ELSEVIER Journal of Mathematical Economics 28 (1997) 341–373

JOURNAL OF Mathematical ECONOMICS

Accelerating new product development by overcoming complexity constraints

Spyros Vassilakis

Instituto Universitario Europeo, C.P. No. 2330, I-50100 Firenze Ferrovia, Italy

Submitted 1 March 1996; accepted 1 September 1996

Abstract

The economic theory of technological change is a theory of investment. Agents invest in an activity called research; a black box process returns a value for a variable that shifts the production function. This paper proposes a way to open the black box. It is motivated both by theoretical arguments for opening this box and by recent empirical literature on new product development that stresses the importance of nontraditional factors in determining firm performance. An example of the former is Solow: "... the production of new technology may not be a simple matter of inputs and outputs. I do not doubt that high financial returns to successful innovation will divert resources into R & D. The hard part is to model what happens then". An example of the latter is Baily and Gersbach, who discuss the determinants of operating efficiency of firms: "Traditional determinants, such as capital intensity and scale were found to play a role. But innovations such as design for manufacturing and workplace organization turned out to be even more important".

This paper considers technological change, and in particular new product and process development, from a new perspective, that of complex problem solving. Technological change is made possible by organizational techniques that reduce the complexity of problem solving; differences in the technological performance of firms are attributed to their different problem-solving styles. More specifically, the paper identifies rework as the main reason and symptom of slow NPD, and proposes three techniques that provably minimize rework: problem decomposition, decision ordering, and communication before design.

PII S0304-4068(97)00803-8

JEL classification: C69; D20; L23; M11; O32

Keywords: New product development; Organization and management of technological innovation; Management of complexity

1. Introduction

The economic theory of technological change is a theory of investment. Agents invest in an activity called research; a black-box process returns a value for a variable that shifts the production function. Economic analysis focuses on the appropriability of the returns to investment in research, i.e. on reasons why the social returns to investment in research might differ from private returns, and on policy measures designed to bring them closer together. It is clear from the literature that more is involved in the determination of technological change. Solow (1994, p. 52) states: "... the 'production' of new technology may not be a simple matter of inputs and outputs. I do not doubt that high financial returns to successful innovation will divert resources into R & D. The hard part is to model what happens then"; and "... I think the best candidate for a research agenda right now would be an attempt to extract a few workable hypotheses from the variegated mass of case studies, business histories, interviews, expert testimony, anything that might throw light on good ways to model the flow of productivity-increasing innovations and improvements". New product development (NPD) is a good place to start taking Solow's advice. There is a growing empirical literature on it that does offer a few workable hypotheses to start with. The mathematical model introduced in this paper is directly motivated by it; I will spend some time describing its main points.

The first point is that most NPD involves application of already existing, widespread knowledge, not a scientific breakthrough or any significant new learning. Gomory (1989) states: "There is another process of innovation, which is far more critical to commercializing technology profitably. Its hallmark is incremental improvement, not breakthrough. It requires turning products over again and again; getting the new model out, starting work on an even newer one. There is no brand new product here, no revolutionary technology.... It is competition among ordinary engineers in bringing established products to market". I will concentrate on such routine NPD and I will call 'mature' the industries with a well-established stock of relevant scientific and technological principles known to all firms in the industry. The automobile industry is an example.

The second point is that in mature industries NPD cannot deliver long-lasting monopoly positions. The critical performance variables are, instead, the cost, variety, quality and time-to-market of the new products. The ability to develop new products faster than competitors do, in particular, has a number of strategic advantages. Demand and technology change in unpredictable ways; fast NPD reduces the risk of coming up with a product no longer in demand. Note that the

strategic advantage of fast NPD does not depend on customers' preferences for wide variety or frequent model upgrades. Womack *et al.* (1989, p. 127) make the point. "The producers in full command of these techniques can use the same development budget to offer a wider range of products or shorter model cycles — or they can spend the money they save by implementing an efficient development process for developing new technologies. And in every case, the shorter development cycle will make them more responsive to sudden changes in demand. The choice, and the advantage, will always lie with lean producers."

