient bandwidth, additional operators, such as placing a Zip component or transferring the zipped stream from another node, need to be added to the plan. The value of the CPU, on the other hand, needs to be only checked, and operators that modify the value of the available CPU, such as placing another component on the same node, need not be considered.

In the CPP, the distinction between the important and unimportant resources is obvious (only interface properties are important). In traditional metric planning problems, all resource values are considered unimportant, and branching is performed based only on Boolean variables. Sekitei 3 requires the compiler module to mark all propositions as important or unimportant, and uses this information in search (see Appendix B for the specification of the Compiler interface).

Logical mutexes. In all graphs of Sekitei 3, different levels of the same resource variable are mutually exclusive. Operators that have preconditions mutex with the current state, or produce a logically inconsistent state are pruned from the search.

Repetition of logical state. Suppose a new operator op_1 to be added to the plan tail has a set of important logical preconditions S_1. Suppose there exists another operator op_2 in the same plan tail whose purpose is to achieve the same set of important propositions S_1, i.e., the intersection of important effects of op_2 with the logical state of op_2's parent node equals S_1. If the resource map resulting from execution of op_2 is contained in the optimistic map of op_1, then adding op_1 to the plan tail will create a *useless loop*, and thus can be pruned from the search.

Consider the example in Figure 6.10. The new operator `placeSp(n1)` to be added to the plan tail has precondition `L(ibw(M,n1))=2`. Operator `placeMr(n1)` already present in the plan has this proposition as its effect. Comparing the optimistic map of `placeSp(n1)` with the map resulting from execution of `placeMr(n1)` we can determine that adding `placeSp(n1)` does not improve the resource situation in any way, and therefore can be pruned.

Permutations of a DAG. Considering sequences of operators has the danger of evaluating all permutations corresponding to the same DAG. For example, it does not matter in our scenario which stream, text or image, is delivered to host `n1` first. We therefore prune a new node of the MRG if it corresponds to a permutation of another node already present in the graph (both the states and the plan tails are compared).

6.4 Evaluation

Extending the basic model of the CPP with cost functions and resource levels pursues two goals: allowing the planner to find solutions in resource constrained situations (Scenario 1 in Section 6.1) and specifying preferences over plans (Scenario 2). We decided to achieve this functionality by optimizing a cost function depending on resource consumption. Given an approximation of actual resource values by discrete levels, our algorithm optimizes the minimum cost of the plan instead of the exact cost. However, in our examples this approximation was sufficient.

The ability of our planner to achieve the desired functionality depends greatly on the actual specification of levels. Without levels, or with a poor choice of values for levels, the benefits from additional functionality are lost (however, solutions found by the planner are still correct). On the other hand, using multiple levels for each resource increases the size of the problem and negatively affects performance of the planner.

The following experiments show how the choice of levels affects scalability of the planner and quality of solutions.

6.4.1 Experiment

We tested the planner on the webcast application (Section 2.3.2) with three different sizes of the network and five different level specifications.

The CPP involves delivering a media stream from the server to the client (see Appendix A.2.1 for detailed specification of components). Locations of both the server and the clients are given. The client requires at least 90 units of bandwidth of the media stream, and the server is capable of producing up to 200 units. The costs of component placement and link crossing are proportional to the processed/transfered bandwidth. Such definition of the cost favors application configurations with the minimum number of additional components and the minimum bandwidth consumption along the data path.

The three networks used in our experiments have the following distribution of resources. LAN links of the networks have bandwidth 150 units, WAN links 70 units. Given the assumed models of resource consumption, the CPU resources on all nodes are sufficient for placing Splitter and Zip (or Unzip and Merger) components to process up to 111 units of the media stream. The *Tiny* scenario corresponds to the two-host network shown in Figure 6.8. Given any cost function that favors using less bandwidth if possible and does not differentiate the cost of component execution on different hosts, the plan in this case contains 7 actions (the six actions shown on Fig-

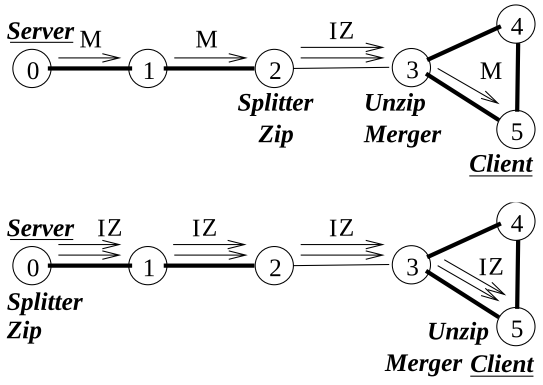


Figure 6.11: Suboptimal and optimal plans for the *Small* network.

ure 6.2 plus the client placement). The *Small* scenario involves a 6-host network. The shortest plan has 10 actions and cost of 72 (Figure 6.11 top). Since the media stream is sent over the LAN links, the bandwidth required there is 90 units. The optimal plan has 13 actions and a cost of 63 (Figure 6.11 bottom). Given the functions used in this experiment, this plan requires only 27+31.5=58.5 units of bandwidth of LAN links. Finally, the *Large* scenario corresponds to the similar problem in the N_{99} network used in the previous chapter. Most of the hosts of this network do not participate in the plan, but cannot be statically pruned.

Table 6.2 shows the five resource scenarios. Scenario A corresponds to Sekitei 2 (without resource levels). In this scenario, the limited network resources prevent the planner from finding any plan. Table 6.3 shows experimental results for the other four scenarios on each of the three network configurations.

Scenario	Levels of bandwidth of M	Levels of link bandwidth
A	$[0, \infty)$	$[0, \infty)$
B	$[0, 100)$, $[100, \infty)$	$[0, \infty)$
C	$[0, 90)$, $[90, 100)$, $[100, \infty)$	$[0, \infty)$
D	$[0, 30)$, $[30, 70)$, $[70, 90)$, $[90, 100)$, $[100, \infty)$	$[0, \infty)$
E	$[0, 30)$, $[30, 70)$, $[70, 90)$, $[90, 100)$, $[100, \infty)$	$[0, 31)$, $[31, 62)$, $[62, \infty)$

Table 6.2: Resource level scenarios. Bandwidth levels of interfaces T, I, and Z are proportional to those of the M stream.

6.4.2 Quality of solution

Even a single cut point 100 introduced in scenario B, which puts an upper bound on resource consumption, allows the planner to find a solution where only 100 units of the available stream are processed. However, in the *Small* and *Large* networks the found application configuration is suboptimal with respect to the reserved LAN link bandwidth (Column 4).

To ensure that the found plan is optimal with respect to the user-specified cost function, the lower bound on the cost function (Column 2 in the table) obtained by the planner needs to approximate the real cost of the plan as close as possible. The level specifications of scenarios C, D, and E allow the planner to select the best configuration.

The plans selected in scenarios C, D, and E involve processing 100 units of bandwidth of the M stream, which is more than strictly required to satisfy the client's requirements. The best quality of a solution would be achieved if the bandwidth of the media stream is cut at two points exactly around 90. Obtaining such values automatically requires reversibility of resource functions. Scenario C approximates the ideal values: It selects the optimal configuration, but requires slightly more resources

Scenario		*Quality of the solution*			*Work done by the planner*				
		lower bound on cost	operators in plan	reserved LAN bw	total # of operators	graph sizes (total/unexpanded)			planning time (ms)
						PLRG	SLRG	MRG	
1		2	3	4	5	6	7	8	9
Tiny	B	7	7	N/A	32	44 / 0	24 / 2	25 / 5	261 / 70
	C	42	7	N/A	46	43 / 0	25 / 4	26 / 8	280 / 80
	D	42	7	N/A	76	43 / 0	25 / 4	26 / 8	370 / 80
	E	42	7	N/A	174	65 / 0	51 / 10	39 / 14	380 / 80
Small	B	10	10	100	152	122 / 5	417 / 181	344 / 196	811 / 470
	C	63	13	65	222	185 / 0	188 / 80	178 / 79	741 / 381
	D	63	13	65	364	185 / 0	188 / 80	178 / 79	801 / 340
	E	63	13	65	1172	433 / 0	2851 / 1282	1136 / 410	2524 / 1692
Large	B	11	11	100	2564	3010 / 35	3305 / 2095	1751 / 1390	17655 / 1742
	C	63	13	65	3750	3080 / 0	1046 / 704	404 / 244	17505 / 1011
	D	63	13	65	6144	3080 / 0	1046 / 704	404 / 247	19968 / 610
	E	63	13	65	20712	7680 / 0	66518 / 48926	3946 / 2344	37944 / 11847

Table 6.3: Scalability evaluation of Sekitei 3.

than absolutely necessary. To ensure correctness of the solution, the planner decides to process 100 units of stream bandwidth on the server side (the upper bound of the chosen interval), which requires 65 units of link bandwidth. The optimal values, as discussed above, are 90 and 58.5 respectively.

6.4.3 Scalability

Table 6.3 also provides information on scalability of our algorithm. Column 5 gives the total number of operators evaluated after leveling and using the pruning procedure. The sizes of the three graphs characterize memory requirements of the planner. For each graph, the table gives the number of generated nodes and the number of nodes in the A* queue at the moment when a solution is found. Column 9 shows the total running time of the planner including time spent reading problem files and constructing operators (the first number); and the portion of this time spent in search and construction of the graphs (the second number).

As the results show, introduction of resource levels improves both the quality of the solution and the performance of the algorithm. Although adding resource levels to the problem specification significantly increases the number of generated operators, it also permits identification of some resource conflicts at earlier (and cheaper) phases of the search, which explains the improved performance of Scenario C compared to Scenario B and in several cases that of scenario D over C.

Adding more levels of interface bandwidth (scenario D) and leveling link bandwidth (scenario E) does not always improve the quality of solution, but negatively affects performance of the planner. The good choice of levels depends on requirements of application components and on the definition of the cost function. In the presence of non-reversible resource functions the choice of levels needs to be performed by a domain expert, possibly, based on profiling results. Although not demonstrated in this experiment, we expect that for some problems it might be beneficial to discretize additional resources such as link bandwidth and node CPU.

6.4.4 Limiting expansion of the PLRG

The main purpose of the PLRG is to identify relevant portions of the search space. Like the other two graphs of Sekitei 3, PLRG construction uses the bounded bidirectional A* algorithm, and can take advantage of a heuristic estimate of the lower bound of the cost of proposition to focus the search and bound the size of the graph.

In our implementation, we use the length (the number of links) of the shortest path to a server as such a lower bound. However, this seems not to be enough.

As column 6 of Table 6.3 shows, in all three of our test cases PLRG expands to include almost all hosts of the network for all resource levels that may be used in a solution. This includes all levels that were not pruned during the leveling of operators and were picked later by the envelope-construction procedure. We tested the algorithm on a bigger, 600-node network shown in Figure 6.12 and observed similar results.

The reason for the complete expansion of the network is the large difference between costs of data transfers and component executions in our evaluation applications. The bounds on expansion, provided to PLRG by SLRG and MRG, take into account the cost of components. By the end of the search, SLRG considers nodes with costs greater than 60 (the cost of the final plan is 63). Recall that the earlier levels of the envelope hierarchy need to be expanded enough to make a correct choice at the later

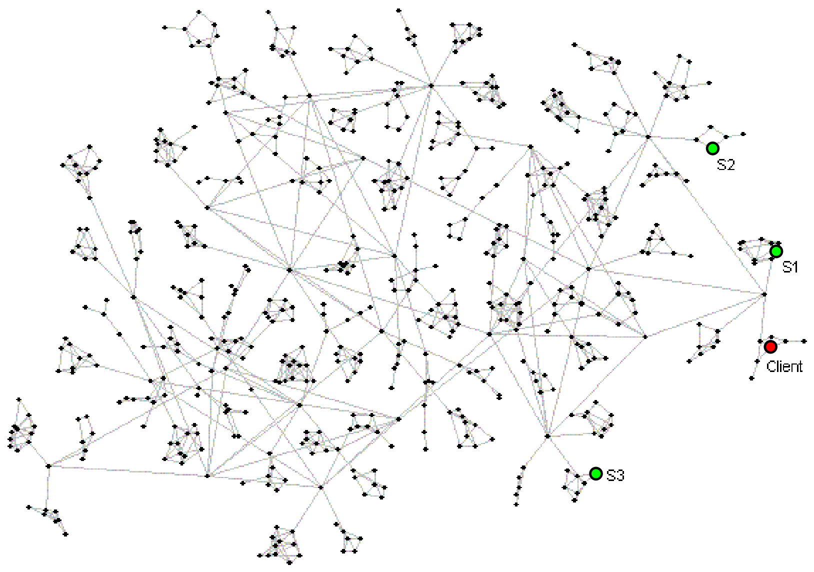


Figure 6.12: The 600-host network. The locations of the client and the three servers used in the experiment in Section 6.4.4 are shown.

levels. Therefore, the expansion bound for the PLRG eventually exceeds 60.

As opposed to the SLRG, the PLRG does not detect the fact that some components may be necessary to build the final plan. Instead, to obtain lower bounds on costs, the PLRG essentially counts only network hops. The minimum cost of a data transfer in our example is 1. Therefore, to guarantee optimality, PLRG needs to include all hosts within 60 hops from the client host, which includes the whole network.

One way of addressing this expansion is by using a hop distance multiplier comparable to the cost of component execution. The use of a multiplier in the PLRG is similar to the idea used in any-time weighted A* search [68]. The effect of the multiplier is that it biases the search towards direct paths, thus limiting expansion of the PLRG in the network and improving the planning time. Table 6.4 shows results for three locations of the server marked in Figure 6.12 and version C of resource levels.

