Configuration as Diagnosis: Generating Configurations with Conflict-Directed A* - An Application to Training Plan Generation -

Florian Grigoleit, Peter Struss
Technische Universität München, Germany
email: {struss, grigolei}@in.tum.de

Abstract

Although many approaches to knowledge-based configuration have been developed, the generation of optimal configurations is still an open issue. This paper describes work that addresses this problem in a general way by exploiting an analogy between configuration and diagnosis. Based on a problem representation consisting of a set of ranked goals and a catalog of components, which can contribute in combination to their satisfaction, configuration is formulated as a finite constraint-satisfaction-problem. Configuration is then solved by state-search, in which a problem solver selects components to be included in an appropriate configuration. A variant of Conflict-Directed A* has been implemented to generate optimal configurations. To demonstrate its feasibility, the concept was applied, among other domains, to personalized automatic training plan generation for fitness studios.

1 Introduction

Besides diagnosis, the task of configuration has been one of the earliest application areas of work on knowledge-based systems, initially in the form of rule-based "expert systems", for instance in [1]. Today, systems for automated configuration have reached maturity for practical applications, as shown in [2], [3], and [4]. Despite this success, developing algorithms for computing **optimal** or optimized configurations with general applicability still deserves more research efforts.

Driven by a number of different configuration tasks, we developed GECKO (Generic constraint-based Konfiguration), a generic solution to the configuration problem that can be specialized to different application domains and that, among other objectives, aims at supporting the generation of optimal configurations.
In a nutshell, the solution exploits an analogy:

- The configuration task can be seen as searching for an assignment of *active* or *non-active* to the components in a given repository, representing whether or not a component is included in the configuration, such that it achieves some goals in an optimal way
- Diagnosis has been formalized as a search for an assignment of behavior modes (*normal* or *fault_x*) to a set of system components such that it is optimally compliant with a set of observations.

Based on this analogy, we exploit a search technique that has been developed as consistency-based diagnosis, see [5]), and as a generalization for optimal constraint satisfaction, called conflict-directed A*, see [6]

In the following section, we discuss related work on configuration systems. In section 3, we present some examples of configuration problems that we tackled using GECKO and that will serve for illustration purposes. Next, we introduce our formalization of the configuration task and the key concepts of GECKO. In section 5, we discuss the analogy between diagnosis and configuration, the application of CDA*, variants of utility functions and how they relate to different types of configuration applications. The results are shown in section 6. Finally, our current work and some of the open issues are discussed.

2 Knowledge-based Configuration

Applications of configuration are immensely diverse, but they all share a number of common problems, such as compliance with domain knowledge, size of the solution space, and the resulting complexity of the problem solving task. It requires knowledge-based approaches to support the problem-solving activities, such as product configuration or variability management see [3] and [4].

Current research on configuration, especially for large applications, tends to neglect global optimization, focusing on local optimization, user interaction, or aiming at producing "good" solutions, see [3] and [7].

The focus of this paper is a generic, constraint-based configurator (GECKO) for solving optimal configuration problems. The core of GECKO is a variant of Brian Williams' Conflict-Directed A* (CDA*, [6]). The solution works on a generic representation of configuration knowledge and tasks. We consider the task of generating configurations as similar to consistency-based diagnosis. Instead of assigning modes for fault identification as in [5], GECKO assigns the activity to components contributing to goals. A configuration is consistent if all task-relevant goals are satisfied. The quality of a configuration is given by the level of goal satisfaction and the amount of resource consumption. Our approach allows the arbitrary selection of optimization criteria, like minimal resource consumption or maximal goal contribution. In the presented case study, our aim was to maximize the number of satisfied goals under consideration of available resources.

3 Application Examples

Configuration problems are almost ubiquitous in modern life, with applications as different as creating a customized computer as done by R1 in [1] and adapting the system functions of a car, see [8]. To illustrate the versatility of GECKO, we present three applications.

3.1 Car Configuration

Today, car manufacturers offer a vast number of models, model variants, and equipment options to their customer. The resulting complexity does not only prohibit a comprehensive exploration of the solution space, but is also likely to provide customers with sub-optimal car variants. A domain model for car configuration was created and mapped to the GECKO concepts, which are presented in 4.2([9])

3.2 User Interface Configuration

The Beam Instrumentation group at CERN is responsible for the design and implementation of particle beam measurement systems. These systems are specifically built for each case, resulting in extensive work on constructing them. While the generation of the GUIs, that is the implementation, is automated, the configuration is not. This task currently requires an expert to select libraries, graphical elements, and data sources and to parameterize them. Such tasks are typical configuration tasks and thus enable the automation of the configuration of the GUIs by GECKO ([10]).

