# Graph-Based System Configuration*

Horst Moldenhauer

Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation,
Vincenz-Prießnitz Str. 3, 76128 Karlsruhe, Germany
molden@ipd.info.uni-karlsruhe.de

**Abstract.** We present an optimization problem that arises in the context of system-design and the refinement of high-level specifications. We present a graph-based formalization of the problem, thus defining a new optimization problem, which we call MINIMUM CONFIGURATION. We prove MINIMUM CONFIGURATION to be NP-hard and also hard to approximate, i.e. not to be in APX. These results are obtained by reduction from graph coloring and SHORTEST PATH WITH FORBIDDEN PAIRS. We show how to apply the introduced concepts to the application domain and propose a way to identify subclasses in the input space, for which MINIMUM CONFIGURATION can be solved efficiently.

## 1 Introduction

In this paper we present an optimization problem on graphs, that has applications to software construction using preexisting building blocks. Often the methodologies for reusing older work in software construction are based on libraries of building blocks and on the refinement of some high-level specification of the system to be developed [3, 10, 12]. Refinement of the specification implies selection of one implementation for each component of the system from a possibly large number of alternatives. We call this selection process the configuration of the system. Our work focuses on the automation of configuration. We base the selection process on the concept of cost, which is a means to formalize and rate the non-functional aspects of building blocks. While the system specification mostly defines the required functionality of each part of the system, handling and optimization of many non-functional aspects is deferred to the refinement process. The notion of cost may be used to express a variety of non-functional properties, e.g. runtime, resource allocation, development time or the possibility of fault. The task of configuration is to control the refinement process such that the cost of the final system is minimized. This minimization is not a trivial task, because decisions for or against certain implementation alternatives may depend on each other, that is the rate of a combination of implementations is different from the sum of the single implementation rates. Applications for system configuration include program construction using abstract data types, programming

* This work has been supported by the Deutsche Forschungsgemeinschaft under grant GRK 209/2-96

with federated databases, program acceleration using configurable hardware and the management of software development.

In the next section we define the theoretical foundations of system configuration and formally define MINIMUM CONFIGURATION. In section three and four we prove that MINIMUM CONFIGURATION is NP-hard and hard to approximate. In section five we demonstrate how to apply the introduced theoretical concepts to the application areas mentioned above and propose a way to identify subclasses of problem instances for which MINIMUM CONFIGURATION can be solved efficiently. Section six presents some examples for the application of configuration. Section seven concludes.

## 2 Graph-Based Modelling

In this section we define the theoretical foundations of configuration. We define MINIMUM CONFIGURATION and introduce the notion of configuration graph, that is used to model the input instances.

MINIMUM CONFIGURATION can be informally defined in the following way: Given a graph $G = (V, E)$ with weighted edges and a partition of $V$ into $n$ disjoint sets. We want to select exactly one vertex from each set of the partition, such that the subgraph $G'$, induced by these vertices, has minimal weight. $G'$ must not include edges with infinite weight. In the formal definition we follow Crescenzi and Panconesi [5], who define an NPO problem as a four-tuple $F = (\mathcal{I}_F, S_F, m_F, opt_F)$, where $\mathcal{I}_F$ is the space of input instances, $S_F$ is the space of feasible solutions, $m_F$ is the objective function and $opt_F$ defines the criterion of optimality.

**Definition 1.** MINIMUM CONFIGURATION:
$\mathcal{I} = \{G = \langle V, E\rangle : G$ is a graph, $P = (p_1, p_i, \ldots, p_n) : P$ is a partition of $V$ into $n$ disjoint sets, $w : E \mapsto \mathbb{N} \cup \{\infty\} : w$ is a weight function $\}$,
$S(\langle V, E, P, w\rangle) = \{(v_1, ..., v_n)$ a sequence of $n$ different vertices,such that $\forall i : v_i \in p_i \wedge \forall v_i, v_j : (v_i, v_j) \in E \Rightarrow w(v_i, v_j) \neq \infty\}$,
$triv(\langle V, E, P, w\rangle) = V$,
$m(\langle V, E, P, w\rangle, (v_1, \ldots, v_n)) = \sum_{\substack{(v_i, v_j) \in E \\ i,j \in [1..n]}} w(v_i, v_j)$,
opt $=$ min.