Scenario		Solution characteristics			Graph sizes (total/unexpanded)		
Server	multiplier	cost	length	planning time	PLRG	SLRG	MRG
S1	0	63	13	26148 / 25857	33273 / 4801	4632 / 3795	1176 / 884
	1	63	13	8282 / 4136	3211 / 554	5764 / 4820	1315 / 1007
	3	63	13	8522 / 4416	3510 / 1071	5705 / 4796	1148 / 869
	10	63	13	10516 / 6410	5596 / 156	5705 / 4796	1148 / 870
S2	0	70	15	26658 / 26368	33282 / 4801	13404 / 11288	165 / 149
	1	70	15	14061 / 9945	8472 / 2149	18448 / 15700	218 / 202
	3	70	15	15773 / 11637	11075 / 2638	18046 / 15370	218 / 202
	10	70	15	16544 / 12418	14113 / 3047	18456 / 15710	218 / 202
S3	0	77	17	34299 / 34009	33282 / 4801	33879 / 27964	210 / 192
	1	77	17	23434 / 19298	15747 / 3261	44705 / 37508	267 / 249
	3	77	17	36242 / 32096	33282 / 4801	44417 / 37255	267 / 249
	10	77	17	26167 / 22011	22196 / 4137	44792 / 37538	267 / 249

Table 6.4: Effect of hop distance multiplier on search efficiency.

Note that the multiplier makes the resulting estimate of the remaining cost no longer admissible and theoretically can cause loss of optimality. However, in all of our examples the plans found were exactly the same as in the case of the complete network expansion. Once again, using multiple phases of the search appears to compensate for imprecise computations.

6.5 Summary

This chapter concludes our discussion of how the techniques **T1–T6** can be used to design an efficient planner for the component placement problem without placing artificial restrictions on the expressiveness of the supported model.

Sekitei efficiently prunes irrelevant parts of the search space and demonstrates good scalability in the presence of non-reversible numeric functions. In addition, our planner can perform approximate optimization of resource consumption given a discretization of continuous resources. Our aim in developing Sekitei was to support maximum expressiveness of the model. However, our planner makes several assumptions.

First, all functions need to be monotonic in the sense described in Section 3.2.1. This assumption requires that an interval of results is computable given intervals for parameter values. Most functions can be implemented so as to satisfy this assumption. Moreover, resource levels can be used to make the function piecewise monotonic.

Second, all functions need to be forward computable. In some cases it is useful to be able to describe resource behavior of components using equations involving both input and output parameters. Such functionality can be implemented using general constraints instead of functions. However, such an extension of the model would prevent us from using envelope hierarchies, which provide the basis for Sekitei's efficiency.

Third, Sekitei assumes that all numeric values are precise. In practice, measurements are noisy. One would ideally like to accommodate such imprecision, e.g. by performing sensitivity analysis. We will revisit this issue in Chapter 8.

Finally, the scalability and the optimization power of Sekitei 3 depend on the choice of resource levels. It is hard to determine good levels automatically, which means that help of a domain expert may still be required when new component types are introduced to the system. We believe that it is possible to improve the quality of solutions produced by Sekitei by using an optimization phase [57, 86] after the envelope-based pruning. Such a numeric optimization phase would ignore the resource discretization, and therefore compensate for the unavoidable imperfections in the choice of resource levels.

Chapter 7

Planning for grids

In the previous chapters we introduced a set of techniques (**T1–T6**), which provide the basis for Sekitei — an AI-style planner for solving the component placement problem. In this chapter we show how the same techniques can be used for solving a different variety of the application configuration problem — construction and scheduling of workflows for computational grid applications. We also describe how complex constraints can be used to reason about simultaneous reservation of multiple resources (**T7**).

Because of the high computational complexity of application configuration, to achieve good performance, a planner for the ACP needs to take advantage of natural restrictions of a particular variety of the general problem the planner is designed for. For example, the component placement problem is characterized by the absence of the time aspect (all configurations are snapshots) and small number of parallel data streams. These features allow Sekitei to consider data transfer operations separately for each link, and to use a per-set resource propagation to achieve good performance on problem instances involving large networks and non-reversible resource functions.

In this chapter we present an algorithm for solving the application configuration problem in the computational grid setting. Grid applications may involve many (up to hundreds) sequences of jobs that can be executed in parallel given sufficient resources. Therefore, a planner for grids needs to reason about sharing of resources both in network space and physical time. On the other hand, the size of the network is usually small compared to that of the CPP. Grid applications we consider are computationally expensive and require specialized hardware (e.g., a supercomputer) for execution. Therefore, the total number of hosts capable of running components is usually small (in the tens).

Despite these difference between the CPP and grid planning, the techniques used for solving the CPP can be used to design a planner for grids, which supports an expressive problem model and demonstrates high performance.

The rest of this chapter is structured as follows. In section 7.1 we describe our model for grid applications and the assumptions made by the planner. Section 7.2 describes the GPRS algorithm (Grid Planner with Reservations and Sharing), and section 7.3 evaluates its performance. Finally, in section 7.4 we review the basic principles behind our solution.

7.1 Model

A configuration of a grid application is a workflow mapped onto underlying network resources. The workflow consists of jobs (component instances) that require and produce files. Jobs can be executed on network hosts, and files can be transferred over network links. Thus, planning for grids is an instance of the general application configuration problem described in Section 3.1.

7.1.1 Network

The network consists of hosts connected by links. Together, hosts and links are referred to as network resources.

Network resources have *lists of properties* associated with them, such as the CPU speed, amount of available memory, link bandwidth. The properties of network resources can change over time. We assume that each property can be represented by a piecewise constant function.

A network resource can be reserved to perform an action. Such reservation is specified by the start time, duration, and a list of required property values. An active reservation may change the values of the required properties, e.g. link bandwidth can be consumed for the duration of a file transfer. Alternatively, the properties may not be affected by the reservation, but be required to have specific values, e.g. a link may be required to be secure. The latter can be viewed as a special case of the former, therefore, in this chapter we focus on the more general case of consumable properties.

Multiple actions can be performed on a resource in parallel, assuming the underlying OS/scheduler provides such functionality. For example, different sets of processors of a multiprocessor machine can be allocated to different jobs. However, at any given moment, for each property, the total value required by all active actions cannot exceed the available value at that moment.

7.1.2 Jobs and files

The element of data in grid planning is a file. A file is uniquely identified by its logical file name (LFN) [19], a network-wide identifier for this file. For each ground LFN, the size of the file can be computed, perhaps by consulting an external service. There may be multiple instances (replicas) of the same logical file on different network hosts. The planner's decision of which replica to use depends on the structure of the workflow and on available resources.

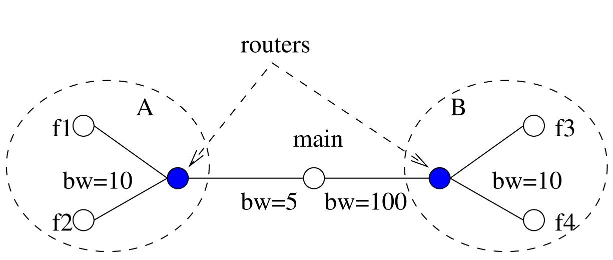
A single execution of a component instance in grids is referred to as a job. A job can be executed on a network host. During execution, the job occupies some portion of the host resources. Upon job completion, these resources are released.

A job requires a set of logical files and produces a set of logical files. For a job to be executed on a host, replicas of all required files should be present on the host before start of the job. Upon job completion, instances of all files produced by the component become available on the host.

File instances can be transfered between network hosts. The transfer is performed along a network path chosen by some external entity, e.g. a router. During the transfer, the same value of bandwidth (path bandwidth) is reserved on all network links participating in the path.

To model sharing of links in network paths, some hosts can be designated as routers. Routers are unable to perform computation, which is specified by zero availability of host resources, such as CPU. For example, the network in Figure 7.1 consists of two clusters, each of which is connected to the `main` host via a router.

The four end hosts in Figure 7.1 each store a file with size 10 units. As with



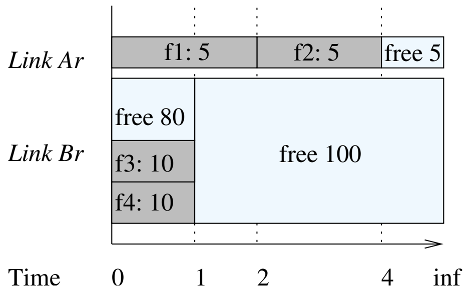


Figure 7.1: Parallel and sequential reservations of link bandwidth. Transfers of files `f1` and `f2` are scheduled sequentially over low bandwidth link `Ar`, while transfers of files `f3` and `f4` are performed in parallel over a high-bandwidth link.

Sekitei, we assume that the on-demand compiler (**T1**) module normalizes units of measurement. All four files need to be delivered to host `main` as soon as possible. Suppose internal links in both clusters have bandwidth 10 units, the link from the router A (`Ar`) to the main host has bandwidth 5 units, and the link from router B (`Br`) to the main host has bandwidth 100 units.

The transfer of files from cluster A to the main host is limited by the bandwidth of the link `Ar-main`. In our model, two sequential data transfers will be scheduled for this portion of the network, each involving two links (end host-`Ar` and `Ar-main`) and reserving 5 units of bandwidth. The file transfers from cluster B can be done in parallel, each reserving 10 units of bandwidth of the involved links. Both files `f3` and `f4` can be delivered to the main host within 1 time unit, while transfer of files `f1` and `f2` will take 4 time units all together.

Currently we do not model temporary storage of files, i.e. once a file is transfered to a host or is created as a result of a job execution, the file is always available on that host. The amount of necessary storage is defined by the computation, which is limited by host and network resources. The current implementation of the planner assumes that a host will have enough space to store all data products of the computation, and that created files will be available on the hosts until completion of the workflow.[1]

7.1.3 Metrics

A grid planning problem is specified by a network; a set of job types, each specified by input and output files and resource property requirements; the initial availability of files on network hosts; and a goal specified as a pair of a ground LFN and a host. The goal of the planner is to construct an executable workflow that will make the required file available on the required host as soon as possible.

To construct the fastest workflow, it may be necessary to decide how to obtain each of the intermediate files. In the example shown in Figure 7.2, the goal is to obtain a small file `Result` on host 3. `Result` can be computed in 1 time unit given two files `pA` and `pB`, each of size 100. Each of these files can be computed in 2 time units from files `rA` and `rB` respectively. All four files, `pA`, `pB`, `rA`, and `rB`, are present in the network.

Given the available link bandwidth shown in Figure 7.2, the optimal strategy is to transfer `pA` from the remote host and recompute `pB` using the `rB` file. Note that, if the execution of the component producing the final result is co-located with the recomputation of `pB`, the time used to transfer `pA` overlaps with the computation of `pB`. Usually it is possible to overlap computation and communication.[2]

In the above example, finding an optimal solution requires trading off computation

[1]In the current implementation, there is a Boolean flag that specifies that a host does not have any storage.

[2]Our model permits modeling of reservation of host resources for data transmission/reception.

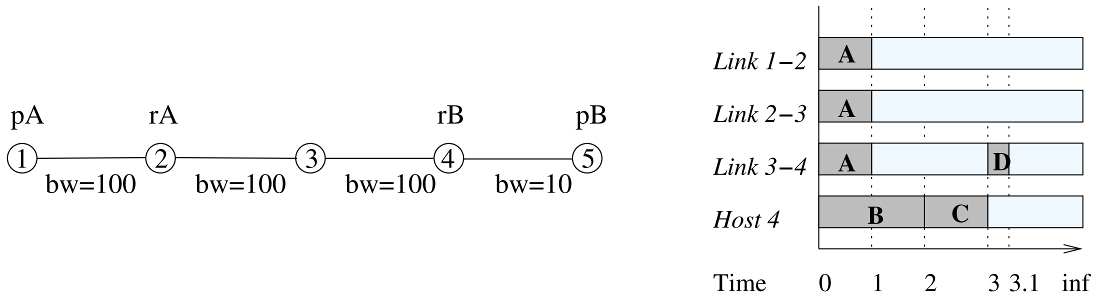


Figure 7.2: Trade-off between computation and communication. Because of the low bandwidth of link 4-5, an optimum configuration requires transferring file `pA` and recomputing file `pB`. The actions of the plan are: **A** transfer `pA` over path 1-2-3-4; **B** recompute `pB` on host 4; **C** compute `Result` on host 4; **D** deliver the result to host 3.

and data transmission, thus changing the structure of the workflow. This additional degree of flexibility distinguishes the grid planning variety of the ACP from traditional scheduling. In addition, when the underlying grid infrastructure supports explicit reservations, the planner needs to take into account time-varying resource availability, which makes the problem even harder.

7.1.4 Compiler module

Similar to Sekitei, GPRS employs the on-demand compilation technique (**T1**) to keep the planner independent of the framework and uses an envelope graph (**T3**) to obtain lower bounds on the completion time of parts of the workflow.

For construction of the envelope graph, the compiler needs to be able to create a graph node for the goal, determine if a particular data node is present in the initial state, and return a list of operator nodes that can achieve a given data node.

A grid-specific implementation of the compiler is aware of data transfers and job executions, but makes no assumptions about the job resource requirements and names of the required and produced files.

Depending on a particular application, a user needs to create one or more classes for jobs (components) and file types. Each file type should be able to compute file size for a given instance of the logical file. Each component type should be able to construct a job node given a host and a desired data product, or return an error if the host does not satisfy the component's requirements.

Using the on-demand compilation technique (**T1**) in this way allows us to use GPRS for applications with complex logical structure, such as that of the Montage application described in Section 2.3.3, without making any additional assumptions in the planner.

7.1.5 Implementation details

The Java implementation of GPRS used in this chapter for illustration and evaluation makes several additional assumptions about the problem.

First, property lists are limited to one property only (CPU for hosts and bandwidth for links). This limitation is imposed by the current implementation of the compiler module, and not by the algorithm.

Second, numeric resources are assumed to be released upon completion of an operator. In practice, some resources, such as quotas, are strictly consumable. The algorithm can be easily extended to support such resources.

Third, for the purpose of evaluating GPRS, the duration of a file transfer is computed as the file size divided by the path bandwidth. This computation is represented as a separate procedure (constraint), and a more accurate model can be used where available.

Finally, the current implementation of the planner optimizes the completion time of the computation (makespan). Extension of GPRS to support more general metrics is a topic for future research.