3.3 Training Planning in Sport Science

At a first glance, training planning may appear to be a typical scheduling task, instead of a configuration problem. Taking a closer look shows that it mainly involves activities we consider the core of configuration: selecting, parameterizing, and arranging components to satisfy goals, whereas assigning time slots to the selected exercises is, in general, fairly straightforward

A trainer has to analyze the biometric state of his trainee, such as fitness or age, to consider constraints on the created training plan, for example duration or available equipment, and to select and order appropriate exercises.

The sheer number of existing exercises and the size of the solution space show that training planning includes optimization. In general, a trainer tries to maximize the training effect within the available time and under consideration of the trainee's goal and abilities. The specialization of GECKO to training planning is described in section 5.

4 GECKO - Foundations

4.1 Intuition

With GECKO, we aim at developing a generic solution to configuration problems, which can be tailored towards a particular domain by specializing some basic classes and creating a knowledge base in terms of domain-specific constraints. Its design is driven by the following objectives:

- supporting both automatic and interactive configuration;
- enabling the use of the system without deep domain knowledge, esp. about how high-level goals of the user break down to more detailed and technical ones;
- handling also soft domain constraints and user preferences, and
- offering support to the user by providing explanations for generated parts of the configuration and for unavailable options and by suggesting revisions to resolve inconsistencies.

However, this paper focuses on the basis, a generic problem solver for (optimal) configuration. Determining the solution – the configuration - means selecting a set of instances of given types of elements - components -, perhaps with certain attribute values and organized in a particular structure. The configuration has to

1. satisfy a set of high-level user goals,
2. be compliant with particular attributes and restrictions supplied by the user,
3. be realizable both in principle (i.e. not violating domain-specific restrictions on valid configurations),
4. under consideration of available resources, and
5. optimal (or near optimal) according a criterion that reflects the degree of fulfilling the goals and the amount of resources consumed.

Configurations can be physical devices, such as turbines, communication systems, and computers, abstract ones like a curriculum or a company structure, or a software system. In contrast to a design task involving the creation of new types of components, configuration assumes that all required Components are instances of component types from a repository ([11]). This leads to different kinds of reasoning involved: innovative design has to verify that its result satisfies the goals by inferring that they achieved by the system behavior based on behavior models of the components, whereas for a configuration task, it is assumed that behavioral implications of aggregated components have been compiled into explicit interdependencies of Goals and Components. As a result, software systems for configuration are typically based on knowledge encoded as constraints or rules, as in [1] and [2], and do not require the exploitation of behavior models.

4.2 Core Concepts

The core concepts of GECKO are derived from the description above, as depicted in Fig. 1:

- **Goals** express the achievements expected from a specific configuration. They may have an associated priority dependent on the task and different criteria for goal satisfaction.
- **Components** are the building blocks of the Configuration. They may be organized in a type hierarchy (for example, Lithium battery is a voltage source). In addition, there may be Components that are aggregations of lower level components.
- A **Task** specifies the requirements on a configuration from the user's perspective. It is split into three kinds of restrictions:

Task
= TaskGoals ∪ TaskParameters ∪ TaskRestrictions.

TaskGoals are a collection of Goals the user is aware of and which can be (de)activated or prioritized by the user. Each $TaskGoal_n$ is stated as a restriction $TaskGoal_n$.Satisfied=T in the Task description.

While **TaskGoals** represent objectives a user requires, **TaskParameters** associate values to properties of the Task, hence have the form $TaskParameter_k=value_{kj}$. For instance, in vehicle configuration, the target country may have an influence on daytime running lights being mandatory. However, these implications are not drawn by the user (who only provides the country information), but by the domain knowledge represented in the system.

In contrast, **TaskRestrictions** refer explicitly to the choice of Components and their attributes, e.g. that for the user, a convertible is not an option or that the engine should be a Diesel engine.

A specific, and often essential, TaskRestriction can be included:

- A **ResourceConstraint** limits the cost of the configuration, which may be indeed money (car configuration) or time (in training plan configuration), but also computer memory etc. Components have to have an attribute that allows calculating the resources needed for the entire configuration (often as the sum).