A graph with weighted edges together with a partition of its vertices is called a configuration graph. Given a configuration graph $G_C$, a solution to MINIMUM CONFIGURATION is called a *configuration* for $G_C$. We define a corresponding decision problem.

**Definition 2.** A configuration graph $G$ is *c-configurable* iff there exists a solution $s \in S(G)$, such that $m(s) \leq c$.

In the next section we prove that MINIMUM CONFIGURATION is NP-hard in the general case.

## 3 Minimum Configuration is NP-Hard

For proving NP hardness of Minimum Configuration we present a reduction from graph coloring [6].

**Theorem 3.** *Deciding c-configurability is NP-complete.*

*Proof.* We prove the theorem by reduction from graph coloring.
Given a graph $G = (V_G, E_G)$ and a number $k$, we want to decide whether $G$ is k-colorable or not. We construct a configuration graph $K = (V_K, E_K)$ using the following procedure:

**Construction of K.** For each vertex $A \in V_G$ we create a set $p(A)$ with $k$ vertices. Each vertex in $p(A)$ represents a possible coloring of $A$. We define $V_K := \bigcup_{A \in V_G} p(A)$ and regard each $p(A)$ to be a partition of $V_K$. Consider vertices $A$ and $B$ of $G$ and let $a \in p(A)$ and $b \in p(B)$. Let $col(a)$ and $col(b)$ denote the color that is represented by $a$ and $b$, respectively. If $(A, B) \in E$ we add an edge $(a, b)$ to $K$. The weight of $(a, b)$ is set to $c > 0$ if $col(a)$ equals $col(b)$ and to zero otherwise.

The construction of $K$ can be done in polynomial time. It is easy to see that there is a configuration for $K$ with a weight of zero, if and only if $G$ is $k$ colorable. So, if there was an algorithm that could decide in polynomial time whether $K$ is $(c-1)-configurable$, then we could decide in polynomial time whether $G$ is $k$-colorable or not. As deciding $k$-colorability is NP-complete (even for $k = 3$) [6], we conclude that deciding $c$-configurability is NP-complete. □

**Corollary 4.** Minimum Configuration *is NP hard.*

## 4 Minimum Configuration is Hard to Approximate

In this section we show that the configuration problem is hard to approximate, i.e that it is not in APX. In fact we demonstrate that a restricted version of Minimum Configuration is NPO PB-complete[2], so it cannot be approximated within $n^{\varepsilon}$ for some $\varepsilon > 0$, where $n$ is the size of the problem instance, provided that $P \neq NP$. To achieve these results, we construct a reduction from Shortest Path with Forbidden Pairs [9, 8], that preserves the value of the objective function[3]. First we formally define Shortest Path with Forbidden Pairs(from [9]).

**Definition 5.** Shortest Path with Forbidden Pairs:
$\mathcal{I} = \{G = \langle V, E \rangle : G$ is a graph, $s \in V, f \in V, P \subset V \times V\}$,
$S(\langle V, E, s, f, P \rangle) = \{(v_1, ..., v_k)$ a sequence of $k$ different vertices in $V$ such that

[2] NPO PB is the class of NPO problems, whose objective function is polynomially bounded by the size of the input instance [9]

[3] Note that this restates the fact that Minimum Configuration ist NP-hard.

$v_1 = s, v_k = f, \forall i \in [1..k-1]((v_i, v_{i+1}) \in E \wedge \forall j \in [i+1..k](v_i, v_j) \notin P\}$,
$triv(\langle V, E, s, f, P\rangle) = V$,
$m(\langle V, E, s, f, P\rangle, (v_1, \ldots, v_k)) = k - 1$,
opt = min.