7.2 Algorithm

At a high level, the algorithm performs critical path scheduling using an envelope graph similar to the per-proposition LRG of Sekitei 3 to bound exploration of the search space and obtain lower bounds on duration of computation.

7.2.1 Constructing the envelope graph

The envelope graph contains information about possible ways of achieving ground propositions. For clarity, we will use grid terminology in the presentation of the algorithm, even though the algorithm is not limited to grid planning. In the grid case, the ground propositions of the envelope graph correspond to availability of file instances on network hosts.

The graph has two types of nodes. **OR nodes** correspond to propositions (files). **AND nodes** correspond to operators (job executions and data transfers). For each OR node n_o, **support** of n_o is a set of AND nodes corresponding to operators that can achieve the proposition of the node n_o. For each AND node n_a, support of n_a is a set of OR nodes corresponding to preconditions of the operator of node n_a.

GPRS optimizes the makespan (total duration) of computation. Therefore, the

cost of a node is defined as the earliest completion time (ECT) for a job and earliest availability time for a file. In the rest of this section we use the term *cost* to refer to the ECT.

The algorithm admits arbitrary monotonically non-decreasing functions for combining costs (ECTs) of the support nodes to compute the cost (ECT) of the supported (sink) node. To support such generality, we create separate **variables** within each node describing time points and durations (Figure 7.3). These variables are connected by the means of functions referred to as **constraints**. Constraints are used to *propagate* changes of values of variables during envelope construction and solution extraction. Variables of envelope nodes, resources, artificial variables, and constraint connecting them form a **constraint network**. In the current implementation, the cost of an OR node is the minimum of the costs of supporting AND nodes, and the cost of an AND node is computed using a *reservation constraint* (RC) described in Section 7.2.3.

During the graph expansion, leaves of the graph are OR nodes. The cost of a leaf node is assumed to be zero. After a leaf node is expanded by creating or reusing AND nodes for all operators that can possibly achieve the corresponding proposition, a new estimate of the cost of the OR node is computed.

The cost combining functions may be represented as constraints that use external variables, such as resources, in addition to the costs of support nodes. Because the leaf costs are zero and the cost combining functions are non-decreasing, at any given moment during graph expansion, the estimated costs of nodes are lower bounds on the actual costs. A constraint propagation engine discussed below is used to obtain tight bounds of node costs.

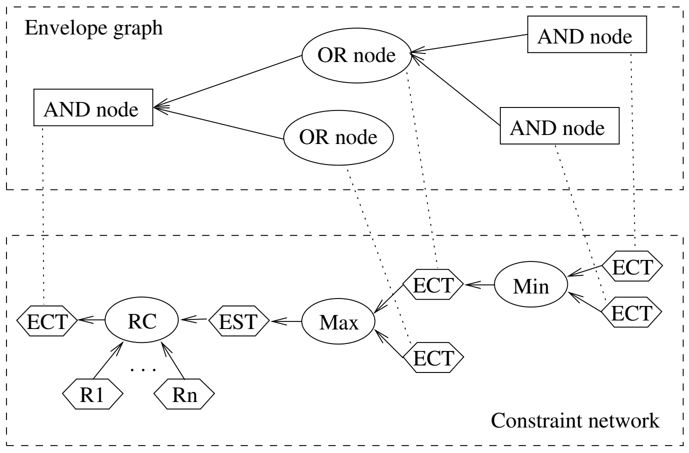
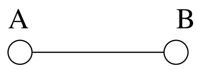


Figure 7.3: GPRS envelope and constraint network. Earliest completion time variables (ECT) of the envelope nodes, artificial variables, such as Earliest Start Time (EST), resources (R1 to Rn), and constraints form a constraint network for the envelope graph. Dotted lines connect the ECT variables to the nodes they belong to.

7.2.2 Constraint propagation

To ensure good performance, we require that constraints propagate values only in one direction (compare to the forward computability of functions in Sekitei, Section 3.2.1). A constraint may use several variables as sources, but all variables whose value the constraint can change should belong to the same node, referred to as the **sink** of the constraint. We also require that each variable is affected by at most one constraint. Artificial variables may be created if necessary to enforce the latter restriction.

During constraint propagation, every constraint whose source variable has changed since the previous quiescent state is recomputed. This operation can result in changing values of variables of the sink node. Constraint propagation is performed until



The goal is to have a copy of file `f` on host A. No job can produce the file, and the file is not available anywhere in the network, so the problem has no solution. Expanding the leaf node `f@B` creates an infinite cycle in the envelope graph.

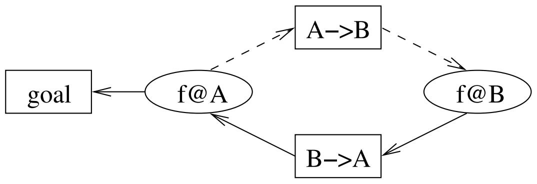


Figure 7.4: Envelope graph with an infinite loop

there is no more change, i.e. until a new quiescent state is reached.

To minimize thrashing during constraint propagation, the constraints are sorted by the decreasing distance of the changed source variable from the goal node. This is similar to propagation of cost updates in Sekitei.

Similar to PLRG and SLRG of Sekitei 3, the envelope graph of GPRS can contain loops. Remember that costs of nodes are lower bounds. Therefore, expansion of the leaf node, which increases the cost estimate of that node, may cause infinite loops in constraint propagation along graph cycles (see Figure 7.4 for an example). To avoid this, before expansion, the costs of all nodes reachable from the leaf node being expanded are set to infinity. Subsequent constraint propagation can only decrease the bounds, when possible. In case of an infinite loop, such as the one shown in Figure 7.4, the lower bounds of costs of unreachable nodes will remain infinity, and the constraint propagation will terminate after checking each node only once.

7.2.3 Reservation constraint

Completion of actions (job executions and data transfers) in grid planning depends on availability of required files, availability of network resources, and duration of the

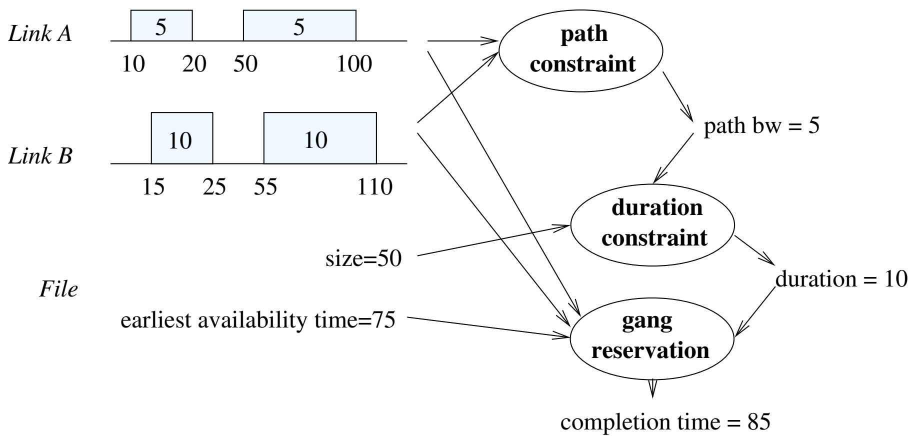


Figure 7.5: Implementation of the reservation constraint.

action. We represent this dependency using a **reservation constraint**. A reservation constraint has as sources the earliest start time and a set of resources (hosts or links). The earliest start time can be an artificial variable linked by a maximum constraint to earliest availability times of all required files (see Figure 7.3). A reservation constraint also has a list of the values of properties of the resources required by the action.

The GPRS planning algorithm can support different implementations of the reservation constraint. In the current implementation, the duration of an action depends only on the set of resources and not on the start time of the action, so for the purposes of the reservation constraint it can be considered constant. This allows us to implement a reservation constraint as three separate constraints connected using artificial variables, such as path bandwidth and duration of reservation. Figure 7.5 shows such an aggregate reservation constraint of a two-link network path for a file transfer.

Each of the network resources keeps a profile of property values as a piecewise constant function of time. Using this profile, a resource can compute the earliest time after a given moment when the given set of property values (produced by the path constraint) is available for the given length of time (produced by the duration constraint). The gang reservation constraint uses this facility to compute the earliest time when *all* resources are available. Gang reservation is important, for example, for scheduling data transfers over a multi-link network path.

The reservation constraint affects the completion time of an action. Note that because of the varying resource availability, the completion time of the action may be greater than the sum of the start time and duration.

7.2.4 Limiting transfer sequences

In Sekitei, where a snapshot application configuration is constructed, each network link is considered separately. This means that for each network host only its immediate neighbors are considered as possible targets for data transfers. This property allows Sekitei to perform graph expansion for networks with large numbers of hosts.

In GPRS, several links may participate in a single data transfer. This means that for a given network host every other host in the network may be considered as a target for a transfer. This can cause explosion of the number of data transfer operations and reservation constraints. In addition, loops consisting of multiple data transfer operations will be created, which significantly slows down constraint propagation.

However, the following observation provides a basis for a solution to this issue. Since in our model we do not explicitly consider temporary storage of data, sequences of data transfers are unnecessary. If a file can be transfered from host A to host B and then, without any intermediate computation, from host B to host C, then this file can be transfered directly from host A to host C (along the path A-B-C or some other path

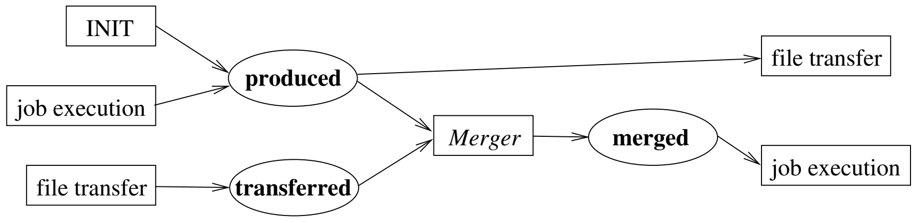


Figure 7.6: The relationship between three types of file nodes and types of actions. *Merger* is an artificial action used to connect file nodes of different types.

connecting the end hosts). We can exploit this feature by creating three different types of file nodes: **merged**, **produced**, and **transfered** (Figure 7.6).

A merged node has as support a single node of a special type that combines a produced file node and a transfered file node. A data transfer action always produces a transfered node and requires a produced node. A job node requires a merged node and produces a produced one. Files initially available in the network correspond to produced file nodes.

Although this modification triples the number of file nodes in the graph, it prevents the planner from considering sequences of data transfers by requiring interleaving transfers with computation. This in turn reduces the number of cycles in the graph, achieves significant (more than an order of magnitude) speedup on large problems, and improves scalability of the algorithm with respect to the network size. Note that this optimization is local to the grid compiler and has no effect on the design of the core planning algorithm.

7.2.5 Critical path scheduler

The envelope graph is expanded until the **best tree**, defined recursively using the cheapest support node for each OR node and all nodes for each AND node, has all its leaves true in the initial state.

The envelope graph is optimistic, because it does not take into account interactions between different branches of the application DAG.[3] Therefore, the best tree of the envelope graph does not necessarily represent a valid solution.

To construct a valid solution, we use critical path scheduling for the final plan extraction phase of the GPRS algorithm. A **critical path** is a path in the best tree leading from the root (the goal node) to a leaf, which chooses the most expensive support node for each AND node.

The planner *commits* nodes of the critical path starting from the leaf. Committing a node involves making all resource reservations belonging to the node permanent. Such commitment of reservations is possible as long as the constraint network is quiescent.

As a result of committing resource reservations, the property value profiles of the involved network resources may change, which may affect other reservation constraints that involve the same resources, and, via constraint propagation, completion times of various nodes of the envelope graph. The change of completion value of the graph nodes may change the portion of the graph considered to be the best tree and therefore result in further expansion of the envelope graph.

The current implementation of GPRS does not support backtracking. Once a node

[3]Although, as we discuss in section 7.4, this can be changed.

is committed, the commitment cannot be revoked. Because the envelope graph is optimistic, such a non-backtracking nature of the solution extraction phase may result in suboptimal solutions. Moreover, the non-backtracking algorithm is incomplete and may fail to find a solution in the presence of budget restrictions (quotas on resource usage and/or deadlines).

On the other hand, the non-backtracking solution extraction phase is fast. Moreover, the use of the envelope graph during the solution extraction phase allows the planner to make continuous adjustments to the best tree including rescheduling actions and replacing whole subtrees. This flexibility usually leads to good quality of solutions, and appears to compensate in practice for the theoretical incompleteness of the algorithm.

In the next section we present a detailed evaluation of performance of GPRS. In section 7.4 we discuss some ways of improving the quality of solutions.

7.3 Evaluation

In this section we evaluate the ability of the GPRS to handle the expressiveness of the model of grid applications with explicit resource reservations and sharing, the scalability of the planner with respect to the network and application size, and the quality of solutions produced by the planner.

7.3.1 Handling expressiveness

A planner for computational grids with explicit reservations needs to reason about resource sharing in physical time and network space. This includes sharing of numeric resources between jobs and data transfers running in parallel. The planner also needs to reason about action durations and start and completion times in the presence of time-varying resource availability. Finally, the planner needs to be able to select different options, such as file replicas or component types capable of producing a given data product, and trade off computation and communication so as to minimize the total duration of computation. To check if GPRS can correctly handle the expressiveness of the model, we ran the following experiment, which exercises the features listed above.

The abstract structure of the application derived from Montage-like workflows is shown in Figure 7.7. The application consists of three jobs (components), organized in two levels. Each of the two jobs of the first level requires three input files, produces two output files, and takes 100 time units to complete. The third job requires four files and produces one file in 40 time units. The size of files `a`, `b`, `c`, `d`, and `e` is 500 units. Files `f`, `g`, `k`, and `q` are 100 units each. The final result file `r` has size 1000. Note that file `c` is required by two components, and file `k` can be produced by two different components. In the latter case the semantics is that the file `k` will contain exactly the same data regardless of which of the two jobs produced it.

The network structure for our problem is shown in Figure 7.8. Replicas of several files are available on different network hosts as shown in the figure. The bandwidth of the links connecting storage nodes to the storage router is at most 5; the bandwidth of all other links is at most 10 units. We assume that only host `Computer` can perform computation.