The third point is that some firms seem to be systematically faster in developing new products; and that their speed advantage cannot be reasonably attributed to extra effort, better engineers, more computing power, less complex products, or chance. In fact, the most surprising result of the Clark and Fujimoto (1991) and Womack *et al.* (1990, Ch. 5) studies was that the faster product developers were also those using significantly *less* engineering effort, even after adjusting for differences in the products developed. The companies studied were all world-class, with open access to the best available resources. In fact, Clausing (1994, p. 5) emphasizes that there are no significant differences in the quality of engineers in the countries covered by the studies (US, Japan, Germany). The fastest product developers (Honda, Toyota) have since demonstrated, by designing products outside their home markets, that their speed advantage is not related to regional differences; and wide firm differences within each region (Toyota and Nissan are frequently cited) have reinforced this point. Furthermore, the speed advantage of the fastest product developers was not achieved by sacrificing low production cost or high product quality. In fact, Honda and Toyota are pioneers of quality function deployment (QFD), design for manufacturing, and robust design techniques that accelerate the design of new products, taking into account quality and cost targets. Finally, faster product developers were not merely luckier product developers. They have repeated their success across time and across different markets. They use distinctly different NPD techniques; when these techniques were adopted by other firms (Clausing (1994, p. 47) cites Ford and Xerox as "early outstanding successes"), development time was drastically reduced.

The fourth point reinforces the third. There was a large difference in the initial condition, of Honda or Toyota on the one hand, and Ford and GM on the other. According to the investment/appropriability theories of technological change, their larger volume should have provided the Detroit firms with the advantages of scale economies and learning-by-doing; drawing from the same labor market should have provided the advantages of technological spillovers and of cross-pollination of ideas; recruiting from the top universities in the world should have provided the advantages of human capital; and outspending their rivals in R & D, and having a larger stock of knowledge to begin with, should have provided the advantages of accumulated knowledge capital. Despite these accumulated advantages, Detroit firms lost market share to firms (like Toyota and Honda) that were able to offer *simultaneously* lower cost, higher quality, more variety and faster

model replacement. Womack *et al.* (1989) document this story and provide the following example as an illustration of the fact that faster product development is not due to the 'usual suspects' of more knowledge or more capital. They describe how Toyota in the 1970s redesigned 4-cylinder engines to maximize their power, and in the process changed its image among consumers from a 'low-tech weakling' to a 'high-tech wonder'. They conclude (*ibid.*, p. 132): "This perception on the part of consumers was enormously frustrating to engineers in many of the mass-production companies who knew that all these 'innovations' had been around the motor industry for decades... What's more, when they tried to copy these 'innovations' on a wide scale, the weaknesses of their engineering systems were exposed. In many cases it took years to introduce a comparable feature, and often it was accompanied by drivability problems or high production costs. GM, for example, lagged Toyota by four years in introducing many of the features we just listed in its Quad Four engine; it needed two more years to reach a high level of refinement." (Womack *et al.*, 1989, p. 116).

The fifth point is that NPD is a problem-solving activity. In particular, NPD is not merely a special case of the theory of investment under uncertainty. Wheelwright and Clark (1992, p. 218) state "... in the final analysis, when we search for an understanding of truly outstanding development, we must eventually get down to the working level where individual designers, marketers and engineers work together to make detailed decisions and solve specific problems. The magic of an outstanding product is in the details. Thus, detailed problem solving is at the core of outstanding development."

The sixth point is that NPD is a *complex* problem-solving activity. In Clausing's (1994, p. 57) words, "in developing a complex product, there may be 10 million decisions. Although individuals can make most decisions, the most critical decisions (roughly 1000 to 10000 for large, complex products) require more attention, and most of them do not lie entirely within the experience of any individual or group." Ulrich and Eppinger (1995, p. 4) state that "... very few products can be developed in less than a year, many require three to five years, and some take as long as ten years." Developing the Chrysler Concorde car, for example, requires the design of 10000 unique (to this car) parts, 2250 people, 3.5 years and one billion dollars (*ibid.*, p. 6). Clark and Fujimoto (1991, p. 73) find that, on average, a new car design takes 2.5 million engineering hours, spent over a period of 54 months.