4.3 Constraints on Configurations

The configuration knowledge of a particular application comprises the domain-specific specialization and instantiation of Goals, Components (possibly including component attributes and their domains), and relevant TaskParameters and their domains as well as constraints that capture interdependencies among these instances. Dependent on which kinds of objects are related, we distinguish between the following (illustrated in Fig. 1):

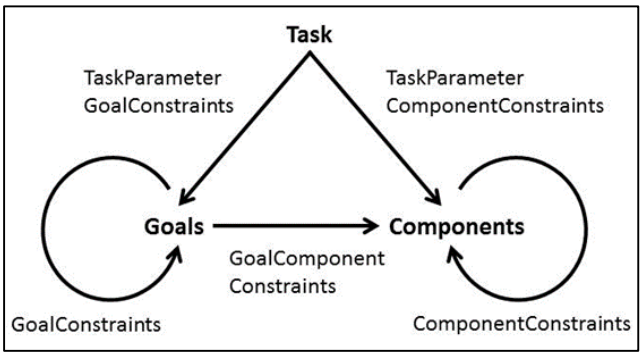

Fig. 1 Task constraints in GECKO

- **TaskParameterGoalConstraints** express that certain TaskParameter values may exclude or require certain goals
- **GoalConstraints** relate goals to each other, in particular for refinement of higher-level (esp. TaskGoals) to lower-level ones, such as goals related to various muscle groups that should be exercized, although the user is not aware of this
- **GoalComponentConstraints** capture essential configuration knowledge, namely whether and how the available components contribute to the achievements of goals
- **ComponentConstraints** establish interdependencies among components (and their attributes): a component may be dependent on or incompatible with the presence of another component in the configuration
- **TaskParameterComponentConstraints** may include or exclude certain components based on TaskParameter values

A fundamental constraint type is

Requires (x, y)

which is defined by

$x.active=T \Rightarrow y.active=T$

and used to express dependencies among goals (e.g. refining a goal to a set of mandatory sub-goals) and components (e.g. cruise control requires automatic transmission) and as the fundamental coupling between goals and components (to achieve high-speed driving, an engine of a certain power is needed). Furthermore, in order to express that several goals or components provide some partial contributions that jointly result in the satisfaction of a goal (or establish the preconditions of a component), we introduce the concept of a **choice**, which can also fill the role of y in a Requires-constraint. A choice is given by a relation

$GoalChoice \subset Goals \times ContributionDom$

or

$ComponentChoice \subset Components \times ContributionDom$,

where ContributionDom specifies a set of values for quantifying how much a goal or component contributes to the satisfaction of the choice and needs a zero element and an operator Σ to add up contributions (e.g. addition of integers). The idea behind choices is implemented by three kinds of constraints. The degree of the satisfaction of a (component) choice is given by the combined contributions of the active components of the choice:

$Choice.satLevel = \Sigma\ Choice.goal.actContribution$

and

$Choice.goal.actContribution =$

$Choice.goal.contribution$ IF goal.active=T

zero IF goal.sctive=F .

The choice is satisfied, if the satLevel lies in a specified range, satThreshold:

$Choice.active = T \Leftrightarrow$

$Choice.satLevel \in Choice.satThreshold$.

This allows implementing not only a minimum level as a precondition for the satisfaction of a choice, but also a maximum. Preventing "over-satisfaction" may not be a common requirement, but in the fitness domain, one may want to restrict the set of exercises that impose a load on a particular muscle group.

Another predefined general type of constraint is

Excludes (x, y)

defined by

$x.active=T \Rightarrow y.active=F$

to express conflicting goals, incompatible components, and TaskParameterGoal/ComponentConstraints (e.g. high body weight may rule out certain exercises).

The application-specific configuration knowledge is, thus, basically encoded as a set of the constraints explained above. This, together with the domain-specific ontology (as a specialization of the basic GECKO concepts, including choices, and associated attributes) and, perhaps, specific contribution domains and operators, establishes the con-

figuration knowledge base, called **ConfigKB** in the following.
We make some reasonable fundamental assumptions about ConfigKB:

- Each potential **TaskGoal is supported**: it is the starting node of a connected hyper graph of Requires constraints that includes components, i.e. it actually needs a (partial) configuration in order to be satisfied (which does not mean it can actually be satisfied).
- **Closure assumption**: the encoded interdependencies, esp. the Requires constraints, are complete. In other words, if all constraints Requires (x, y) associated with x are satisfied by a configuration, then x is satisfied.
- It is **consistent**.

4.4 Definition of the Configuration Task

The goal is to select an appropriate subset of the available components, which we call the active ones, and possibly determine or restrict their attributes.