We further assume that the availability of the `Computer` host and the availability of the link `StorageL` between the main `Router` and the storage router `StorageR`

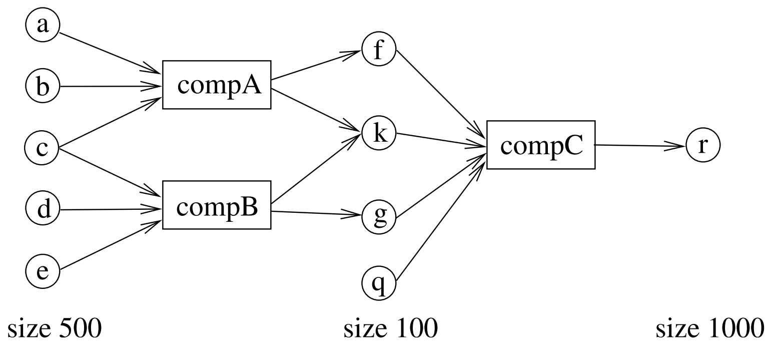


Figure 7.7: Abstract structure of a grid application. Circles represent files, and rectangles jobs.

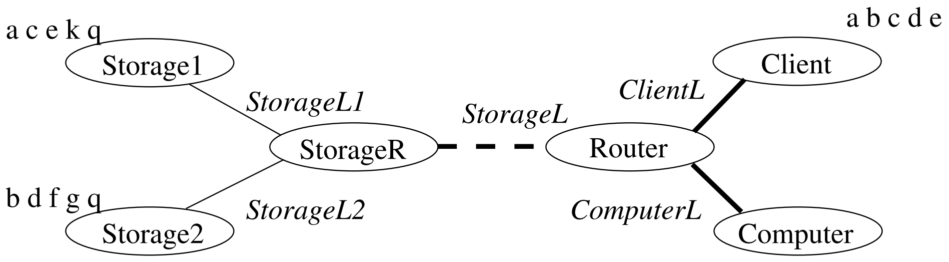


Figure 7.8: Network structure. Link names and shown in italics. Availability of file replicas is shown in normal font next to the hosts. Width of the lines corresponds to the link bandwidth. The dashed line shows the link, whose availability varies over time.

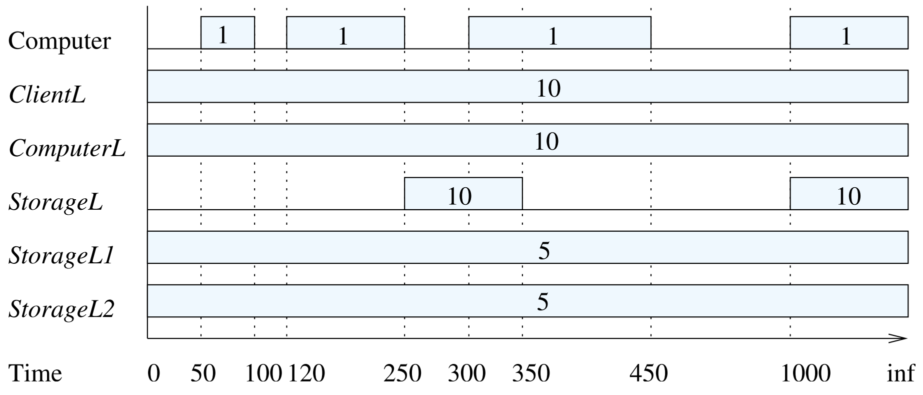


Figure 7.9: Resource availability. Numbers in the bars show the amount of the resources.

vary with time due to reservations from other ongoing computations. The availability windows for network resources are shown in Figure 7.9.

The goal of this problem is to obtain the `r` file on host `Client` as quickly as possible.

The plan found by GPRS is shown in Figure 7.10. This plan has the optimum duration given the resource availability. Note that because of the limited availability of both computational and link resources, the planner decided to recompute two of the intermediate data products and fetch the other two from where they are stored. The replica of file `q` is chosen so that the transfers of files `g` and `q` can be done in parallel.

The planner also correctly handles multiple reservations and sharing of resources (Figure 7.11). For example, transfer of file `g` from `Storage2` to `Computer` requires simultaneous reservation of bandwidth of three links. Two of these links (`StorageL` and `ComputerL`) are used for transfer of file `q` from `Storage1` at the same time.

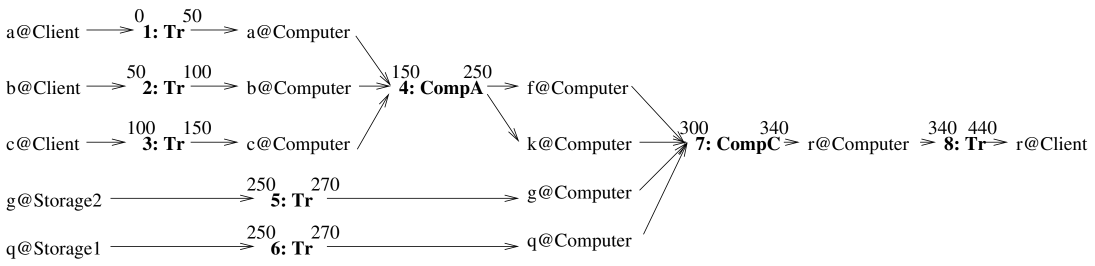


Figure 7.10: The final plan. File nodes describing availability of a file replica on a host are shown in normal font. Actions are shown in bold and preceded by a sequence number. The numbers above each action node are the start and end time of that action. The start time of an action depends on the earliest availability of the required files and on the resource availability.

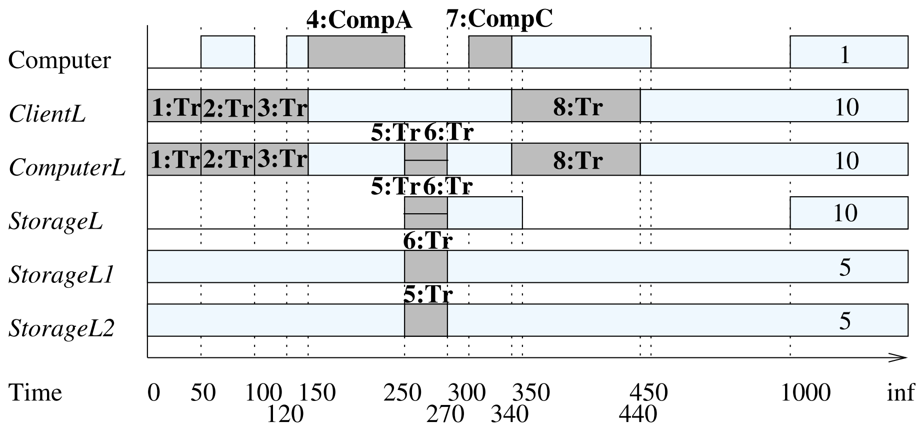


Figure 7.11: Resource reservations by actions. Multiple resources may be simultaneously used by a single action, and several actions can concurrently use the same resources (e.g. link *StorageL*).

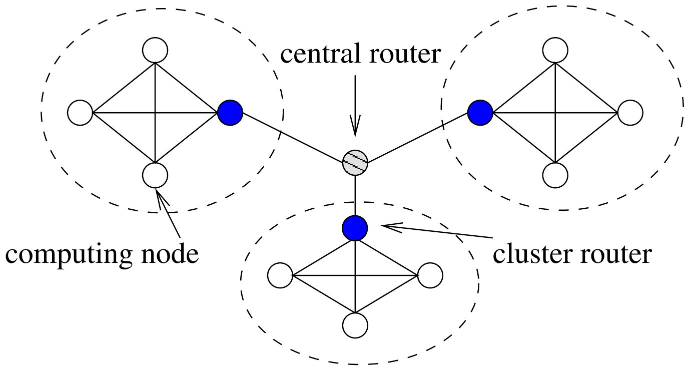


Figure 7.12: Synthetic network used in performance evaluation.

7.3.2 Performance

To evaluate the scalability of the planner we used synthetic applications and networks.

Figure 7.12 shows the network used in our experiments. This network consists of C clusters each containing N computational hosts and one router, which connects the cluster to the central master router. Hosts within a cluster are fully connected. Bandwidth of intra-cluster links is 10 units, the bandwidth between cluster routers and the central router is 30 units. In total, the network contains $(N+1)\times C+1$ hosts, of which $N \times C$ can perform computation.

Figure 7.13 shows the application kernel used in the experiments. The structure of this kernel is modeled after existing grid applications such as Montage [6]. This application kernel is parameterized, which allows us to analyze scalability of the planner with respect to different properties of the application.

The application consists of S segments limited by the splitting and merging components. The portion of the segment between these components contains W parallel execution sequences, each consisting of H processing jobs. An instance of this appli-

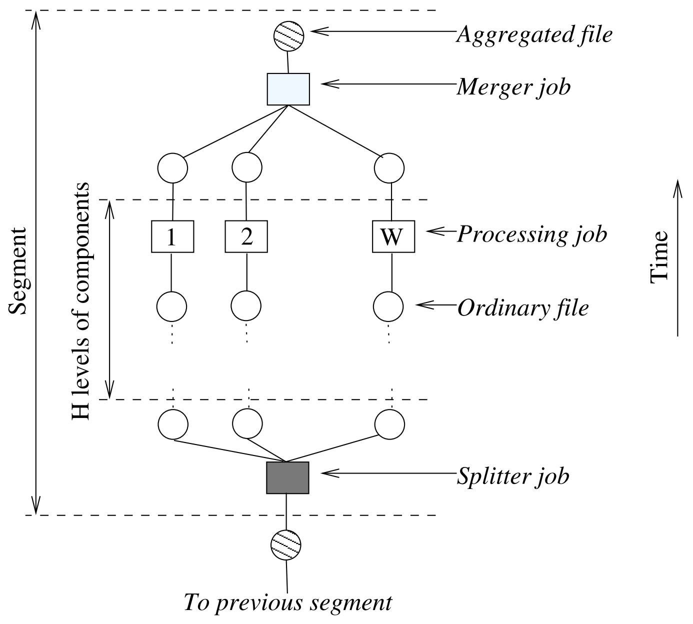


Figure 7.13: Application kernel.

parameter	**value**
Size of intermediate file	10
Size of merged file	100
Execution time of merger	30
Execution time of splitter	30
Execution time of processing job	10

Table 7.1: Parameters of synthetic application.

cation contains $(H \times W + 2) \times S$ jobs and $((H+1) \times W + 1) \times S$ files. The splitting and merging components serve as synchronization and data aggregation points. Table 7.1 shows the application parameters used in our study.

In our experiments, we varied values of C, N, S, H, and W. In all cases, the goal is to achieve availability of the merged file of the top-most segment on the first computational host of the first cluster. Initially, all intermediate files of the first level of the first segment are available in the network. These initial files are distributed

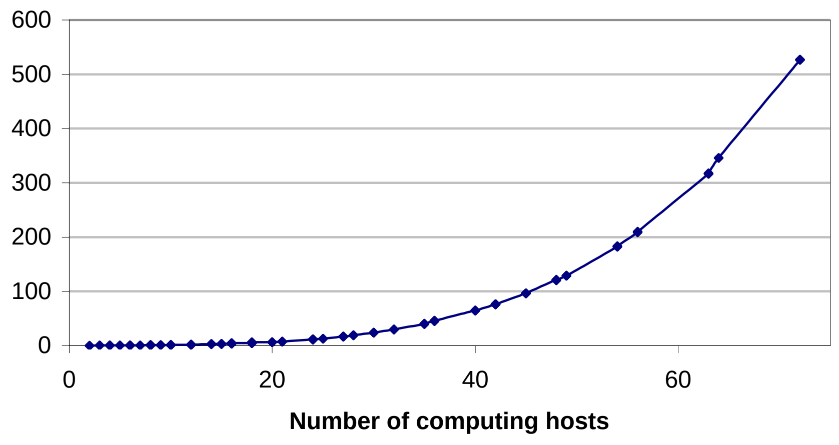


Figure 7.14: Scalability of GPRS wrt the network size.

sequentially to all computing nodes of the network. Appendix D gives an example of a generated problem.

Scalability with the network size

To evaluate scalability of GPRS with respect to the size of the network, we run the planner for a set of networks with parameters $C \in \{1..9\}$, $N \in \{2..9\}$ using the kernel application with $S = 1$, $H = 3$, $W = 5$. Figures 7.14 shows planning time as a function of the number of computing hosts in the network. As can be seen from the figure, the planning time grows fast. This can be explained by the fact that, to support gang reservations of network links, the planner considers all hosts of the network as targets for file transfers. The number of hosts contributes to the branching factor of the search space. The fact that the algorithm still scales to networks of considerable size can be explained by the pruning power of the envelope graph.

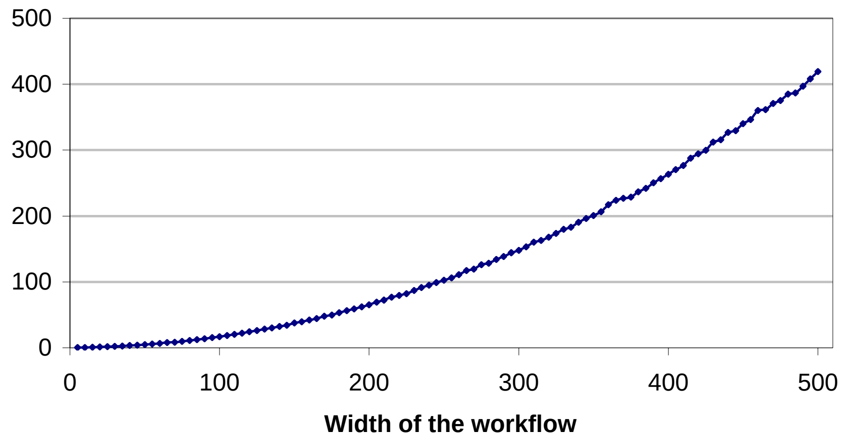


Figure 7.15: Scalability of GPRS wrt the width of the workflow.