The seventh point is that fast and slow developers differ in their approach to solving complex problems; and that this shows up in the amount of rework (iteration, backtracking, design revision) they generate. Slow developers divide an NPD problem into sub-problems functionally, i.e. using the technical nature of the decisions as a dividing criterion. For example, all customer-related decisions like styling and features are made by marketing; all product engineering-related decisions are made by design; and all process engineering decisions are made by manufacturing. Slow developers typically order their decisions sequentially. Tech-

nology-driven firms start with the decisions that product a technically feasible, functioning product, and worry about manufacturability and customer appeal later; styling-driven firms start with the decisions that produce an attractive product, and worry about technical feasibility later. Finally, slow developers limit communication between sub-problems to occur when a functional group delivers a complete design, or a request for a change in a complete design, to another such group; this is usually called "throwing the design over the wall" to another group. A typical result of such a process is that the first functioning design is too costly to make, so manufacturing returns the design to engineering for rework; or that the first functioning design does not appeal to customers, so marketing returns the design to engineering for rework. Changes to accommodate manufacturing are typically not consistent with those that please marketing, so further rework typically occurs. Fast product developers, on the contrary, use the degree of interaction among decisions, not membership in a skill category, as the criterion of assigning decisions into sub-problems. If a marketing decision like car height interacts with an engineering decision like engine power (and therefore size), then these decisions must belong to the same sub-problem, even if they belong to different functional specialties. Sub-problems are usually identified with teams, because each sub-problem is the responsibility of a dedicated team. In Clausing's (1994, p. 40) words, "Many critical interfaces have a dedicated team. Teams are formed wherever they are needed to achieve an integrated approach to the development of the new product. The formation of the best interlocking structure of teams is a key success factor." This is the subject of parts 5 and 6 of this paper.

Problem partitioning is made necessary by complexity; incorrectly done, it generates rework. Similarly, complexity dictates that interdependent decisions have to be made one at a time. The order of making decisions matters. Earlier decisions constrain later ones, thus reducing search effort; but earlier decisions could constrain later ones by eliminating the desirable options, thus generating rework. Two of the "cash drains of traditional product development" described by Clausing (1994, pp. 19–20) are examples of incorrect ordering of decisions. They are "technology push" (clever technology is developed that does not satisfy any significant customer need); and "disregard for the voice of the customer". Clausing (1994, p. 20) states: "The first step in the development of a specific new product is the determination of the customer's needs. In traditional product development the product has often been doomed to mediocrity before the completion of the needs activity. The biggest culprit has been the deployment of the voices of corporate specialists, rather than the voice of the customer." Fast developers, instead of starting with how to make a technically satisfactory, functioning product, start with making decisions on the values of the variables that are directly relevant to customers. This paper discusses the ordering of decisions in Section 4.

The need for communication in NPD is also dictated by complexity. Decisions made in different sub-problems have to be checked for consistency; and earlier

decisions have to be checked for consistency with later decisions. Lack of consistency implies revision of decisions already made, i.e. rework. Slow developers are characterized by infrequent communication between decision-makers, relatively late in the NPD process. Fast developers are characterized by a large amount of communication early in the NPD process. Womack *et al.* (1989, p. 115) state "In the best team projects, the numbers of people involved are highest at the very outset. As development proceeds, the number of people involved drops as some specialities are no longer needed. By contrast, in many mass-production design exercises, the number of people involved is very small at the outset but grows to a peak very close to the time of launch, as hundreds or even thousands of extra bodies are brought in to resolve problems that should have been cleared up in the beginning." Communication in slow NPD groups takes place *after* design decisions have been made on a large part of the product; these decisions are "thrown over the wall" to a group responsible for another set of decisions. Communication in fast NPD groups takes place *before*, as well as after, design decisions have been made. Communication before decision-making does not intend to render already taken decisions consistent; but to render more concrete the goals of NPD by eliminating values of variables that are not consistent with any values of other variables, and thus avoid rework in the future. For example, the goal of achieving durability of a part may be achieved by gold-plating it; if this is consistent with none of the allowed values of the total cost variable, the gold-plating option can be eliminated before design begins: gold-plating could never be part of an acceptable design. This paper discusses communication in Section 4.

2. Design problems as constraint satisfaction problems

To define the design problem mathematically, I will first borrow a description of the design problem from the literature on 'quality function deployment'; see Hauser and Clausing (1988), Wheelwright and Clark (1992, p. 229), and Clausing (1994). Suppose that the design task is the development of a gear system for the automatic film rewinder of a camera. The consumer of a camera cares about some of its attributes: speed of rewinding, sound, reliability, cost, size, ... Each of these attributes can be represented by a variable that takes values in some finite domain. The values of the variables are restricted by consistency constraints: for example, a camera of some size and reliability cannot cost less than some amount. To deliver the attributes valued by the customer, engineers have to decide on a number of engineering attributes: number of gears, diameter of gears, number of teeth per gear, ... Engineering attributes can also be represented by variables that take values in some finite domain. There are also consistency constraints between engineering and customer attributes; for example, two gears of a certain size and tooth profile cannot generate a sound lower than some level. Finally, there are consistency constraints between the engineering attributes themselves: for exam-

ple, the weight and size of gears are related. In a similar way, one can link engineering attributes to production process attributes, and therefore to cost, quality and variety attributes. Finally, the objectives of design can also be represented by a relation (a constraint) between attributes; for example, deliver a rewind system that costs less than a certain amount, rewinds in less than a certain amount of time, and will not break down before it is used a certain number of times.