We now demonstrate how to map an instance of SHORTEST PATH WITH FORBIDDEN PAIRS to an instance of MINIMUM CONFIGURATION. We name this mapping $t_1$. Given a graph $G = (V_{sp}, E_{sp})$, vertices $S \in V_{sp}$, $F \in V_{sp}$ and a set of forbidden pairs $P$, we create a configuration graph $K = (V_{mc}, E_{mc})$ by applying the following procedure:

**Specification of $\mathbf{t_1}$.** For each vertex $A$ in $V_{sp}$ let $in(A)$ be the number of incoming edges (indegree) and let $out(A)$ be the number of outgoing edges (outdegree). We name the incoming edges of $A$ $A^i_{in}, i = 1..in(A)$ and the outgoing edges $A^i_{out}, i = 1..out(A)$. Note that an edge in $G$ from some vertex $A$ to some other vertex $B$ is denoted by two aliases: $A^i_{out}$ and $B^j_{in}$ for some $i$ and $j$. For each vertex $A$, except $S$ and $F$, we define $p(A)$ to contain $in(A) * out(A) + 1$ vertices that are named $a_{ij}, i = 1..in(A), j = 1..out(A)$ and $a_{00}$. We define $p(S)$ to contain $in(S)$ vertices $s_i, i = 1..in(S)$ and $p(F)$ to contain $out(F)$ vertices $f_i, i = 1..out(F)$. For all $A \in V_{sp} : p(A)$ is a partition of the MINIMUM CONFIGURATION problem instance and we define $V_{mc} := \bigcup_{A \in V_{sp}} p(A)$.

We now define when to add edges between vertices in $V_{mc}$. Let $A \in V_{sp}$ with $a_{ij} \in p(A)$ and $B \in V_{sp}$ with $b_{kl} \in p(B)$. If $(A, B) \notin E_{sp}$, then there is no edge between $a_{ij}$ and $b_{kl}$. If $(A, B) \in E_{sp}$ we assume that it leads from $A$ to $B$ and that it is the $n$-th outgoing edge of $A$ and the $m$-th incoming edge of $B$. Then there is an edge between $a_{ij}$ and $b_{kl}$ with weight $w$, where $w$ depends on $i, j, k, l$ and can be determined from Table 1.

**Table 1.** Weight assignment to edge $(a_{ij}, b_{kl})$

| | $kl = 00$ | $k = m$ | $k \neq m$ |
|---|---|---|---|
| $ij = 00$ | 0 | $\infty$ | 0 |
| $j = n$ | $\infty$ | 1 | $\infty$ |
| $j \neq n$ | 0 | $\infty$ | 0 |

For each forbidden pair we add additional edges to $K$. Let $(A, B) \in P$ be a forbidden pair. For all $a_{ij} \in p(A)$ and $b_{kl} \in p(B)$, we add an edge with weight $w$, where $w$ depends on $i, j, k, l$ and can be determined from Table 2.

We now define the mapping $t_2$ from solutions of MINIMUM CONFIGURATION to a subset $G'$ of $G$. Remember that a solution for MINIMUM CONFIGURATION selects a vertex from each partition of $V_{mc}$. Due to the construction of $K$, each partition of $V_{mc}$ corresponds to a vertex in $V_{sp}$. Thus a solution for MINIMUM

**Table 2.** Weight assignment to edge $(a_{ij}, b_{kl})$ to express forbidden pairs

| | $kl = 00$ | $kl \neq 00$ |
|---|---|---|
| $ij = 00$ | 0 | 0 |
| $ij \neq 00$ | 0 | $\infty$ |

CONFIGURATION defines a mapping $k : V_{sp} \mapsto V_{mc}$. If $k(A) = a$, we say $A$ is *instantiated* to $a$.

**Specification of $\mathbf{t_2}$.** The following statements specify $t_2$:

- $\forall A \in V_{sp} \setminus \{S, F\} : k(A) = a_{00} \Rightarrow A \notin G'$
- $\forall A \in V_{sp} \setminus \{S, F\} : k(A) = a_{ij} \wedge ij \neq 00 \Rightarrow A \in G' \wedge A^i_{in} \in G' \wedge A^j_{out} \in G'$
- $S \in G' \wedge F \in G'$
- $k(S) = s_i \Rightarrow S^i_{out} \in G'$
- $k(F) = f_i \Rightarrow F^i_{in} \in G'$

The following Lemma states that $G'$ is a valid solution for the original instance of SHORTEST PATH WITH FORBIDDEN PAIRS.