Scalability with the width of workflow

Grid workflows are usually characterized by relative small depth of the workflow DAG, but large number of parallel processing sequences. To assess the scalability of GPRS with respect to such applications, we tested our planner on the synthetic problem described above with the value of W in the interval $\{2..500\}$. Other parameters of the model were as follows: $C = 2$, $N = 2$, $S = 1$, $H = 3$. As Figure 7.15 shows, GPRS can scale to large instances of Montage-like applications.

Scalability with the depth of workflow

Finally, we evaluated scalability of GPRS with respect to the depth of the workflow. Figure 7.16 shows the planning time as the function of the number of segments and the number of processing stages within a segment. Parameters of the experiments are $\{C = 2, N = 2, S = 1, W = 5, H = 2..100\}$ and $\{C = 2, N = 2, S = 1..36, W = 5, H = 1\}$.

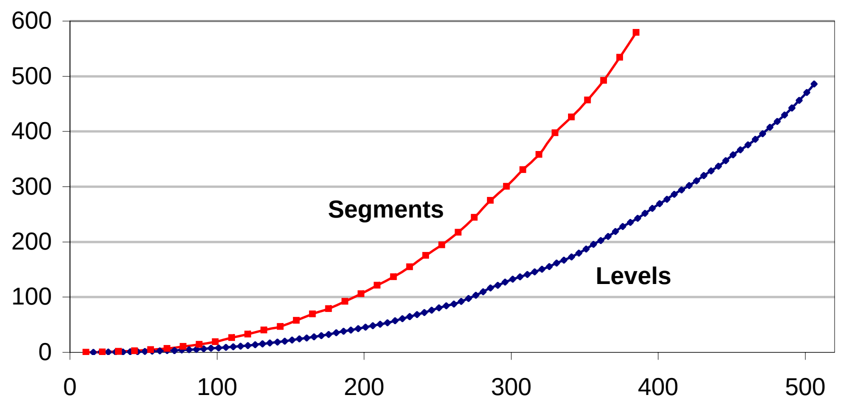


Figure 7.16: Scalability of GPRS wrt the depth of the workflow.

As the results demonstrate, the planning time grows more than quadratically with the depth of the workflow. This can be explained by the fact that in the current implementation a complete constraint propagation is performed after every action choice during the solution extraction phase. The complexity of this propagation is close to linear with respect to the width of the workflow, but grows faster with respect to the depth of the workflow. We expect that a different (lazy) implementation of constraint propagation would lead to significant speedup of the algorithm. Note, however, that grid workflows tend to have few stages, and therefore scalability of the planner with respect to the depth of the workflow is less important than that with the number of parallel execution sequences.

It is noteworthy that, despite the use of the same techniques, the scalability characteristics of GPRS are reverse of those of Sekitei, which scales well with the network size, but assumes few parallel streams in the application DAG. This fact demonstrates that the techniques are not limited to a particular feature of the problem specification, and can be used to design planners for different domains.

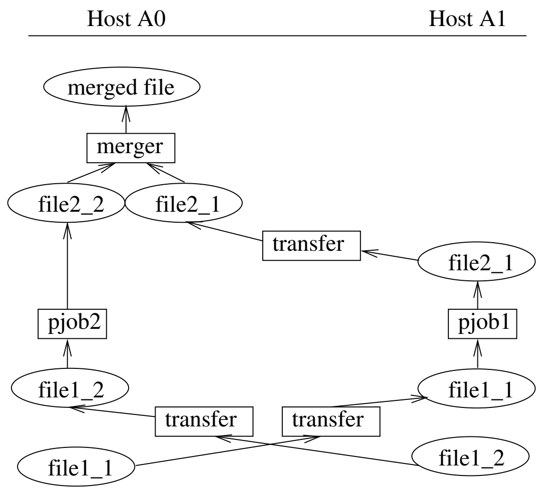


Figure 7.17: Sub-optimal plan generated by GPRS. The planner first commits the critical path (processing of the second file), and then schedules the rest using the remaining resources. This results in two unnecessary file transfer operations.

7.3.3 Solution quality

Due to the use of the critical path scheduler with commitments for plan extraction, the plans found by GPRS are suboptimal. For example, Figure 7.17 illustrates the plan produced by GPRS for the problem with $\{C = 1, N = 2, S = 1, W = 2, H = 1\}$. Appendix D gives the complete specification of this problem instance.

The solution shown in the figure contains two file transfers more than the optimal solution. The cause of such sub-optimality is the use of the critical-path scheduler. In the presented example, the scheduler made a greedy decision about scheduling the longest path of the workflow first, and then had to schedule the rest of the workflow using the remaining resources. In Section 7.4 we discuss possible approaches to address this greediness issue.

Despite possible sub-optimality, GPRS still performs good load balancing. The scheduling problem is NP-hard, so it is problematic to find an exact optimum for problems with reasonable size. However, it is easy to check optimality of the solution in some special cases.

To assess the quality of solutions, we asked the planner to find a configuration of the kernel application described above with one segment with one job level of width 300 for a network with 2 clusters with 4 computing hosts each. We set the link bandwidths to a very high value, so that delays introduced by data transfers are negligible.

This application contains the total of 301 jobs, which can be executed on any of the 8 computing hosts. GPRS assigned 38 jobs to each of the hosts of the B cluster, 39 jobs to three hosts of A cluster (A1,A2, and A3), and 40 jobs to host A0. Since A0 is the host where the final answer was requested, the job assignment is indeed optimal.

The load-balancing effect can be explained by the fact that all decisions made by the scheduler are immediately taken into account by the envelope graph. The envelope is built over all possible execution sequences, and at any moment chooses the best way to achieve every subgoal given the current set of resource reservations.

7.4 Summary

The techniques used in Sekitei to achieve good pruning and search guidance in large-scale problems with numeric interactions between actions are applicable in other domains as well and can yield similar improvements.

In this chapter we have presented GPRS — a planner for grid applications. Grid planning is a variety of the general application configuration problem, whose properties are very different from those of the component placement problem discussed in previous chapters. Grid planning is characterized by relatively small network sizes, but large (wide) application graphs and the need to reason about resource sharing in physical time in addition to network space.

We have shown that, despite the differences in the problem structure, the same techniques that allowed our planner for the CPP achieve good performance without sacrificing expressiveness of the model can be used as a basis for creating a grid planner.

GPRS uses a per-file envelope graph (**T3**) to derive heuristics used by the critical path scheduler. This graph is very similar to the per-proposition logical regression graph of Sekitei 3. Both employ the regression-progression (**T2**) technique, which allows on-the-fly discovery of the relevant portion of the problem specification by accessing external services using a compiler module (**T1**). The numeric values, e.g. the availability times, are propagated forward, which allows the planner to admit complex numeric constraints (**T7**).

In fact, our implementation allows for adding new types of constraints, which may involve multiple variables. As an example, we use a gang reservation constraints to simultaneously reserve multiple links constituting a network path for a data transfer. Using a similar mechanism for early identification of resource conflicts between different branches of the same workflow is an attractive research direction. While creation of a per-set graph similar to the second-level graph of Sekitei 3 may be too expensive due to the large width of the grid workflows, specialized constraints may provide significant pruning and improve both performance of the planner and the quality of the solutions.

Expansion of the envelope graph is naturally bounded (**T3**) by the delays along each of the paths. As the bounds change in the course of constraint propagation after the plan extraction phase commits some reservations, parts of the envelope graph may be further expanded. This allows GPRS to identify good candidate actions without performing exhaustive search.

The suboptimality of solutions found by the current implementation of GPRS is due to the use of a critical path scheduler. A more advanced plan extraction algorithm can significantly improve the quality of the solutions. For example, Cesta et al [18] show that profile-based local search schedulers perform very well when used together the planning graph based planners.

Chapter 8

Conclusions and future work

This thesis has introduced techniques for constructing efficient algorithms for solving the application configuration problem with expressive models. We also identified the kinds of information a description of components and environment needs to provide to enable automated reasoning.

Because of the computational complexity of the problem (see Section 3.3.3), to achieve good performance, it is imperative to take advantage of the natural restrictions of the particular variety of the ACP being solved. In this thesis we presented algorithms for two such varieties. Both algorithms use the techniques enumerated in Section 1.3. In Section 8.1 we summarize our insights on when and how these techniques should be applied to design planners for new domains.

The objective of our work was to provide tools for automated configuration of applications. The models assumed by our algorithms are, necessarily, an approximation of the real behavior of the underlying system. In Section 8.2 we present several extensions that could be incorporated in these models.

Our aim in this thesis was to test the limits of applicability of exact search techniques given the unique challenges of the application configuration problem: large scale open worlds, complex (numeric) dependencies between choices, and the need to optimize the quality of the solution in the presence of non-reversible functions. The work described in this thesis allows one to significantly improve performance and scalability of AI-based planners on the ACP. That said, such improvements alone may be insufficient to solve realistic problems. Other techniques, in addition to exact search, may be useful for construction of production-quality real-world systems, and are discussed in Section 8.3.

8.1 How to design planners for the ACP

Given the undecidability of the general formulation of the application configuration problem, it is unreasonable to expect that a single algorithm will achieve good performance on all varieties of this problem. However, techniques described in this thesis are applicable across a wide range of problems. In this section we summarize the basic principles for design of algorithms for the ACP that we discovered based on our experience with Sekitei and GPRS.

8.1.1 Algorithm

How to hide problem details, such as units of measurement, from the planner? Use the on-demand compilation technique (**T1**). The compiler and decompiler are very simple interfaces, easy to implement for a new framework. They allow to hide semantics of the problem and communication with external services from the planner, thus making the search algorithm domain-independent.

How to deal with large problem specifications and non-reversible functions? Use regression-progression combination (**T2**). The ACP is usually characterized by a small goal specification and large world state. The regression part of the search algorithm ensures that only relevant portions of the search space are explored, while the progression part allows the algorithm to use non-reversible functions for pruning. The progression part also propagates cost estimates, thus efficiently guiding the combined search towards good solutions.

How to get pruning for numeric functions? Use envelope graphs (**T3**). An envelope is a data structure that aggregates information about reachable world state. Envelopes are (relatively) cheap to compute. The main purpose of envelopes is to identify resource conflicts. By doing so, envelopes help prune the search space and may lead to significant, sometimes exponential, speedup.

How to trade off pruning effectiveness with computational requirements? Use several envelope graphs organized in a hierarchy (**T4**). Levels of such a hierarchy communicate with each other by passing (i) relevant operators and data, and (ii) expansion bounds. Using envelopes of different granularity at different levels of the hierarchy makes it possible to simultaneously achieve scalability of the cheapest of the envelopes and the pruning power of the most expensive one.

How to avoid exhaustive search over the network topology? Use weighted distance as the heuristic in the coarsest envelope graph. Although, in theory, it may cause loss of strict optimality, usually it does not, while significantly limiting expansion of the graph.

How to cut loops in And-Or graph expansion? Push infinity (the maximum possible value) through the graph before expanding a leaf node. In case of decreasing cost functions, use the minimum possible value instead. This technique helps to avoid long cycles of propagation and significantly improves performance of the algorithm.

How to encode preferences over feasible plans? Use cost functions depending on the numeric parameters (**T5**). Defining such functions for each operator allows the planner to make local choices, and using a global cost function permits the planner to find a globally optimal configuration.

How to optimize resource consumption if functions are non-reversible? Use resource discretization (**T6**). Although this technique does not guarantee strict optimality, it helps to significantly improve the quality of solutions without incurring large computational overhead.

How to encode dependencies between parts of application, which are not limited to a single object? For complex dependencies, such as gang reservations of resources, use general constraints (**T7**). Requiring these constraints to propagate only in one direction helps to design efficient algorithms for constraint propagation.

How to schedule propagation of updates? In general, cycles in constraint propagation are unavoidable. However, if all constraints affect a single node and can be computed in one direction, using the lower bound on the distance of the sink node from the goal to sort constraints leads to significant performance improvements.

8.1.2 Modeling

How to select objects in an ACP? First, all entities we may need to control need to be represented by objects of the model. This includes components and hosts.

Second, parts of entities may be useful even if we cannot control them directly. For example, in Sekitei, to facilitate reasoning about sharing of network resources between concurrent streams, it is useful to model individual links even though we cannot control them directly.

Finally, data also needs to be represented as objects. We use data items to communicate information about component compatibility. This approach permits great flexibility in specifying application structures, but requires data to be treated as first-class objects.

How to model dependencies so that algorithms are efficient? Make sure all functions are easily computable in one, usually forward, direction. This applies to constraints also.

Second, make sure dependencies affect only one object. This condition helps to organize efficient propagation of dependencies. More general dependencies, e.g. those represented by equations, also can be modeled, but at the expense of degrading performance.

How to discretize resources? Cut close to important constants. Make sure that cuts affect cost bounds, so that the cost-based pruning keeps graphs from explosion.

8.2 Model extensions

In this thesis we investigated design of exact algorithms for planning problems with complex numeric dependencies between operators. Considering the model used in our study in the context of real applications, several extensions can be suggested.

First, we assumed that the semantics of data is captured by the data type. In practice, determining local compatibility of data types may be a hard problem [15]. The two properties of the ACP — large open worlds and non-reversible nature of functions — are also applicable to the ACP with semantic dependencies, e.g. web service composition. Evaluation of our techniques on such applications is a topic for future work.

Second, in this thesis we assume that the data provided by the service descriptions is complete and precise. Network measurements are inherently noisy. Moreover, resource values, such as available link bandwidth, constantly fluctuate. It would be desirable to support such imprecise data in the model. One possible approach worth investigating is to use distribution functions instead of intervals. An issue related to supporting imprecise knowledge is the sensitivity analysis of the plans produced.

In large worlds, such as the Internet, it is infeasible to monitor values of all variables. Therefore, it is desirable that, in addition to the configuration, the planner outputs a set of variables that can affect feasibility and/or optimality of the current plan. Note that some of these variables may not be currently used by the configuration. For example, it may be useful to monitor the status of a currently busy network path if a better configuration can be obtained when that path becomes available.