The considerations of Section 1 motivate the study of the following problem.

Definition 2.1. A constraint satisfaction problem (CSP) is a tuple $\langle X, D, R, s, e\rangle$
- $X = \{x_1, \ldots x_n\}$ is a finite set of variables;
- $D = \langle D_1 \ldots D_n\rangle$ is a tuple of finite sets;

D_i is the domain of variable x_i.
- $\boldsymbol{R}$ is a tuple of relations (constraints). Each relation $c \in R$ is defined by its scope $s(c) \subseteq X$, i.e. the variables it involves, and its extent $e(c) \subseteq \prod_{i \in s(c)} D_i$, i.e. the tuples that satisfy c. Products respect the order of variables, i.e. $\prod_{i \in \{3,2\}} D_i$ equals $D_2 \times D_3$. This involves no loss of generality.

Definition 2.2. A solution of a CSP $\langle X, D, R, s, e\rangle$ is a tuple $u = \langle u_1 \ldots u_n\rangle$ such that
- $u_i \in D_i$ for all $i = 1 \ldots, n$
- for each constraint $c \in R$ with scope $s(c) = \{x_{i1} \ldots x_{ik}\}$, we have $(u_{i1}, \ldots, u_{ik}) \in e(c)$, i.e. all constraints are satisfied by u.

This formulation of the product design problem is sufficiently expressive to capture not only the basics, but also design for manufacturing, and robust design. Design for manufacturing takes into account manufacturing cost at the time the product is designed; seemingly minor design details can have a major impact on cost. For example, there is usually some room to vary the number of components that make up a product, and a number of options as to how the components will be attached to each other. More components imply less fabrication cost but greater assembly cost; using screws to attach components to each other increases assembly time, but facilitates product disassembly for repair and maintenance. These concerns can be captured in a CSP by introducing extra variables for the number of components of the product, and for the method of attachment; and extra constraints to capture the relations between the values of these extra variables and other variables appearing in the CSP (e.g. cost). Design for manufacturing is discussed in Ulrich and Eppinger (1995, Ch. 9) and their references. Robust design takes into account the fact that product performance is conditional on the values of variables not controlled by the customer; it aims to deliver the same product performance over a wide range of the values of these variables, i.e. to deliver "robust quality" (Taguchi and Clausing, 1990). This increases the value of the product to the customer and, according to Clausing (1994, p. 74), "robustness also

greatly shortens development time by eliminating much of the rework that is known as build, test and fix." An example of the kind of variables that affect product performance but are not under customer control is given by Clausing (1994, p. 75) "A lemon is a car that has excessive production variations. To overcome this, the production process has to be robust so that they produce less variation, and the car design has to be more robust so that its performance is less sensitive to production variations. The customers also want a car that will start readily in winter and not overheat during the summer; that is, they want a car that is robust with respect to the variations of customer use conditions." Robustness considerations can be expressed by a CSP if each variable is replaced by a set of variables, each of which expresses the same attribute in a different state of nature. Robustness is imposed by requiring that all these variables take the same value, i.e. by extra constraints. Still more constraints can express relations of attributes in different states of nature.

The finiteness of the domains of a CSP ensures that we can always decide whether a given CSP has a solution or not. The method is called generate and test (GT): for each tuple u in $\prod D_i$, test if each constraint c in R is satisfied. If yes, declare a solution. If no, generate another tuple u and test it. There are only finitely many tuples in $\prod D_i$, so GT will eventually terminate. GT is obviously grossly inefficient, and we cannot expect to solve any CSP of reasonable size using it. Formally, the number of steps GT takes to compute a solution to a CSP is an exponential function of the number of variables in the problem (Mackworth, 1992, p. 6). Perhaps surprisingly, it is not known whether there exists or not an algorithm that solves the CSP in a number of steps that is a polynomial function of the number of variables ('in polynomial time'). What is known is that if such an algorithm exists, then many other combinatorial problems will have such an algorithm (CSP is NP-complete). Such problems are widely believed to be intractable, i.e. not solvable by an algorithm in polynomial time. For solving such problems, therefore, it is imperative to have good structuring techniques (problem-partitioning and solution-integration techniques) that reduce the complexity of problem solving, search techniques more efficient than generate-and-test, and 'decision ordering' and 'communication' techniques that reduce the amount of rework.