Definition 1 (Complete Configuration)
A **configuration**
PARCONFIG = (ACTCOMPS, COMPATTR)
is **complete** if includes exactly the active components:

$$comp \in ACTCOMPS \Leftrightarrow comp.Active = T.$$

GECKO has to generate a configuration PARCONFIG that satisfies the criteria stated in section 4.1.

Definition 2 (Solution to a Configuration Task)
A **configuration task** is a pair

(ConfigKB, Task)

(as specified in sections 4.3 and 4.2, respectively), and a complete configuration PARCONFIG is a **solution** to it, if it is consistent with the ConfigKB and the Task,

$$PARCONFIG \cup ConfigKB \cup Task \not\models \perp .$$

This may seem too weak, because criterion 1 in section 4.1 requires the entailment of the satisfaction of the TaskGoals in Task.

Proposition 1
If PARCONFIG is a solution to a configuration task (ConfigKB, Task), then

$$PARCONFIG \cup ConfigKB \models \forall\, goal \in TaskGoals\ goal.Satisfied = T.$$

This follows from the closure assumption: Since for the chosen TaskGoals, Satisfied=T is explicitly introduced in Task, it follows that all Requires constraints related to them are satisfied, and, hence, they are not only consistent, but entailed. As for the other criteria of section 4.1:

2. Compliance with specific application requirements is guaranteed by consistency with the TaskParameters under the TaskParameter-Goal/ComponentConstraints in ConfigKB and with TaskRestrictions in Task
3. Realizability is established by consistency with ComponentConstraints
4. The ResourceConstraint is also consistent.

Criteria 5, optimality, will be discussed in the following section.

5 Generating (Near) Optimal Configurations

5.1 Configuration as Diagnosis

The current version of GECKO is based on the assumption that there exists a **finite set of components, COMPS**, as a repository for all configurations. This means, no new instances of components types are created during configuration and, more specifically, a component will not be duplicated if it is included in the configuration due to several constraints. In this case, determining ACTCOMPS of a complete configuration can be seen as an activity assignment

$$AA: COMPS \rightarrow \{active, inactive\},$$

indicating the inclusion in or exclusion from the configuration, and the consistency test of Definition 2 becomes

$$AA \cup ConfigKB \cup Task \not\models \perp .$$

This representation shows the analogy to the consistency-based formalization of component-oriented diagnosis: an assignment MA of modes (i.e. nominal or faulty behavior) to a set of components,

$$MA \rightarrow \{OK, fault1, fault2, \ldots\}$$

characterizes a diagnosis, if it is consistent with the domain knowledge (a library of behavior models and a structural description), called system description, SD, and a set of observations, OBS:

$$MA \cup SD \cup OBS \not\models \perp .$$

In both cases, the **assignments to the components**

$$AA \leftrightarrow MA$$

are checked for consistency with a fixed set of constraints representing the **domain knowledge**

$$ConfigKB \leftrightarrow SD ,$$

and a set of constraints representing a **specific problem instance**

$$Task \leftrightarrow OBS .$$

In consistency-based diagnosis, theories and algorithms have been developed to determine diagnostic solutions, which can be exploited for the configuration task based on the analogy outlined above.

5.2 Conflict-directed A*

Based on the above formalization, many implementations of consistency-based diagnosis exploit a best-first search for consistent mode assignments, using probabilities of individual behavior modes as a utility function (and usually making the assumption that faults occur independently) as SHERLOCK does([12]). Classical A* search has been extended and improved by pruning the search space based on inconsistent partial mode assignments that have been previously detected during the search (called **conflicts**), exploiting a truth-maintenance system (TMS, such as the assumption-based TMS [13]) as a dependency recording mechanism that delivers conflicts. From the diagnostic solutions, this approach has been generalized later as conflict-directed A* search, see [6].

Procedure CDASTAR

1) Terminate=F
2) Solutions=∅
3) Conflicts=∅
4) VA=$VA_{initial}$
5) DO WHILE Terminate=F
6) Apply Constraints(VA)
7) Check consistency of VA
8) IF consistent
9) THEN append VA to Solutions
10) Terminate=Solutions.Terminate
11) ELSE
12) Conflicts=APPEND(Conflicts, newConflicts)
13) END IF
14) VA=Conflicts.BestCandidateResolvingConflicts
15) END DO WHILE
16) RETURN Solutions

The effectiveness of the pruning of the search space based on previously detected inconsistencies (highlighted in the above pseudo code) grows with the number of (non-redundant) conflicts that are extracted. Achieving this, however, can be computationally expensive and may have to be traded off against the computational cost of the consistency test and/or the optimality of the solution. We will get back to this issue below.