In many systems, it is not reasonable to assume that complete information is available for a fully automated system. For example, in web-service composition, many dependencies are still represented in human-readable but informal comments. In such situations, a mixed initiative system may be a good solution [61]. The role of the planner in such systems is to check dependencies and make suggestions, leaving the final decision to the human expert.

All examples used to illustrate our algorithms involved constructing an application configuration from scratch. In practice, it is often necessary to fix an existing configuration when it becomes infeasible or no longer satisfies client requirements due to a change in resource availability. By defining operators for component migration (in addition to initial deployment), the algorithms presented in this work can be extended to apply to application configuration repair as well. However, it is not clear how to trade off repair effort and the quality of the resulting configuration in a cost function. Recall that our algorithms require a single function to incorporate all optimization metrics.

8.3 Algorithm improvements

Both algorithms described in this thesis are centralized planners based on the idea of planning graphs.

For large-scale deployments it is desirable to have multiple planners communicating with each other as agents. This way, authority of each planner may be limited to a single administrative domain, thus limiting the need to expose information about the state of the domain. Each planner constructs an optimal solution for some set of clients. Communication between cooperative planners would allow them to achieve a global optimum, thus improving overall performance of the system. For example, a planner for a newly added client can ask its peer to change a configuration created by that planner to enable sharing of parts of the application configurations between several clients. Such sharing may lead to locally suboptimal configurations, but improve overall performance of the system.

The performance of the planner and the quality of produced solutions can be improved even in the centralized case. Earlier (Section 2.2.3) we mentioned that compilation-based planners provide natural support for optimization, but cannot cope with large search spaces. The envelope hierarchy described in this thesis provides efficient pruning of large search spaces. Using compilation into an optimization problem *in addition* to the envelope hierarchy may significantly improve the quality of the solution.

The techniques developed in this thesis target domain-independent planners. Wilkins and desJardins [93] suggest that domain-independent planners may not be the right solution for real world applications. For example, HTN-based planners [21, 27, 74] can incorporate domain knowledge, which allows them to exceed scalability of domain-independent algorithms by orders of magnitude. Using machine-learning techniques together with a planner, domain-dependent or independent, one can further improve performance of the algorithm without sacrificing flexibility of the system [94].

The two presented planners — Sekitei and GPRS — represent two extremes in using the heuristic information provided by the envelope graph(s). Sekitei performs search for a strictly optimal solution with respect to the given model. GPRS, to achieve good performance, never backtracks on committed decisions. A better way to use heuristics may be the limited discrepancy search [91], which combines efficiency (given a high-quality heuristic) with completeness. LDS can also be used as an any-time algorithm, which may be very helpful in the presence of real-time constraints.

Another alternative for organizing the plan extraction phase is local search [40, 39, 80]. Local search is in general suboptimal, but empirically results in high-quality solutions and is quite efficient.

8.4 Conclusion

In this thesis we demonstrated that AI planning can be used to design algorithms for the application configuration problem that achieve good performance without sacrificing expressiveness of the model. We hope that the presented work will contribute to development of adaptive systems and models for describing component-based distributed applications.

It is interesting to notice that several other planning domains share the features of the application configuration problem. In particular, in the aerospace and logistics domains the choice of actions is often driven by the numerical rather than logical part. For example, it is not (logically) necessary to refuel unless you are running (numerically) out of gas. The techniques developed in this thesis are applicable in such domains as well.

Appendix A

Performance of metric planners on the CPP

A.1 Performance evaluation for existing planners

We evaluated performance of the following three planners on two examples of the application configuration problem: Sapa 1.02 (April 2004), Metric-FF (2002), LPG-td (June 2004). All reported measurements were taken on a 1.2GHz Pentium III with 512M of RAM. Sapa is implemented in Java, Metric-FF and LPG-td in C.

Sapa and LPG were among the top performers at the two recent International Planning Competitions [50] (2002 and 2004). Sapa [24] performs forward search using a heuristic derived from solving a relaxed problem. The relaxation involves ignoring numeric effects. However, some numeric information is later incorporated to update the heuristic estimates. LPG [40, 39] performs local search with restarts.

Metric-FF [49] is a metric variant of FF [48], one of the fastest planners currently available. Metric-FF admits only linear functions, and uses numeric parts of action specifications to derive heuristics.

None of these planners is guaranteed to find optimal plans.

A.1.1 Webcast application

The CPP does not have the time aspect, so it is compiled into a planning problem with unit action duration (see Section 3.2.3). The PDDL files for the domain and problem specifications for the webcast application are provided in Section A.2.

Metric-FF failed to solve the webcast problem. Two operators, `placeZp` and `placeUn` completely undo effects of each other, both logical and numeric. This causes a loop in one of the internal data structures of FF, and the planner exits with an error message. In the rest of this section we describe performance of Sapa and LPG-td on the webcast problem.

Both Sapa and LPG easily solved the problem instance with a direct connection between two hosts. However, resource-constrained instances of the problem, in which additional components are required to satisfy the client's requirements, were challenging for both planners.

In the domain file shown in Section A.2.1 we used four link crossing operators with disjoint preconditions to avoid using conditional effects. In all our examples, among these four operators only `cross2` is used in the plan. The presence of the other three `cross` operators negatively affects performance of the planners, because these operators create logical loops in possible configurations, which can be pruned only based on numeric information.

We ran both planners with three variations of the domain file. The first one, called `oneCross` includes only one link crossing operator `cross2`. The `twoCross` domain includes `cross1` and `cross2` operators. Finally, `allCross` uses all four link crossing operators.

Further, to make the task easier, we assign CPU resources to hosts so that to limit the total number of possible configurations. The optimal plan requires 27 units of CPU on the server host, 36 units on the client host, and does not use resources of any intermediate hosts. In our problem files, the available CPU values are set to 30, 40, and 0 respectively.

Sapa easily solved the 2-host problem with `oneCross` (less than 1 second search time). On `twoCross` and `allCross` domains the search took 8 and 9 seconds respectively. The answers found for these domains were suboptimal.

For the 6-host problem, Sapa found a solution for the `oneCross` domain in 10 minutes. On the `twoCross` domain, no solution was found in 20 minutes.

The reason for this performance is that Sapa's heuristics are based on logical reachability. For the webcast problem these heuristics fail, so Sapa falls back to exhaustive search and considers all possible application configurations that can be designed given available network resources. Given the tight CPU amounts in our problem specification, this approach can find solutions for small instances. However, if we increase the value of available host CPU, even the 2-host problem becomes unsolvable for Sapa.

On the 2-host problem, LPG-td (*speed* configuration) found an optimal solution for the `oneCross` and `twoCross` domains in under a second. For the `allCross` domain the search took 1.35 seconds.

On the 6-host network, an optimum plan for the `oneCross` domain was found in

217 seconds. For the `twoCross` domain, LPG did not leave the *Computing mutex...* stage for more than 20 minutes and was terminated. The source code for LPG is not yet available, so it is hard to determine the reason for such behavior.

A.1.2 Grid application

We also tested the planners on a simple grid application that involved a sequence of components combining together groups of four files. In total, the problem involved 9 files, 2 component instances, and 2 network hosts. The bandwidth of the link connecting the hosts allows multiple simultaneous file transfers.

This ACP can be compiled into a planning problem with durative actions. Metric-FF does not support this model. Sapa found a correct plan in 17 minutes. However it scheduled all data transfers sequentially, so that the final plan had suboptimal duration. LPG reported an internal error in reachability analysis.

A.2 PDDL specification of the ACP for webcast application

A.2.1 Webcast domain file

```
(define (domain Webcast)
  (:requirements :strips :typing :fluents)
  (:types component node interface)
  (:constants M T I Z - interface Cl Sp Mr Zp Un - component)
  (:predicates (av ?i - interface ?n - node)
               (link ?n1 ?n2 - node)
               (placed ?c - component ?n - node))
  (:functions  (lbw ?n1 ?n2 - node)
               (cpu ?n - node)
               (ibw ?i - interface ?n - node)
               (cost))
  (:action placeCl
   :parameters (?n - node)
   :precondition (and (av M ?n) (>= (ibw M ?n) 91) (>= (cpu ?n) 9))
```

```
 :effect (and (placed Cl ?n)
              ;; The following line is needed for SAPA to avoid
              ;; LP_Utility.getResProfile: rhs is
              ;; NOT subsumed by lhs.
              (increase (ibw M ?n) 0)
              (decrease (cpu ?n) 9) (increase (cost) 1))
)
(:action placeSp
 :parameters (?n - node)
 :precondition (and (av M ?n) (>= (cpu ?n) (/ (ibw M ?n) 5)))
 :effect (and (placed Sp ?n)
              (av T ?n) (av I ?n)
              (assign (ibw T ?n) (/ (* (ibw M ?n) 7) 10))
              (assign (ibw I ?n) (/ (* (ibw M ?n) 3) 10))
              (increase (ibw M ?n) 0)
              (decrease (cpu ?n) (/ (ibw M ?n) 5))
              (increase (cost) (+ 1 (/ (ibw M ?n) 10))))
)
(:action placeMr
 :parameters (?n - node)
 :precondition (and (av T ?n) (av I ?n)
                    (>= (cpu ?n) (/ (+ (ibw T ?n) (ibw I ?n)) 5)))
 :effect (and (placed Mr ?n) (av M ?n)
              (assign (ibw M ?n) (+ (ibw T ?n) (ibw I ?n)))
              (increase (ibw T ?n) 0) (increase (ibw I ?n) 0)
              (decrease (cpu ?n) (/ (+ (ibw T ?n) (ibw I ?n)) 5))
              (increase (cost)(+ 1(/(+ (ibw T ?n)(ibw I ?n))10))))
)
(:action placeZp
 :parameters (?n - node)
 :precondition (and (av T ?n) (>= (cpu ?n) (/ (ibw T ?n) 10)))
 :effect (and (placed Zp ?n) (av Z ?n)
              (assign (ibw Z ?n) (/ (ibw T ?n) 2))
              (increase (ibw T ?n) 0)
              (decrease (cpu ?n) (/ (ibw T ?n) 10))
              (increase (cost) (+ 1 (/ (ibw T ?n) 10))))
)
(:action placeUn
 :parameters (?n - node)
 :precondition (and (av Z ?n) (>= (cpu ?n) (/ (ibw Z ?n) 5)))
 :effect (and (placed Un ?n) (av T ?n)
              (assign (ibw T ?n) (* (ibw Z ?n) 2))
              (increase (ibw Z ?n) 0)
              (decrease (cpu ?n) (/ (ibw Z ?n) 5))
              (increase (cost) (+ 1 (/ (ibw Z ?n) 5))))
)
```

```
  (:action cross1
   :parameters (?i - interface ?from ?to - node)
   :precondition (and (av ?i ?from) (link ?from ?to)
                      (> (ibw ?i ?from) (lbw ?from ?to))
                      (> (lbw ?from ?to) 0))
   :effect (and (av ?i ?to)
                (assign (ibw ?i ?to) (lbw ?from ?to))
                (increase (ibw ?i ?from) 0)
                (assign (lbw ?from ?to) 0)
                (increase (cost) (+ 1 (/ (ibw ?i ?from) 10))))
  )
  (:action cross2
   :parameters (?i - interface ?from ?to - node)
   :precondition (and (av ?i ?from) (link ?from ?to)
                      (<= (ibw ?i ?from) (lbw ?from ?to)))
   :effect (and (av ?i ?to)
                (assign (ibw ?i ?to) (ibw ?i ?from))
                (increase (ibw ?i ?from) 0)
                (decrease (lbw ?from ?to) (ibw ?i ?from))
                (increase (cost) (+ 1 (/ (ibw ?i ?from) 10))))
  )
  (:action crossBack1
   :parameters (?i - interface ?from ?to - node)
   :precondition (and (av ?i ?from) (link ?to ?from)
                      (> (ibw ?i ?from) (lbw ?to ?from))
                      (> (lbw ?from ?to) 0))
   :effect (and (av ?i ?to)
                (assign (ibw ?i ?to) (lbw ?to ?from))
                (increase (ibw ?i ?from) 0)
                (decrease (lbw ?to ?from) (lbw ?to ?from))
                (increase (cost) (+ 1 (/ (ibw ?i ?from) 10))))
  )
  (:action crossBack2
   :parameters (?i - interface ?from ?to - node)
   :precondition (and (av ?i ?from) (link ?to ?from)
                      (<= (ibw ?i ?from) (lbw ?to ?from)))
   :effect (and (av ?i ?to)
                (assign (ibw ?i ?to) (ibw ?i ?from))
                (increase (ibw ?i ?from) 0)
                (decrease (lbw ?to ?from) (ibw ?i ?from))
                (increase (cost) (+ 1 (/ (ibw ?i ?from) 10))))
  )
)
```

A.2.2 Webcast problem file

```
;; Achieving the goal requires splitting the stream and
;; compressing the text portion
(define (problem Webcast6n)
  (:domain Webcast)
  (:objects n0 n1 n2 n3 n4 n5 - node)
  (:init
    ;; network topology
    (link n0 n1) (link n1 n2) (link n2 n3) (link n3 n4)
    (link n3 n5) (link n4 n5)

    ;; network resources
    (= (lbw n0 n1) 100)
    (= (lbw n1 n2) 100) (= (lbw n2 n3) 70)
    (= (lbw n3 n4) 100) (= (lbw n3 n5) 100) (= (lbw n4 n5) 100)
    (= (cpu n0) 30) (= (cpu n1) 0) (= (cpu n2) 0) (= (cpu n3) 0)
    (= (cpu n4) 0) (= (cpu n5) 40)

    ;; Initially M is available on n0
    (av M n0) (= (ibw M n0) 100)

    ;; The cost is incremented by every action
    (= (cost) 0)

    ;; To complete specification of the initial state
    (= (ibw M n1) 0) (= (ibw M n2) 0)
    (= (ibw M n3) 0) (= (ibw M n4) 0) (= (ibw M n5) 0)
    (= (ibw T n0) 0) (= (ibw T n1) 0) (= (ibw T n2) 0)
    (= (ibw T n3) 0) (= (ibw T n4) 0) (= (ibw T n5) 0)
    (= (ibw Z n0) 0) (= (ibw Z n1) 0) (= (ibw Z n2) 0)
    (= (ibw Z n3) 0) (= (ibw Z n4) 0) (= (ibw Z n5) 0)
    (= (ibw I n0) 0) (= (ibw I n1) 0) (= (ibw I n2) 0)
    (= (ibw I n3) 0) (= (ibw I n4) 0) (= (ibw I n5) 0)
  )
  (:goal (and (placed Cl n5)) )
  (:metric minimize (cost))
)
```