3. Rework

Generate-and-test is inefficient because it first generates a complete vector of values $u_1, \ldots, u_n$ and then checks for consistency. If inconsistency between, say, u_2 and u_3 could be detected before u_4 is generated, then we would not need to check the whole vector for consistency (nor expand it beyond u_4). This idea is captured by depth-first search of the search-tree generated by the CSP in question.

Definition 3.1. Let $\langle X, D, R, s, e\rangle$ be a CSP. Order $X = \{x_1 \ldots x_n\}$ and each domain D_i in an arbitrary way. The search tree generated by $\langle X, D, R, s, e\rangle$ is defined to be a rooted tree; the children of the root are the elements of D_1 in ascending order; they are called first-level nodes. The children of any ith level node, $1 \leq i \leq n-1$, are the elements of D_{i+1} in ascending order. Level n nodes are leaves, i.e. have no children. A path from the root to a leaf is a complete assignment (of values to variables). A path from the root to an internal node at level i is a partial assignment up to level i.

Definition 3.2. A partial assignment up to level i is consistent if it satisfies all the constraints whose scope is a subset of $\{x_1, x_2, \ldots, x_i\}$.

Assignments, partial or complete, can be thought of as analytical prototypes. Ulrich and Eppinger (1995, p. 219) define a prototype "as an approximation of the product along one or more dimensions of interest. Analytical prototypes represent the product in a non tangible, usually mathematical, manner". The backtracking (BT) algorithm builds an analytical prototype gradually, by successively assigning values to variables and checking for consistency.

When no variable has been assigned a value, the algorithm is at level 0 (at the root of the search tree). The algorithm starts by assigning to x_1, the first variable in X, its first value; it is then at level 1. Suppose that the algorithm has built up a partial assignment $u_1 u_2 \ldots u_k$ up to level k, $1 \leq k \leq n$. If $k = n$, the algorithm terminates with success. If $k < n$, the algorithm picks the first value u_{k+1} in D_{k+1} consistent with $u_1 \ldots u_k$; and extends the assignment by setting $x_{k+1} = u_{k+1}$. If no such value of x_{k+1} exists, the algorithm destroys the assignment $x_k = u_k$, i.e. it backtracks to level $k-1$; and creates the assignment $x_k =$ successor of u_k, thus returning to level k. If u_k has no successor (because it is the last element of the domain of x_k), the algorithm backtracks to level $k-2$ by destroying the assignment $x_{k-1} = u_{k-1}$ as well; it then returns to level $k-1$ by setting $x_{k-1} =$ successor of u_{k-1}; and it checks this assignment for consistency with values in D_k. If the algorithm backtracks to level zero, and u_1 has no successors, then the algorithm terminates with failure.

The BT algorithm will always terminate, since it traverses a finite search tree (depth-first). It is sound, in the sense that if it terminates with success, then the complete assignment associated with successful termination is in fact a solution of the CSP by construction; and if it terminates with failure, the CSP has in fact no solution, since all branches of the search tree have been examined without finding a solution. It is complete in the sense that if the CSP has a solution, the algorithm will terminate with success; and if the CSP has no solution, the algorithm will terminate with failure. These properties, together with its superiority over the generate-and-test algorithm, motivate its use.

The BT algorithm generates rework every time it backtracks; a previously made decision is changed, and additional constraint checks take place. What is the

importance of rework in determining development time? To see this, let $r = |R|$ be the number of constraints, and $K = \max_i |D_i|$ the number of values in the largest domain. A solution obtained without rework requires at most rKn constraint checks. (The BT algorithm in this case never encounters a non-extendable partial assignment. At each level it checks at most K values of x_{t+1} for consistency with the existing assignment up to level t. There are r constraints, so at each level at most rK checks take place. And there are n levels, so at most rKn checks take place overall.) A solution in general, however, can require up to rK^n constraint checks. This is because, in the worst possible case, all K^n possible complete assignments will have to submit to at most r consistency checks each. The difference $rK^n - rKn$ is due entirely to rework, and is huge even for moderate values of n; for $r = 1$, $K = 2$, $n = 54$, and assuming a million constraint checks per second, the difference is a thousand years. Minimization of rework is thus practically equivalent to minimization of total development time. The three techniques introduced later on reduce the 'effective' K, n and r respectively. The communication technique reduces the size of the domains by eliminating options. The problem-partitioning technique reduces the number of variables. The decision-ordering technique reduces the number of constraint checks. I start with an important special case.