**Lemma 6.** *For all instances $x$ of* SHORTEST PATH WITH FORBIDDEN PAIRS *the following holds: if $y$ is a solution to $t_1(x)$, then $t_2(y)$ is a solution to $x$, such that $m(y) = m(t_2(y))$.*

*Proof.* We have to show that $t_2$ maps a valid configuration for $K$ to a path $G'$ from $S$ to $F$, that respects all forbidden pairs and that the cost of the configuration is equal to the length of that path.

The second statement in the specification of $t_2$ (see above) defines edges $A^i_{in}$ and $A^j_{out}$ to be in $G'$ if $k(A) = a_{ij}, ij \neq 00$. We say the instantiation of $A$ *authorizes* the edges $A^i_{in}$ and $A^j_{out}$. As all edges in $V_{sp}$ are named by two different aliases (see definition of $t_1$) there are two vertices (the two end vertices) whose instantiation may authorize an edge. Although from definition of $t_2$ it is possible that an edge is authorized by one of its end vertices only, the weight assignment to the edges in the configuration graph (Table 1) ensures, that all edges in $G'$ are consistently authorized by both of its end vertices. To prove this consider an edge $e = (A, B)$, with alisas $B^m_{in}$ and $A^n_{out}$. If $e$ is authorized by $A$ or $B$ then it holds that $k(A) = a_{ij}$ with $j = n$ or $k(B) = b_{kl}$ with $k = m$. As a valid configuration must not include edges with infinite weight, it follows from Table 1 that both must be true, that $e$ is authorized by $A$ and $B$. Additionally we see that for each $e$ in $G'$ there is an edge in $K$ ($(a_{ij}, b_{kl})$ in this case) that has a weight of 1 and that for all one-weighted edges in $K$ there is an edge $e$ in $G'$.

As all edges are authorized by both end vertices and each vertex in $G' \setminus \{S, F\}$ authorizes one of its incoming and one of its outgoing edges, it follows that for each vertex $A \in G' \setminus \{S, F\}$ there is exactly one edge in $G'$ that leads to $A$ and exactly one edge that leaves $A$. $S$ authorizes exactly one of its outgoing and $F$

authorizes exactly one of its incoming edges. It follows that $G'$ is a simple path from $S$ to $F$. The one-to-one correspondence of one-weighted edges in $K$ and edges in $G'$ implies that its lenght is equal to the weight of the configuration for $K$.

It remains to show that $G'$ respects all forbidden pairs: $G'$ results from a valid configuration. Suppose $G'$ includes two vertices $A$ and $B$, such that $(A, B)$ is a forbidden pair. Then neither $a_{00}$ nor $b_{00}$ is part of the configuration. Because of this the configuration must include an edge with infinite weight (see Table 2), which is a contradiction to its validity. □

**Theorem 7.** *1.* MINIMUM CONFIGURATION *cannot be approximated within $n^\varepsilon$ for some $\varepsilon > 0$, where $n$ is the size of the problem instance, provided that $P \neq NP$.*

*2. A restricted version of* MINIMUM CONFIGURATION*, where all non-infinity weights $w$ are less than some upper bound $b$ is NPO PB-complete.*

*Proof.* 1. Let $x$ be an instance of SHORTEST PATH WITH FORBIDDEN PAIRS and $y = t_1(x)$. The size of $y$ is bounded by some polynomial in the size of $x$. So there exists some $c > 0$ such that $|y| < |x|^c$. As SHORTEST PATH WITH FORBIDDEN PAIRS is NPO PB-complete it cannot be approximated within $|x|^{\varepsilon_1}$ for some $\varepsilon_1 > 0$, if $P \neq NP$ [9]. If we could approximate $y$ within $|y|^{\varepsilon_2}$ for all $\varepsilon_2 > 0$, then we could approximate $x$ within $|x|^{\varepsilon_1}$ by using the linear reduction defined by $t_1$ and $t_2$ and by approximating $y$ within $|y|^{\varepsilon_1/c}$.