The straightforward mapping of the configuration problem to CDA* is obtained by representing configurations as variable assignments:

VARS={ $Comp_i$.active | $Comp_i$∈COMPS}
DOM($Comp_i$.active)={T, F} .

To illustrate how the algorithm works using a simple example, assume that goal G_1 depends on a component choice that involves 3 components, C_i, each with a contribution of 1 in this choice, which has a satisfactionThreshold (2,3), i.e. it is satisfied if at least two of the components are active. Search starts with an empty configuration (active=F for all components) which leads to an inconsistency with the constraints related to the choice. Each pair of inactive components establishes a (minimal) conflict:

{ C_1,active=F, C_2,active=F },
{ C_1,active=F, C_3,active=F },
{ C_2,active=F, C_3,active=F }.

Configurations resolving these conflicts are the ones with active components

{ C_1, C_2}, { C_1, C_2}, or { C_2, C_3},

and the best one would be checked further. If this is done against another choice for a goal G_2, which is based on components C_3, C_4, C_5 (again all with contribution 1) and a threshold (1, 3), then a new conflict

{ C_3,active=F, C_4,active=F, C_5,active=F }

is detected, and the configurations resolving all include active components are

{ C_1, C_3}, { C_2, C_3}, { C_1, C_2, C_4}, or { C_1, C_2, C_5}.

5.3 Diagnosis vs. Configuration

Despite the mentioned basic commonality, there are some important distinctions at a conceptual level, but with a potentially strong impact on the computational complexity.

Partial vs. complete assignments

In diagnosis, it is possible to check partial mode assignments to detect useful conflicts. In configuration, we have to consider **complete** variable assignments, which, in assigning T or F to activity variables of **all** components, correspond to complete configurations. The reason is that, as illustrated by the above trivial example, the constraints related to a choice deliver important **conflicts** based on components being **not active**. A partial configuration, e.g. assigning active=T to, say, C_1 only, is consistent with the respective choice; that this configuration does not satisfy G_1 is detected only, if all other components are assumed to be inactive (Of course, if the satThreshold has an upper limit, we obtain conflicts involving too large sets of active components, as well). This observation is related to another difference:

(NON-)Locality of the Domain Theory

In diagnosis, the domain theory is as modular as the device: it consists of constraints that represent the local interaction of components and constraints that capture the local behavior of components under certain modes. Checking the consistency of a partial mode assignment requires applying the directly related constraints only. In contrast, constraints representing configuration knowledge are almost by definition non-local: they are meant to relate many components across the entire configuration, e.g. as choices. If choices play a major role and are large, this can be a source of severe problems.

The training plan generation application forms an extreme example: choices may involve in the order of 100 components, because many exercises may be related to a particular muscle group, while only a handful of them together satisfy the goal. In addition, exercises are challenging several muscle groups. If the lower boundary of the satisfactionThreshold of a choice is k and the size of the choice is n, then (assuming a contribution 1 for each component), the number of resulting minimal conflicts will be

$$\binom{n}{k-1}$$

– prohibitively large in the training application. This has an impact on the algorithm, as discussed in section 5.5.

First, we have to introduce appropriate utility functions to measure the quality of a configuration.

5.4 Utility Functions

The utility of a configuration should essentially reflect

- the degree of **fulfillment** of the relevant **goals** and
- the amount of resources required.

A measure of the former may also consider **priorities** of goals. The same holds for individual components. Since inactive components neither make contributions nor consume resources, it is plausible to assume that the **utility** of a configuration depends on its **active components only**.

In the following, it is assumed that

- the contribution of a configuration is obtained solely as a **combination of contributions** of the active components included in the configuration and otherwise independent of the type of properties of the components,
- we can define a subtraction "-" of contributions,
- the cost of the contribution is given as the sum of the cost of the involved active components and will usually be numerical,

- we can define a ratio "/" of contributions and resources,
- there is a function that maps priority of goals to a weight of the contributions and a kind of multiplication,
  *: DOM(weight)xDOM(contribution)
  → DOM(contribution),

  is defined.

Then the following specifies a family of utility functions (where we simplify the notation by writing $Goal_j$.SatThreshold instead of $Choice_j$.SatThreshold etc.):

For an active Goal Goalj, the TotalContribution of a Configuration is

Configuration.TotalContribution($Goal_j$):=
$\Sigma_{Compi \in Configuration.ACTCOMPS}$ $Comp_i$.Contribution($Goal_j$)

where Σ denotes the Combine operation, the ActualContribution is given as

Configuration.ActualContribution($Goal_j$):=
max(Configuration.TotalContribution($Goal_j$),
$Goal_j$.Combine($Goal_j$ SatThreshold,posTolerance)),

and a penalty (for over-satisfaction) as

Configuration.Penalty($Goal_j$):=
max (0,
Configuration.TotalContribution($Goal_j$)
- $Goal_j$.Combine($Goal_j$ SatThreshold,posTolerance).