4. Binary, tree-structured constraint satisfaction problems

A CSP is binary if the scope of each constraint contains at most two variables. It is tree-structured if its primal graph (see Definition 4.1) is a tree, i.e. a connected acyclic graph. Recall hat graphs consist of nodes and lines between nodes, called edges. All graphs in the paper are undirected; i.e., if there is an edge from x to y, then there is also an edge from y to x. Paths in a graph consist of consecutive edges. A graph is connected if there is a path linking any two of its nodes. It is acyclic if there is no path starting from and ending at the same node.

Definition 4.1. The primal graph of a binary CSP $\langle X, D, R, s, e\rangle$ has X as its node set; and there is an edge (x, y) in the primal graph if and only if there is at least one constraint in R with scope $\{x, y\}$.

The main result in this section is that a decision-ordering and a communication technique applied to a binary, tree-structured CSP allow the BT algorithm to find a solution without rework. The two techniques themselves are easy to apply, in a well-defined sense. The next example illustrates the importance of a correct ordering of the variables. Recall that the primal graph, even if it happens to be a tree, is different from the search tree of the same problem.

Example 4.1. Consider the CSP problem with $X = \{x_1, x_2, x_3\}$, $D_i = \{0, 1\}$, $R = \{c_{12}, c_{23}\}$ $s(c_{ij}) = \{x_i, x_j\}$, $e(c_{12}) = \{00, 11\}$, $e(c_{23}) = \{01, 10\}$. Its primal

graph is a tree. Consider first what happens if the variables are searched in the order x_1–x_3–x_2. The BT algorithm first sets $x_1 = 0$, $x_3 = 0$ (there are no constraints linking x_1 to x_3). It then performs the constraint checks $c_{12}(0, 0)$ $c_{23}(0, 0)$ $c_{12}(0, 1)$, and discovers that $x_1 = 0$, $x_3 = 0$ is not consistent with any value of x_2. Hence it backtracks, sets $x_3 = 1$, and performs the constraint checks $c_{12}(0, 0)$, $c_{23}(0, 1)$, both successful. In the ordering x_1, x_2, x_3, however the assignment $x_1 = 0$ is immediately followed by the constraint check $c_{12}(0, 0)$, since x_1 and x_2 are linked by a constraint. This being successful, the BT algorithm performs the test $c_{23}(0, 0)$, which fails. It then tests $c_{23}(0, 1)$, which succeeds. Hence, in the ordering x_1–x_3–x_2 a solution was found with five constraint checks, while in the ordering x_1–x_2–x_3, with only three.

Note that in the ordering x_1–x_3–x_2, x_2 is linked by an edge to *two* of the variables preceding it, forcing two constraint checks of each extension of a partial assignment that requires checking. In the ordering x_1–x_2–x_3, however, each variable is linked by an edge to at most one variable preceding it, so that pruning of the search tree is more gradual and starts earlier than in the ordering x_1–x_3–x_2; in the latter, no constraint limits the (x_1, x_3) assignments, and all pruning takes place when all but one variable have been assigned values, thus negating the advantage of backtracking over generate-and test. This motivates

Definition 4.2. A total order $<$ on the nodes of a graph has width one if for each node u there is at most one $v < u$ such that (u, v) or (v, u) is an edge of the graph.

The nodes of a tree can be ordered with a width-one ordering in time proportional to the number of nodes in the tree. One way to construct such an ordering is to pick a node and designate it as the root of the tree; then rank its successors; then rank the successors of its successors,..., until the children of all nodes are ranked. Then visit the nodes depth-first, starting from the root, and from the first child of each node. Define $u < v$ if u is visited before v; $<$ is a total order.

Fact 4.1. The ordering on the nodes of a tree generated by depth-first search is of width one.