2. The restricted version of MINIMUM CONFIGURATION is included in NPO PB. As SHORTEST PATH WITH FORBIDDEN PAIRS is NPO PB-complete the NPO PB-completeness of restricted MINIMUM CONFIGURATION follows immediately from the reduction defined by $t_1$ and $t_2$. □

**Corollary 8.** MINIMUM CONFIGURATION *is not in APX.*

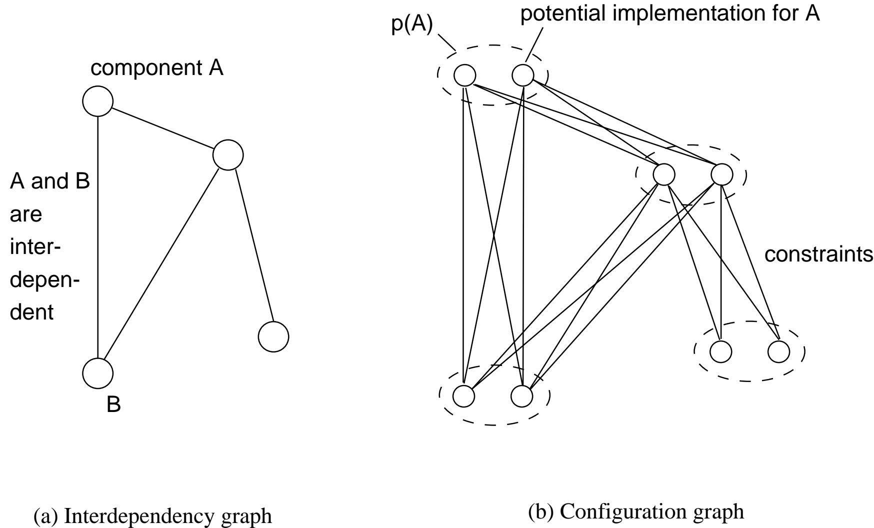
The corollary follows immediately from Theorem 7 and the definition of APX.

## 5 System Configuration

In this section we show how to apply the concept of configuration graph and MINIMUM CONFIGURATION to system configuration. Additionally we propose a way to identify subclasses of systems, that can be configured efficiently.

### 5.1 Construction of the Configuration Graph

We model the system to be configured as a graph $G = (V, E)$, where $V$ is a set of *components*. Components are atomic with respect to configuration and represent a part of the system's functionality. The edges of $G$ represent interdependencies between components that affect the configuration of the system. We name $G$ the *interdependency graph* of the system. We do not show how to derive the interdependency graph from the original system specification, as this is beyond




(a) Interdependency graph

(b) Configuration graph

**Fig. 1.** Interdependency graph and corresponding configuration graph

the scope of this paper. For the purpose of this paper the interdependency graph is the starting point of configuration.

We assume that for each component $A \in V$ we are given a set of implementations that might substitute $A$ during the process of refinement. The set of potential implementations for a component $A$ is denoted by the function $p(A)$,$p : V \mapsto \mathbb{P}^R$, that maps components of $G$ to subsets of the universe $R$ of implementations. Each implementation is associated with some cost, which is described by a function $cr : R \mapsto \mathbb{N} \cup \{\infty\}$. The employment of combinations of implementations might be subject to additional requirements. These requirements are called *constraints* and are also associated with cost, described by a function $cc : R \times R \mapsto \mathbb{N} \cup \{\infty\}$. Given an interdependency graph $G_D = (V_D, E_D)$ of a system, we construct a configuration graph $G_C = (V_C, E_C)$ in the following way:

**Construction of $\mathbf{G_C}$.** We define $V_C := \bigcup_{A \in V_D} p(A)$ and regard each $p(A)$ to be a partition of $V_D$. Consider vertices $A$ and $B$ of $G_D$ and let $a \in p(A)$ and $b \in p(B)$. We add an edge $(a, b)$ to $G_C$ if and only if $(A, B) \in E_D$. If $(a, b) \in E_C$ we define the weight of $(a, b)$: $w(a, b) := cc(a, b)$. As a temporary extension to Minimum Configuration we weight the vertices of $G_C$ with $w(a) := cr(a), a \in V_C$. We later show that this extension is equivalent to the original problem by demonstrating how to transform vertex weights into edge weights.