Based on this, we define the utility function as

Configuration.Utility(ACTGOALS):=
$\Sigma_{Goalj \in Configuration.ACTGOALS}$
weight($Goal_j$.Priority)
* Configuration.ActualUtility($Goal_j$)
+ f * Configuration.Penalty($Goal_j$))
/ $\Sigma_{Compi \in Configuration.ACTCOMPS}$ $Comp_i$.Resource.

The factor f determines whether or not excessive contributions are penalized (by the excessive amount); the weight can emphasize contributions to Goals with high priority, and the tolerance interval can express how exactly the intended SatThreshold has to be hit.

5.5 GECKO Algorithm

For the GECKO variant of CDA* we modified CDA* by activating only the constraints needed at a specific stage, thereby reducing the number of occurring conflicts significantly.

GECKO characterizes a stage in the problem solving process and hence the criteria for constraint activation as a pair

S = (GOALS, configuration),

that is a set of goals that are considered and a configuration to be checked for consistency. This allows for search strategies that do not consider all active goals from the beginning. Therefore, the constraints to be applied are not only determined by the variable assignment, but also by the goals. In our first application, goals are activated in a descending order, according to their priority.

To determine the hitting sets of the conflicts we use different algorithms from [14], depending on the domain. In BestCandidateResolvingConflicts, the next-best solution is generated.

Procedure GECKO Configuration Algorithm

```
1)  ApplyConstraints(Constraints(Initial)
2)  ActComps=ACTCOMPS0
3)  Priority=max(actGoals.Priority)
4)  DO WHILE Priority >=1
5)     ApplyConstraints(Constraints(
          GoalPriorityClass(Priority))
6)     VA=VA(ActComps,COMPS\ActComps)
7)     NewActComps=
          GECKO.CDASTAR(VA).ActComps
8)     ActComps = NewActComps.Commit
9)  END DO WHILE
10) RETURN ActComps
```

In line 8, the algorithm fixes the components added to satisfy the recently considered goals. This means, when trying to satisfy further goals (with lower priority) they will not be de-activated. This heuristic aims at satisfying as many goals as possible with the given resources in the order of their priority, but, obviously, may miss a globally optimal solution.

6 Case Study: Training Plan Generation

We are working on the realization of the three applications presented in section 3. To demonstrate the specialization of GECKO concepts and the capabilities of the GECKO algorithm, we selected the fitness training example. From the three examples, fitness is best suited to illustrate the advantages of CDA* in configuration.

6.1 Domain Theory

In fitness, trainees perform exercises, like push-ups or running, to train body parts under certain aspects (endurance, muscle gain). To train means to improve physical abilities, like endurance, and to influence biometric parameters, such as weight. In configuration terms: exercises contribute to a set of fitness goals. Hence, we created the domain theory for training planning using the concepts specified in section 4.2. Table 1 contains an overview on the most important specializations.

The result may appear straightforward to outsiders, but it is actually the result of several months of analyses carried out jointly with experts from sports sciences, which took as to several versions and revisions of the model.

Table 1: Specialization of GECKO Concepts

GECKO Concept	Fitness Concept	Example
Goal	TraineeGoal	Muscle Gain
	TrainingGoal	Strength
	TargetGoal	Biceps
Component	Exercise	Push-up
Task	Trainee	-
TaskRestriction		TrainingDuration
TaskParameter	TrainingProperty	Equipment
	TraineeProperty	Fitnesslevel

Task

A GECKO Task in fitness is a trainee, or more precisely the request of a training plan by a trainee. A trainee has expectations regarding the result of the training, represented by TraineeGoals. The Trainee also has a set of TraineeProperties, like Fitnesslevel, and sets the TrainingProperties. Furthermore, a trainee has to specify the desired TrainingDuration.

Special among the TraineeProperties are the FitnessTargets and FitnessCategories. A FitnessTarget has to be

trained by an Exercise, such as legs. FitnessCategories are the main abilities of a Trainee, such as strength.

Goals

The domain theory contains three types of goals:

- TraineeGoal: The only **TaskGoal** in fitness, describing the expected effect of the fitness training
- TrainingGoal: Abstract goals, specifying the type of physical ability to be improved e.g. strength
- TargetGoal: Body part the training has to stimulate.