A.3 PDDL specification of the ACP for a grid application

A.3.1 Grid domain file

```
(define (domain GridsTree4)
(:requirements :strips :typing :fluents)
(:types component host file)
(:predicates (av ?f - file ?h - host)
             (link ?n1 ?n2 - host)
             (childA ?p ?c - file) (childB ?p ?c - file)
             (childC ?p ?c - file) (childD ?p ?c - file))
(:functions  (lbw ?n1 ?n2 - host)
             (size ?file - file)
             (hasCpu ?host - host)
             (cost))
(:durative-action Recompute
 :parameters (?host - host ?product ?ca ?cb ?cc ?cd - file)
 :duration (= ?duration (/ (size ?product) 2))
 :condition (and
         ;; Dependencies
         (over all (childA ?product ?ca))
         (over all (childB ?product ?cb))
         (over all (childC ?product ?cc))
         (over all (childD ?product ?cd))

         ;; Have all the files
         (at start (av ?ca ?host)) (at start (av ?cb ?host))
         (at start (av ?cc ?host)) (at start (av ?cd ?host))

         ;; Have enough CPU. Everything is piecewise constant, so
         ;; checking in the beginning should be enough
         (at start (>= (hasCpu ?host) (/ (size ?product) 5))))
 :effect (and
         (at end (av ?product ?host))
         (at start (decrease (hasCpu ?host)
                   (/ (size ?product) 5)))
         (at end   (increase (hasCpu ?host)
                   (/ (size ?product) 5)))
         (at end (increase (cost) (/ (size ?product) 5) )))
)

;; Links and bandwidth are specified with
;;(smaller host id) -> (bigger host id)
;; To allow transfers in both ways, we have 2 actions
(:durative-action transferForward
 :parameters (?file - file ?from ?to - host)
   ;; Duration should depend on the available bandwidth, but to
   ;; keep everything linear we will divide size by a constant
 :duration (= ?duration (/ (size ?file) 10) )
 :condition (and (at start (av ?file ?from))
                 (over all (link ?from ?to))
```

```
                   ;; same constant as in duration denominator
                   (at start (> (lbw ?from ?to) 10)))
   :effect (and (at end (av ?file ?to))
                ;; same constant as in duration denominator
                (at start (decrease (lbw ?from ?to) 10) )
                (at end   (increase (lbw ?from ?to) 10) )
                (at end (increase (cost) (/ (size ?file) 10) )))
  )
  (:durative-action transferBackward
   :parameters (?file - file ?from ?to - host)
   ;; Duration should depend on the available bandwidth, but to
   ;; keep everything linear we will divide size by a constant
   :duration (= ?duration (/ (size ?file) 10) )
   :condition (and (at start (av ?file ?from))
                   (over all (link ?to ?from))
                   ;; same constant as in duration denominator
                   (at start (> (lbw ?to ?from) 10)))
   :effect (and (at end (av ?file ?to))
                ;; same constant as in duration denominator
                (at start (decrease (lbw ?to ?from) 10) )
                (at end   (increase (lbw ?to ?from) 10) )
                (at end (increase (cost) (/ (size ?file) 20) )))
  )
)
```

A.3.2 Grid problem file

```
(define (problem GT4_Problem2)
  (:domain GridsTree4)
  (:objects host1 host2  - host
            Total A B C D Da Db Dc Dd - file)
  (:init
   (at 0.1 (av A host1))  (at 0.1 (av B host1))
   (at 0.1 (av C host1))
   (at 0.1 (av Da host1)) (at 0.1 (av Db host1))
   (at 0.1 (av Dc host1)) (at 0.1 (av Dd host1))

   (= (size A) 40) (= (size B) 40) (= (size C) 40) (= (size D) 40)
   (= (size Da) 10) (= (size Db) 10)
   (= (size Dc) 10) (= (size Dd) 10)
   (= (size Total) 160)

   (childA Total A) (childB Total B)
   (childC Total C) (childD Total D)
   (childA D Da) (childB D Db) (childC D Dc) (childD D Dd)
```

```
    (link host1 host2) (= (lbw host1 host2) 100)
    (= (hasCPU host1) 0) (= (hasCPU host2) 100)

    (= (cost) 0)
   )
   (:goal (av Total host1)))
)
```

Appendix B

Compiler interface for the Sekitei planner

This appendix contains the Compiler interface used by the Sekitei 3 planner (Chapter 6). The purpose of the compiler module is to hide communication with external services and semantics of the framework actions from the planning algorithm. All communication between the planner and the compiler is carried using ground variables and operators.

```
/**
 * The interface for on-demand compilation of framework specific
 * CPP into Sekitei. To be implemented by a framework
 * @author T.Kichkaylo
 */
public interface Compiler {
  /**
   * Framework specific initializer.
   * @param frameworkProblem a framework specific pointer
   * to the problem. For example, a String name of a directory.
   * @throws Exception various parsing and IO exceptions
   */
  void init( Object frameworkProblem ) throws Exception;
```

```
/**
 * Produces a ground String variable name.
 * Used in @link{sekitei.formula.VariableExpression}.
 * @param thing some framework specific thing, for example,
 * a String or a parameterized variable name
 * @return ground variable name
 */
String makeName( Object thing );
/**
 * Gets ground logical goal state
 * @return set of LevelVar for ground propositions that
 *  should be true in the goal state
 */
HashSet getGoalState();
/**
 * @param Name of a ground proposition
 * @return LevelVar for the initial level of the proposition
 */
LevelVar getInitialLevel( String prop );
/**
 * Gets a set of all LeveledOperators that achieve
 * the given proposition
 * @param prop a ground proposition that needs to be achieved
 * @return ArrayList of ground @link{LevedOperator}s
 * that achieve the given proposition
 */
ArrayList opsAchieveProp( LevelVar prop );
/**
 * Gets the value of a resource variable in the initial state
 * @param thing a ground resource variable
 *  (e.g. VariableExpression)
 * @return the value of this variable or <code>null</code>
 */
Constant getInitialValue( Object thing );
/**
 * Gets the optimistic value of a resource variable
 * @param thing a ground resource variable
 *  (e.g. VariableExpression)
 * @return the value of this variable
 */
Constant getOptimisticValue( Object thing );
/**
 * Compiler-specific estimate of the minimum cost of
 * achieving a proposition.
 * This is used as an admissible estimate in {@link PropLRG}.
 * For example, this method can be used to "seed the path"
```

```
    * in {@link PSFCompiler}.
    * @param prop the proposition
    * @return lower bound on cost, 0 being the safe bet
    */
   int getLowerBoundOnCost( LevelVar prop );
   /**
    * Determine degradability/upgradability of a variable using an
    * operator instance as a helper.
    * @param var some representation of the variable
    * (e.g. VariableExpression)
    * @param lop LeveledOperator being considered, as a helper
    * @return -1/0/1/2 (degradable/exact/upgradable/neither)
    */
   int getXgradable( Object var, LeveledOperator lop );
}
```

Appendix C

Component descriptions for PSF webcast application

This appendix contains specifications of the components of the webcast application in the format native to the Partitionable Services Framework. The component descriptions specify execution preconditions and effects. This specification of the webcast application was used for evaluation of the Sekitei 2 algorithm (Section 5.4).

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes" ?>
<ListComponents>
<Component>
  <Name>Client</Name>
  <Linkages>
    <Requires>
      <Interface><InterfaceName>Media</InterfaceName></Interface>
    </Requires>
  </Linkages>
  <Conditions>
    <Expression>( Node.CPU > 9 )</Expression>
    <Expression>( Media.BW > 80 )</Expression>
    <Expression>( Media.LowRes = True )</Expression>
  </Conditions>
  <Effects>
```

```
    <FloatAssignment>Node.CPU:=( Node.CPU - 9 )</FloatAssignment>
  </Effects>
</Component>

<Component>
  <Name>Merger</Name>
  <Linkages>
    <Implements>
      <Interface><InterfaceName>Media</InterfaceName></Interface>
    </Implements>
    <Requires>
      <Interface><InterfaceName>Text</InterfaceName></Interface>
      <Interface><InterfaceName>Image</InterfaceName></Interface>
    </Requires>
  </Linkages>
  <Conditions>
    <Expression>( Node.CPU > 7 )</Expression>
    <Expression>( Image.LowRes = True )</Expression>
  </Conditions>
  <Effects>
    <FloatAssignment>Node.CPU:=( Node.CPU - 7 )</FloatAssignment>
    <FloatAssignment>Media.BW:=( Text.BW + Image.BW )</FloatAssignment>
    <FloatAssignment>Media.LowRes:=True</FloatAssignment>
  </Effects>
</Component>

<Component>
  <Name>Unzip</Name>
  <Linkages>
    <Implements>
     <Interface><InterfaceName>Text</InterfaceName></Interface>
    </Implements>
    <Requires>
      <Interface><InterfaceName>Zip</InterfaceName></Interface>
    </Requires>
  </Linkages>
  <Conditions>
    <Expression>( Node.CPU > 2 )</Expression>
  </Conditions>
  <Effects>
    <FloatAssignment>Node.CPU:=( Node.CPU - 2 )</FloatAssignment>
    <FloatAssignment>Text.BW:=( Zip.BW * 2 )</FloatAssignment>
  </Effects>
</Component>

<Component>
```

```
  <Name>Zip</Name>
  <Linkages>
    <Implements>
     <Interface><InterfaceName>Zip</InterfaceName></Interface>
    </Implements>
    <Requires>
      <Interface><InterfaceName>Text</InterfaceName></Interface>
    </Requires>
  </Linkages>
  <Conditions>
    <Expression>( Node.CPU > 4 )</Expression>
  </Conditions>
  <Effects>
    <FloatAssignment>Node.CPU := ( Node.CPU - 4 )</FloatAssignment>
    <FloatAssignment>Zip.BW := ( Text.BW * 0.5 )</FloatAssignment>
  </Effects>
</Component>

<Component>
  <Name>Splitter</Name>
  <Linkages>
    <Implements>
     <Interface><InterfaceName>Text</InterfaceName></Interface>
     <Interface><InterfaceName>Image</InterfaceName></Interface>
    </Implements>
    <Requires>
      <Interface><InterfaceName>Media</InterfaceName></Interface>
    </Requires>
  </Linkages>
  <Conditions>
    <Expression>( Node.CPU > 7 )</Expression>
  </Conditions>
  <Effects>
    <FloatAssignment>Node.CPU := ( Node.CPU - 7 )</FloatAssignment>
    <FloatAssignment>Text.BW := ( Media.BW * 0.8 )</FloatAssignment>
    <FloatAssignment>Image.BW := ( Media.BW * 0.2 )</FloatAssignment>
    <FloatAssignment>Image.LowRes := ( Media.LowRes )</FloatAssignment>
  </Effects>
</Component>

<Component>
  <Name>Cleaner</Name>
  <Linkages>
    <Implements>
     <Interface><InterfaceName>Image</InterfaceName></Interface>
    </Implements>
    <Requires>
      <Interface><InterfaceName>Image</InterfaceName></Interface>
    </Requires>
  </Linkages>
  <Conditions><Expression>( Node.CPU > 1 )</Expression></Conditions>
  <Effects>
    <FloatAssignment>Node.CPU := ( Node.CPU - 1 )</FloatAssignment>
    <FloatAssignment>Image.LowRes := True</FloatAssignment>
  </Effects>
</Component>

</ListComponents>
```

Appendix D

Grid application test

This appendix provides the complete specification of the synthetic application used for evaluation of the GPRS algorithm (Chapter 7). The instance shown has the following parameters: $C = 1$, $N = 2$, $S = 1$, $W = 2$, $H = 1$.

The *Application* section contains the names of Java classes implementing problem-specific component and file types. The content of the XML tags describing components and file instances (*Component* and *GroundFile*), containing problem-specific initialization information, is passed to the constructors of the dynamically loaded classes.

The problem-specific component and file types are required to implement a simple interface, and can be considered a part of the on-demand compiler (**T1**) for GPRS.

The *Network* section describes properties of the nodes and links using piecewise-constant functions (slots). The *Data* section specifies the initial availability of files on hosts; and the *Goal* section contains the file-host pair representing the goal of the workflow to be constructed.