A solution to Minimum Configuration for $G_C$ selects a vertex from each partition of $G_C$, thus defining a function $k : V_D \mapsto V_C$. Refining each component

$A \in V_D$ by $k(A)$ yields a system that is optimal with respect to the cost described by $cc$ and $cr$. Thus $k$ defines the optimal configuration for the system modelled by $G_D$.

Fig. 1 shows the interdependency graph of an imaginary system with four components, each with two possible implementations, and the corresponding configuration graph. As can be seen in the picture, the interdependency graph provides some sort of abstraction from the configuration graph. The function $p$ maps each vertex $A$ of $G_D$ to a partition of vertices in $G_C$. Additionally we can define a function $p' : E_D \mapsto \mathbb{P}^{E_C}, p'((A,B)) = \{(a,b) | a \in p(A) \wedge b \in p(B)\}$, that maps each edge in $G_D$ to a partition of edges in $G_C$.

*Removal of vertex weights* Now we refer back to the construction of the configuration graph and show that the vertex weights can be transformed to edge weights without affecting the result of configuration. Let $G_D = (V_D, E_D)$ be the interdependency graph and $G_C = (V_C, E_C)$ the corresponding configuration graph. Remember that $p$ and $p'$ map vertices and edges in $G_D$ to partitions of vertices and edges in $G_C$ (see above).

**Construction of $\mathbf{G_C}$ (continued).** Consider $A \in V_D$ and $a \in p(A)$ with weight $w(a) = cr(a)$. To transform the weight of $a$ into edge weights we choose exactly one edge $e$ from the edges that are adjacent to $A$. We define $\pi(a) := \{(v_1, v_2) \in p'(e) | v_1 = a \vee v_2 = a\}$. $\pi(a)$ are the edges in $p'(e)$ that are adjacent to $a$. We redefine the weight of all edges $(v_1, v_2) \in \pi(a)$: $w(v_1, v_2) := cc(v_1, v_2) + w(a)$ and eliminate the weight of $a$.

**Proposition.** *The presented transformation of vertex weights to edge weights does not affect the result of configuration.*

*Proof.* Given a configuration for $G_C$ that induces a subgraph $G'$. The following holds:

- $a$ is part of $G' \Rightarrow$ exactly one edge from $\pi(a)$ is part of $G'$
- $a$ is not part of $G' \Rightarrow$ no edge from $\pi(a)$ is part of $G'$

From this it follows that $w(a)$ is added if and only if $a$ is part of $G'$. So the weight of $G'$ is not affected by the transformation. □

### 5.2 Identification of Subclasses of Systems

We now demonstrate, that the interdependency graph is not only the starting point for configuration, but also has another important property. Because of the results of Sect. 3 and 4, we cannot expect to find good configurations (with bounded error) for all kinds of systems in an efficient way. The abstraction provided by the interdepenency graph provides a means to identify subclasses of systems for which that can be done. These subclasses will be identified by structural properties of the interdependency graph.