To capture the structure of the human body and the differences in fitness categories, we decompose the TargetGoals into three levels:

- RegionGoal
- MuscleGroupGoal
- MuscleGoal

Reflecting the FitnessTargets, TargetGoals are structured in a goal tree. Because FitnessTargets are trained at different levels, the tree is unbalanced. For example Endurance is generally trained for the whole body, while Strength is trained at a muscular level.

Components

All components in fitness are exercises. Each exercise is related to a FitnessCategory, e.g, pushup is a StrengthExercise. Exercises can contribute to multiple TargetGoals, but only TargetGoals of their own FitnessCategory. For example, a StrengthExercise can only contribute to TargetGoals related to strength.

Exercises comprise a set of fixed attributes, such as requiredEquipment or requiredFitnesslevel, as well as a set of unspecified attributes, like TrainingWeight or Durations. The values of such volatile attributes depend on the selected TraineeGoal, because they define how an exercise effects a FitnessTarget – an increase in strength is achieved by a small number of slow repetitions with very high weight, while fat is burnt best with many fast repetitions with little weight.

Utility

The utility of a configuration in SmartFit depends on the contributions of the active components to required Choices
DOM($comp_i$.$contribution_i$) ={20,40,60,80,100}
The satThreshold of the Choices depends on the priority of the associated goal
satThreshold = combine($Goal_i$.Priority,normThreshold),
with DOM(Priority) ={1,2,3,4,5}.

For the example in 6.2, we simply multiplied the priorities with the normThreshold =80.
The domain of the combined contribution is from 0 to 500 in steps of 20. In case of contributions larger than 500, the overshoot is cut, and the value set to 500.

The utility for fitness training is given by the following equation:

Config.Utility (ACTGOALS):=
$\Sigma_{Goalj \in Config.ACTGOALS}$
weight ($Goal_j$.Priority) * Config.ActualUtility ($Goal_j$))
/ $\Sigma_{Compi \in Config.ACTCOMPS}$ Compi.Resource.

6.2 Simplified Example

To make the capabilities of GECKO more tangible, we present a small experiment. For brevity and clarity, we use a reduced knowledge-base, with three MuscleGoals(table 2), 12 exercises(table 3), and 2 TaskParameters, namely Equipment and a general Fitnesslevel – thus omitting the consideration of different Fitnesslevels related to the specific FitnessTarget, as done in the application system. Furthermore, we set the duration of all exercises to require 5 minutes.

Using this reduced knowledge-base, we applied both the basic GECKO algorithms and the goal-focused variant. The results are described in the following subsection.

Table 2: Exemplary muscle goals with priorities

ID	MuscleGoal	Priority: MuscleGain	Priority: GeneralFitness
G1	**Biceps**	1	2
G2	**Triceps**	1	2
G3	**Latissimus**	2	3

Table 3: Exercises and parameters

ID	Exercise	Contributions	Required Equipment	Required Fitness level
C1	**Biceps Curl**	Biceps: 100	None	1
C2	**Dips**	Triceps: 100 Latissimus: 20	None	1
C3	**Lat-Pull**	Biceps:20 Latissimus: 100	Machines	1
C4	**Rev. Butterfly**	Triceps: 40	Machines	1
C5	**Pushup**	Triceps: 80	None	2
C6	**Pushup on knees**	Triceps: 60	None	1
C7	**Shoulder press**	Triceps: 80	Machines	1
C8	**Rowing**	Biceps: 40 Latissimus: 80	Machines	1
C9	**Pull up**	Biceps: 100 Latissimus: 80	None	2
C10	**Triceps Pulldown**	Triceps 100	Machines	2
C11	**Pull up (supported)**	Biceps: 20 Latissimus: 80	None	1
C12	**Rowing one-armed**	Biceps: 40 Latissimus: 100	Machines	2

To compare the results of different tasks, we conducted to experiments with different TraineeGoals and TaskParameter values. For the basic algorithm, we used the Tasks shown in Table 4.

Table 4: Task for experiments A and B

Variable	Values A	Values B
TaskGoal	General Fitness	Muscle Gain
TaskParameter: **FitnessLevel**	Untrained (1)	Trained (2)
TaskParameter: **Equipment**	Machines	none
TaskRestriction: TrainingDuration	15 minutes	30 minutes

The results of the configuration with the basic GECKO algorithm are shown in Tables 5 and 6.