```
<GridProblem>

  <Application>
    <Component id="merger"
       classname="rootpackage.grids.kernel.KernelMerger"
       time="40" width="2" height="1" />
    <Component id="splitter"
       classname="rootpackage.grids.kernel.KernelSplitter"
       time="30" width="2" />
    <Component id="processor"
       classname="rootpackage.grids.kernel.KernelProcessor"
       time="10" height="1" />
    <GroundFile id="middle"
       classname="rootpackage.grids.kernel.KernelFileMiddle"
       size="10"/>
    <GroundFile id="merged"
       classname="rootpackage.grids.kernel.KernelFileMerged"
       size="100"/>
  </Application>

  <Network>
    <Node id="Center">
      <Slot start="0" end="inf" value="0"/>
    </Node>
    <Node id="rtA">
      <Slot start="0" end="inf" value="0"/>
    </Node>
    <Node id="A0">
      <Slot start="0" end="inf" value="1"/>
    </Node>
    <Node id="A1">
      <Slot start="0" end="inf" value="1"/>
    </Node>

    <Link id="lrA" from="rtA" to="Center">
      <Slot start="0" end="inf" value="30"/>
    </Link>
    <Link id="lrA0" from="rtA" to="A0">
      <Slot start="0" end="inf" value="10"/>
    </Link>
    <Link id="lA0-1" from="A0" to="A1">
      <Slot start="0" end="inf" value="10"/>
    </Link>
    <Link id="lrA1" from="rtA" to="A1">
      <Slot start="0" end="inf" value="10"/>
    </Link>
  </Network>

  <Data>
    <DataItem>
      <GroundFile type="middle" segment="1" level="1" offset="1"/>
      <Node ref="A0" />
    </DataItem>
    <DataItem>
      <GroundFile type="middle" segment="1" level="1" offset="2"/>
      <Node ref="A1" />
    </DataItem>
  </Data>

  <Goal>
     <GroundFile type="merged" segment="1"/>
     <Node ref="A0" />
  </Goal>
</GridProblem>
```

Bibliography

[1] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Secure, efficient data transport and replica management for high-performance data-intensive computing. In *Proc. of IEEE Mass Storage Conference*, 2001.

[2] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web services agreement specification (WS-Agreement) version 1.1. http://www.gridforum.org/Meetings/GGF11/Documents/draft-ggf-graap-agreement.pdf, 2004.

[3] A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. DAML-S: Semantic markup for web services. In *Proc. of International Semantic Web Working Symposium (SWWS)*, 2001.

[4] F. Bacchus and Y. W. Teh. Making forward chaining relevant. In *Proc. of International Conference on AI Planning and Scheduling (AIPS)*, 1998.

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

[6] G. B. Berriman, J. C. Good, A. C. Laity, A. Bergou, J. Jacob, D. S. Katz, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, and R. Williams. Montage: A grid enabled image mosaic service for the national virtual observatory. In *Proc. of Astronomical Data Analysis Software and Systems (ADASS) Conference*, 2003.

[7] S. Binato, W. Hery, D. Loewenstern, and M. Resende. A GRASP for job shop scheduling. In P. Hansen and C.C. Ribeiro, editors, *Essays and surveys on metaheuristics*. Kluwer Academic Publishers, 2001.

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

[9] J. Blythe, E. Deelman, Y. Gil, C. Kesselman, A. Agarwal, G. Mehta, and K. Vahi. The role of planning in grid computing. In *Proc. of International Conference on Automated Planning and Scheduling (ICAPS)*, 2003.

[10] J. Blythe, Y. Gil, and E. Deelman. Coordinating workflows in shared grid environments. In *Proc. of ICAPS'04 Workshop on Planning and Scheduling for Web and Grid Services*, 2004.

[11] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proc. of European Conference on Planning (ECP)*, 1999.

[12] F. Bustamante and K. Schwan. Active Streams: An approach to adaptive dis-

tributed systems. In *Proc. of Workshop on Hot Topics in Operating Systems (HotOS)*, 2001.

[13] T. Bylander. Complexity results for planning. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI)*, 1991.

[14] T. Bylander. An average case analysis of planning. In *Proc. of National Conference on Artificial Intelligence (AAAI)*, 1993.

[15] D. Calvanese, G. De Giacomo, M. Lenzerini, and D. Nardi. Reasoning in expressive description logics. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1581–1634. Elsevier Science Publishers, 2001.

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

[17] A. Cesta, A. Oddi, and S. Smith. A constraint-based method for project scheduling with time windows. *Journal of Heuristics*, 8(1), 2002.

[18] A. Cesta, F. Pecora, and R. Rasconi. Biasing the structure of scheduling problems through classical planners. In *Proc. of ICAPS Workshop on Integrating Planning into Scheduling (WIPIS)*, 2004.

[19] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, C. Kesselman, P. Kunszt, M. Ripeanu, K. Stockinger, and B. Tierney. Giggle: A framework for constructing scalable replica location services. In *Proc. of IEEE Supercomputing*, 2002.

[20] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 2001.

[21] K. Currie and A. Tate. O-Plan: the open planning architecture. *Artificial Intelligence*, 52:49–86, 1991.

[22] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz. An architecture for a secure service discovery service. In *Proc. of International Conference on Mobile Computing and Networking (MobiCom)*, 1999.

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

[24] M. Do and S. Kambhampati. Sapa: A scalable multi-objective metric temporal planner. *Journal of Artificial Intelligence Research*, 2003.

[25] M. B. Do and S. Kambhampati. Solving planning-graph by compiling it into CSP. In *Proc. of International Conference on AI Planning and Scheduling (AIPS)*, 2000.

[26] S. Edelkamp and J. Hoffmann. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical report, University of Freiburg, 2003.

[27] K. Erol, J. Hendler, and D. Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proc. of International Conference on AI Planning and Scheduling (AIPS)*, pages 249–254, 1994.

[28] K. Erol, D. Nau, and V. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning: A detailed analysis. Technical Report CS-TR-2797, University of Maryland, 1991.

[29] O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson. An approach to planning with incomplete information. In *Proc. International Conference on the Principles of Knowledge Representation and Reasoning (KR)*, 1992.

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

[31] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and coallocation. In *Proc. of International Workshop on Quality of Service (IWQoS)*, 1999.

[32] 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.

[33] I. Foster, A. Roy, and V. Sander. A quality of service architecture that combines resource reservation and application adaptation. In *Proc. of International Workshop on Quality of Service (IWQoS)*, 2000.

[34] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system

for representing, querying, and automating data derivation. In *Proc. of Conference on Scientific and Statistical Database Management (SSDBM)*, 2002.

[35] M. Fox and D. Long. PDDL2.1: An extension to PDDL for modelling time and metric resources. Technical report, University of Durham, 2001.

[36] J. Frank and E. Kurklu. SOFIA's choice: Scheduling observations for an airborne observatory. In *Proc. of International Conference on Automated Planning and Scheduling (ICAPS)*, 2003.

[37] X. Fu and V. Karamcheti. Planning for network-aware paths. In *Proc. of IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, 2003.

[38] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS: Composable, Adaptive Network Services infrastructure. In *Proc. of USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2001.

[39] A. Gerevini, A. Saetti, and I. Serina. Planning in PDDL2.2 domains with LPG-TD. In *International Planning Competition (in conjunction with ICAPS-04), abstract booklet of the competing planners*, 2004.

[40] A. Gerevini and I. Serina. LPG: a planner based on local search for planning graphs. In *Proc. of International Conference on AI Planning and Scheduling (AIPS)*, 2002.

[41] M. Ghallab and H. Laruelle. Representation and control in IxTeT, a temporal planner. In *Proc. of International Conference on AI Planning and Scheduling (AIPS)*, 1994.

[42] K. Golden. Leap before you look: Information gathering in the PUCCINI planner. In *Proc. of International Conference on AI Planning and Scheduling (AIPS)*, 1998.

[43] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft. SmartFrog: Configuration and automatic ignition of distributed applications. Technical report, HP, 2003.

[44] S. Gribble, M. Welsh, R. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. Katz, Z. M. Mao, S. Ross, and B. Zhao. The Ninja architecture for robust Internet-scale systems and services. *Computer Networks*, 35(4):473–497, 2001.

[45] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2), 1968.

[46] P. Haslum and H. Geffner. Heuristic planning with time and resources. In *Proc. of IJCAI-01 Workshop on Planning with Resources*, 2001.

[47] M. Helmert. Decidability and undecidability results for planning with numerical state variables. In *Proc. of International Conference on AI Planning and Scheduling (AIPS)*, 2002.

[48] J. Hoffmann. A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. In *Proc. of the 12th International Symposium on Methodologies for Intelligent Systems (ISMIS-00)*, 2000.

[49] J. Hoffmann. Extending FF to numerical state variables. In *Proc. of European Conference on Artificial Intelligence (ECAI)*, 2002.

[50] International planning competition. http://ls5-www.cs.uni-dortmund.de/

[51] A.-A. Ivan, J. Harman, M. Allen, and V. Karamcheti. Partitionable services: A framework for seamlessly adapting distributed applications to heterogenous environments. In *Proc. of the IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, 2002.

[52] The Jini architecture specification. http://www.sun.com/jini/specs/jini1_2.pdf.

[53] A. Jónsson and J. Frank. A framework for dynamic constraint reasoning using procedural constraints. In *Proc. of European Conference on Artificial Intelligence (ECAI)*, 2000.

[54] A. Jónsson, P. Morris, N. Muscettola, and K. Rajan. Planning in interplanetary space: Theory and practice. In *Proc. of International Conference on AI Planning and Scheduling (AIPS)*, 2000.

[55] H. Kautz and B. Selman. Planning as satisfiability. In *Proc. of European Conference on Artificial Intelligence (ECAI)*, 1992.

[56] H. Kautz and B. Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Proc. of AIPS-98 Workshop Planning as Combinatorial Search*, 1998.

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

[58] T. Kichkaylo. Timeless planning and the component placement problem. In *Proc. of ICAPS'04 Workshop on Planning and Scheduling for Web and Grid Services*, 2004.

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

[60] T. Kichkaylo, A. Ivan, and V. Karamcheti. Constrained component deployment in wide-area networks using AI planning techniques. In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[61] J. Kim and J. Blythe. Supporting plan authoring and analysis. In *Proc. of International Conference on Intelligent User Interfaces (IUI)*, 2003.

[62] J. Koehler. Planning under resource constraints. In *Proc. of European Conference on Artificial Intelligence (ECAI)*, 1998.

[63] J. Koehler and J. Hoffmann. Handling of inertia in a planning system. Technical Report 122, Albert-Ludwigs-University, 1999.

[64] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In *Proc. of European Conference on Planning (ECP)*, 1997.

[65] R. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 1985.

[66] R. Korf. Divide-and-conquer bidirectional search: First results. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI)*, 1999.

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

[68] M. Likhachev, G. Gordon, , and S. Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Proc. of Advances in Neural Information Processing Systems (NIPS)*, 2003.

[69] J. Lopez and D. O'Hallaron. Evaluation of a resource selection mechanism for complex network services. In *Proc. of the IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, 2001.

[70] J. MacLaren, R. Sakellariou, K. Krishnakumar, J. Garibaldi, and D. Ouelhadj. Towards service level agreement based scheduling on the grid. In *Proc. of ICAPS'04 Workshop on Planning and Scheduling for Web and Grid Services*, 2004.

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

[72] D. V. McDermott. Estimated-regression planning for interactions with web services. In *Proc. of International Conference on AI Planning and Scheduling (AIPS)*, 2002.

[73] N. Muscettola. Computing the envelope for stepwise-constant resource allo-

cations. In *Proc. of International Conference on Principles and Practice of Constraint Programming (CP)*, 2002.

[74] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[75] X. Nguyen and S. Kambhampati. Reviving partial order planning. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

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

[77] E. P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. International Conference on the Principles of Knowledge Representation and Reasoning (KR)*, 1989.

[78] J. Penberthy and D. Weld. Temporal planning with continuous change. In *Proc. of National Conference on Artificial Intelligence (AAAI)*, 1994.

[79] J. S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. International Conference on the Principles of Knowledge Representation and Reasoning (KR)*, 1992.

[80] G. Rabideau, B. Engelhardt, and S. Chien. Using generic preferences to incrementally improve plan quality. In *Proc. of International Conference on AI Planning and Scheduling (AIPS)*, 2000.

[81] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proc. of the IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, 1998.

[82] R. Raman, M. Livny, and M. Solomon. Policy driven heterogeneous resource co-allocation with gangmatching. In *Proc. of the IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, 2003.

[83] I. Refanidis and I. Vlahavas. Heuristic planning with resources. In *Proc. of European Conference on Artificial Intelligence (ECAI)*, 2000.

[84] P. Reiher, R. Guy, M. Yarvis, and A. Rudenko. Automated planning for open architectures. In *Proc. of International Conference on Open Architectures and Network Programming (OPENARCH)*, 2000.

[85] S. Russell. Efficient memory-bounded search methods. In *Proc. of European Conference on Artificial Intelligence (ECAI)*, 1992.

[86] J. Shin and E. Davis. Continuous time in a SAT-based planner. In *Proc. of National Conference on Artificial Intelligence (AAAI)*, 2004.

[87] D. Smith, J. Frank, and A. Jónsson. Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, 15(1), 2000.

[88] B. Srivastava. Realplan: Decoupling causal and resource reasoning in planning. In *Proc. of National Conference on Artificial Intelligence (AAAI)*, 2000.

[89] H. Tangmunarunkit, S. Decker, and C. Kesselman. Ontology-based resource matching in the Grid - the Grid meets the Semantic Web. In *Proc. of the First*

Workshop on Semantics in Peer-to-Peer and Grid Computing, Budapest, Hungary, 2003.

[90] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., 2002.

[91] M. Ginsberg W. Harvey. Limited discrepancy search. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

[92] Z. Wang and D. Garland. Task-driven computing. Technical Report CMU-CS-00-154, School of Computer Science, Carnegie Mellon University, 2000.

[93] D. Wilkins and M. desJardins. A call for knowledge-based planning. In *Proc. of AIPS Workshop on Analysing and Exploiting Domain Knowledge for Efficient Planning*, 2000.

[94] E. Winner and M. Veloso. Automatically acquiring planning templates from example plans. In *Proc. of the AIPS-2002 Workshop on Exploring Real-World Planning*, 2002.

[95] S. Wolfman and D. Weld. Combining linear programming and satisfiability solving for resource planning. *Knowledge Engineering Review*, 2001.

[96] R. Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. In *Proc. of the IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, 1997.

[97] Web Service Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[98] M. Yarvis, P. Reiher, and G. Popek. Conductor: A framework for distributed adaptation. In *Proc. of Workshop on Hot Topics in Operating Systems (HotOS)*, 1999.

[99] H. Younes and R. Simmons. On the role of ground actions in refinement planning. In *Proc. of International Conference on AI Planning and Scheduling (AIPS)*, 2002.

[100] A. Zaremski and J. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.

[101] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proc. of IEEE Infocom*, volume 2, pages 594–602, 1996.

[102] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new resource ReSerVation Protocol. *IEEE Network*, 1993.

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