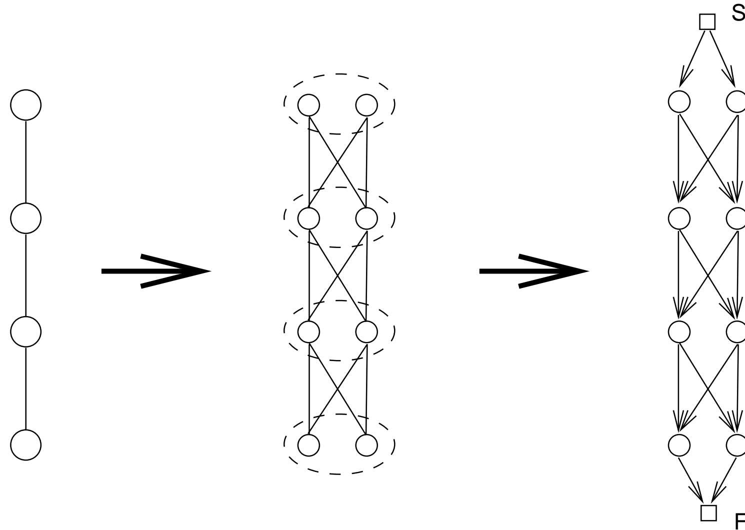



**Fig. 2.** Creation of a DAG from a special interdependency graph

An example of such a special subclass is the class of systems for which the interdependency graph is a sequence of components, that is a tree with two leaves. Every configuration graph that can be derived from such a interdependency graph is stratified and can easily be transformed into a topologically sorted DAG with a single start vertex $S$ and end vertex $F$ (see Fig. 2). Solving Minimum Configuration for this type of configuration graph is equivalent to finding the shortest path from $S$ to $F$ in the corresponding DAG. This can be done in $O(|E_C|)$ time [4].

The algorithm for finding the shortest path in a topologically sorted DAG is based on dynamic programming. It should also be possible to apply this principle to solve Minimum Configuration in polynomial time for configuration graphs that are derived from trees in general. We are currently looking for other properties of interdependency graphs that ensures, that for configuration graphs, derived from these graphs, Minimum Configuration can be solved in polynomial time or can be approximated (at least) within some constant.

In the next section we present some examples for applications that benefit from configuration.

## 6 Applications

In this section we present some applications of configuration and explain how to map the examples to the concepts presented in this paper. We concentrate on examples concerning the runtime of sequential programs. Additionally we give

one example for applications with more complex structural interdependencies and different kinds of cost.

*Programming with ADTs:* Consider a program that uses several abstract data types. We regard each use of an ADT operation and the data of the ADT as a component of the system to be configured. The functionality and the semantics of each component is fully specified but the implementations can be chosen arbitrarily in a later refinement process. The implementations are supposed to exist and can be selected from a given library of algorithms and data structures, e.g. [11, 7] . It is obvious that the selection of concrete implementations for the data structures and the operations can have a significant effect on the runtime of the final program. Independent selection of the cheapest (fastest) implementation for each component is not a solution, because this may produce conflicts in the refinement of the data that is cooperatively used by different operations. A very simple example is the insertion and deletion of elements in/from a sorted set (a priority queue). The fastest solution for the insert operation is the use of an unsorted string of data, whereas the fastest solution for the delete operation is the use of a sorted representation. To make the right choice it is necessary to find the trade-off between several solutions, including those that use both representations and perform a representation change (sorting) at some places in the program. Which of these solutions is minimal with respect to the runtime depends on the type of operations that are to be performed, where and how often in the program they are performed and on the initial size of the set. The use of different data representations for subsequent operations require the transformation of the data. These transformations become the constraints between the implementations. The cost of implementations and constraints is set to the expected runtime of operations and transformations. The form of the resulting interdependency graph depends on the actual program. In simple cases however, it will be a sequence of components. In this example configuration selects the best implementation for each component in the system, such that the expected overall runtime is minimized.

Other examples may be constructed just by exchanging the scenario and what is considered the components and the constraints.

*Programming with Databases:* Here components are query-operations on an (object-oriented) federated database, that are initiated by an application program. The requirements for using a certain implementation of an operation might be the migration of some data to the local host or the construction of a special index structure. Construction of these data structures from the already existing ones is mapped to constraints. Execution time is mapped to cost.

With configuration it is possible to include knowledge of the application into the optimization of query processing, which is not possible with just using classic query optimization techniques.

*Configurable Hardware:* In a third example we consider configurable hardware, e.g. an external FPGA board, that is used to accelerate parts of a program

[2, 13]. In this example we have hardware and software implementations for each component. The time to upload the data to the external board or the time to reconfigure the hardware is mapped to constraints.