Table 5: Configuration results basic GECKO Algorithm

Experiment A	Experiment B
Lat-Pull	Pull up
	Dips
Rowing	Pull up (supported)
	Pushup on knees
Shoulder press	Biceps Curl

Table 6: State of the Goals after running the basic Algorithm

Muscle Goal	Value A	Value B
Biceps	Partially Satisfied	Satisfied
Triceps	Satisfied	Satisfied
Latissimus	Satisfied	Partially Satisfied

Application Evaluation

The results indicate that the GECKO algorithms are capable of generating optimal solutions to configuration problems. In experiment B, it can be seen that GECKO was not able to satisfy G3 completely, since there were not enough consistent exercises available. Thus, the less important goals were satisfied, but not the important one. In experiment A on the other hand, the algorithm was able to fully satisfy G3 but not G1, since the duration resource was only sufficient for three exercises.

6 Discussion and Outlook

The results shown above indicate that treating configuration as a diagnostic problem, and solving it with techniques from consistency-based diagnosis is a promising approach to user-oriented configurators for optimal configuration problems.

The analysis of different application domains, including the ones mentioned in section 3, triggers the insight that variations of the search algorithm may be required in order to reflect the specific requirements and structure of the problems. This is particularly true for applications that involve a high level of interaction, such as leaving choices to the user, providing explanations for system decisions, and allowing him to modify his/her decisions in an informed way. Retracting decisions and also generating explanations can be supported by the ATMS, which also produces conflicts.

The conceptual and algorithmic solution to configuration generation presented in this paper could certainly be implemented using other techniques that have been proposed and used for configuration. However, our choice of an ATMS-based solution (and CDA*) was strongly motivated by the overall objectives stated in section 4.1: we intend to base explanation facilities ("which user inputs and domain restriction prevent option x to be viable?"), preferences and soft constraints, and the possibility to retract input and explore several alternative solutions on capabilities of the ATMS.

A goal of our work is to extract features from the case studies that can support a classification of configuration applications as a basis for selection from a set of predefined algorithm variants and strategies for man-machine interaction.

Other options, such as compiling (parts of) the constraint network and moving search heuristics to a lower technical level (the constraint system) will also be explored.

Furthermore, we are currently preparing an application to configuration of automation systems for collaborative, flexible manufacturing and modular multi-purpose vehicles. This application of GECKO is likely to require stronger spatial and also temporal constraints for structuring a configuration.

Acknowledgments

We would like to thank our project partners for providing their domain knowledge and their assistance, esp. Florian Kreuzpointner and Florian Eibl. Special thanks to Oskar Dressler (OCC'M Software) for proving the constraint system (CS3 or Raz'r). The project was funded by the the German Federal Ministry of Economics and Technology under the ZIM program (KF2080209DB3).

References

[1] JP McDermott, "RI: an Expert in the Computer Systems Domain", *Artificial Intelligence*, 1980.

[2] U. Junker, D. Mailharro, "The logic of ILOG(J) Configurator: Combining Constraint Programming with a Description Logic", *IJCAI*, 2003.

[3] A. Felfernig, L. Hotz, C. Bagley, and J. Tiihonen, "Knowledge-based Configuration: From Research to Business Cases."

[4] D. Sabin, R. Weigel, "Product configuration frameworks – a survey." *IEEE Intelligent System*, 1998.

[5] J. de. Kleer, BC Williams, "Diagnosing Multiple Faults," *Artificial Intelligence*, 1987

[6] BC William, R.J. Ragno, "Conflict-directed A* and its role in model-based embedded systems", *Discrete Applied Mathematics*, 2007.

[7] M. Stumptner, G. Friedrich, A. Haselböck, "Generative constraint-based configuration of large technical systems", *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 1998.

[8] G. Weiß, F. Grigoleit, P. Struss, "Context Modeling for Dynamic Configuration of Automotive Functions", *ITSC*, 2013

[9] C. Richter, "Development of an interactive car configuration system", *Master's Thesis, Tech. Univ. of. Munich*,

[10] A. Verikios, "A tool for the Configuration of CERN Particle Beam Measurement Systems", *Master's Thesis, Tech. Univ. of. Munich*,

[11] U. Junker, "Configuration." *Handbook of Constraint Programming, Configuration, p. 837-868*, 2006.

[12] J. De Kleer, BC Williams,"Diagnosis with Behavioral Model", *IJCAI*, 1993.

[13] J De Kleer, "An assumption-based TMS" *Artificial intelligence*, 1986.

[14] J De Kleer, "Hitting Set Algorithms for Model-based Diagnosis", *Principles of Diagnosis*, 2011.