*Software Design:* Our last example addresses the design process itself. The software architecture of a system is a network of operators[4], each expressing a certain part of the system's functionality and connectors, that support cooperation of the operators [1]. The refinement of the architecture includes the choice between different design alternatives for the operators and the connectors. The variants for the refinement are typically not fully realized by existing building blocks. A cost measure that, besides the already mentioned ones, is a target for minimization, is the estimated development time for the system. The time to implement different parts of the system from scratch, the time to adapt preexisting building blocks to the rest of the system and the degree of interleaving during the development of different parts of the system are mapped to cost of implementations and constraints. Other possible measures are the use of I/O bandwith or the degree of reliability. An especially interesting but of course very ambitious application in this context is the handling of combinations of these cost measures.

We conclude this section with a final remark concerning the use of infinite weights. When introducing MINIMUM CONFIGURATION, we allowed the definition of infinite weighted edges (Def. 1). Infinite weight is used to model combinations of implementations for different components that are impossible to construct. For the purpose of theoretical proof we excluded configurations with infinite weighted edges from the set of valid solutions, thus ensuring that the objective function is finite. For implementations that try to find a good configuration for a given system it will mostly be sufficient to replace infinite weight by a sufficiently large value[5], such that no solution that includes impossible combinations is stated to be the optimum unless there is no other solution.

## 7 Conclusion

In this paper we presented a new optimization problem, MINIMUM CONFIGURATION, that has important applications to software construction. A formal definition of the problem was presented, based on the employment of graphs for the description of the problem instance. We proved MINIMUM CONFIGURATION to be NP-hard and hard to approximate by constructing reductions from graph coloring and SHORTEST PATH WITH FORBIDDEN PAIRS. We introduced the notion of configuration graph and demonstrated how to employ the new concepts to system configuration. A means of abstraction called interdependency graph was defined. We demonstrated that this abstraction is useful for the identification of special subclasses of systems for which configuration can be done

[4] These are usually called components. We use the term operators to avoid ambiguity.
[5] This could be a multiple of the sum of all finite weight values that have to be taken into account during configuration.

efficiently. Examples from the application domain showed how a concrete configuration problem is mapped to the concepts introduced in this paper. Being a sufficient platform for the formal definition of the problem and for demonstrating the applicability we expect that the concepts introduced in this paper act as a theoretical foundation for future work in this area.

Our future work on this problem includes the identification of further classes of systems, characterized by their interdependency graph, for which configuration can be done efficiently. Another important aspect is the use of functions and vectors of different measures for the description of cost, which will extend applicability of the presented concepts.

## References


1. Special Issue on Software Architecture, IEEE Transactions on Software Engineering, April 1995.
2. Peter M. Athanas and Harvey F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3), March 1993.
3. Gianluigi Caldiera and Victor R. Basili. Identifying and qualifying reusable software components. *Computer*, February 1991.
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, Massachusetts, 1990.
5. P. Crescenzi and A. Panconesi. Completeness in approximation classes. *Information and Computation*, 93(2):241–261, 1991.
6. Simon Even. *Graph Algorithms*. Computer Software Engineering Series. Computer Science Press, Rockville Maryland, 1979.
7. A. Frick, W. Zimmer, and W. Zimmermann. On the design of reliable libraries. In *TOOLS 17 – Technology of Object-Oriented Programming*, pages 13–23. Prentice Hall, 1995.
8. H.N. Gabow, S.N. Maheshwari, and L.J. Osterweil. On two problems in the generation of program test paths. *IEEE Transactions on Softw. Eng.*, 2(3), September 1976.
9. V. Kann. Polynomially bounded minimization problems that are hard to approximate. *Nordic J. Comput*, 1:317–331, 1994.
10. Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2), June 1992.
11. Kurt Mehlhorn and Stefan Näher. Leda - a platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1), January 1995.
12. Ruben Prieto-Diaz and James M. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4), November 1986.
13. Markus Weinhardt. Integer programming for partitioning in software oriented codesign. In *Field-Programmable Logic and Applications; 5th International Workshop*, volume 975 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.


This article was processed using the LaTeX macro package with LLNCS style