Proof. Suppose, for contradiction, that this ordering is not of width one. Then there is a node u that is linked by tree edges (u, v_1) and (u, v_2) to two of its predecessors in the ordering generated by depth-first search of the tree. Since $<$ is a total order, assume without loss of generality that $v_1 < v_2 < u$, i.e. v_1 is visited before v_2, and v_2 before u. Since $v_1 < u$ and (v_1, u) is an edge of the tree, depth-first search will visit it immediately after it visits all the children of v_1 ranked before u. Since $v_1 < v_2 < u$, v_2 must be a child of v_1 ranked before u, or a descendant of such a child. In either case, there is a tree edge (v_1, w_1), with w_1 ranked before u, and tree edges $(w_1, w_2)(w_2, w_3)$. (w_{k-1}, w_k) with $w_k = v_2$

$k \geq 1$. But then the tree contains the path $(u, v_1)(v_1, w_1)\ldots(w_{k-1}, v_2)(v_2, u)$, i.e. a cycle, while trees are acyclic.

This decision-ordering technique is designed to reduce the number of constraint checks needed to find a solution. The communication technique I discuss next aims to reduce the size of the domains before search begins by eliminating the 'gold-plating' options discussed in the Introduction; formally, it aims to achieve directed arc consistency (DAC), a property I define next.

Definition 4.3 (Dechter and Pearl, 1989). Let B be a binary, tree-structured CSP with a width-one ordering of its variables. B is directionally arc consistent if for any constraint c with scope $\{x_i, x_j\}$ and $x_i < x_j$), every value a in D_i is supported by some value b in D_j, i.e. (a, b) belong to the extent of constraint c.

The next fact shows why Definitions 4.2 and 4.3 are interesting.

Fact 4.2. Let B be a binary, tree-structured, width-one ordered, directionally arc consistent CSP. Then the BT algorithm will find a solution without rework.

Proof. Rework occurs only at a dead-end, i.e. when a partial assignment $v_1 \ldots v_i$ cannot be consistently extended to the next variable. I show by induction on i that a dead-end can never occur. For $i = 1$, this is guaranteed by DAC. For general i, the assignment $v_1 \ldots v_i$ already satisfies all constraints with scopes in the set $\{x_1, \ldots, x_i\}$, by definition of the BT algorithm. There is at most one constraint with scope containing x_{i+1} and some x_j, $1 \leq j < i$ (width-one ordering guarantees this); call this constraint $c_{j,i+1}$. By DAC there is a value v_{i+1} in D_{i+1} that supports v_j, hence $v_1 \ldots v_i v_{i+1}$ is a consistent extension of $v_1 \ldots v_i$, and no rework will ever take place.

The next fact states the cost of solving a CSP without rework.

Fact 4.3. Let B be a binary, tree-structured, width-one ordered, directionally arc consistent CSP. The BT algorithm will find a solution of B after at most nk constraint checks (k is the size of the largest domain of B).

Proof. The search tree has n levels. By Fact 4.2, the BT algorithm will proceed from one level to the next without rework. At each level, there is at most one $j \leq i$ such that $(j, i + 1)$ is the scope of some constraint. Hence v_j needs to be tested for consistency with some value in D_{i+1} at most k times (until its support in D_{i+1} is found). Hence, at most nk constraint checks will be performed.

These three facts suggest the following strategy for solving a binary, tree-structured CSP; (a) compute a width-one ordering of its variables; (b) make B directionally arc consistent; and (c) find a solution using the BT algorithm. It remains to be seen how task (b) is done; the algorithm that makes B DAC is our communication technique. I present it in two steps. Suppose first that there is an algorithm g that takes as inputs pairs of domains (D_i, D_j) and outputs the set of

elements of D_j that are supported by some element of D_i; if there is no constraint whose scope is the set $\{x_i, x_j\}$, g outputs D_j. The algorithm, call it C, that achieves DAC uses g as a subroutine. It works on a binary, tree-structured, width-one ordered CSP as follows. Start with the last variable x_n in the width-one ordering. Find its parent, i.e. the unique variable preceding it in this ordering and linked to it by an edge of the primal graph. (Each variable has a unique parent because the primal graph is a tree and the ordering is width-one.) Replace $D_{\text{parent}(n)}$ by $g(D_n, D_{\text{parent}(n)})$, i.e. eliminate from the domain of parent(n) all those values not supported by some value in D_n. Repeat for $x_{n-1}, x_{n-2}, \ldots, x_2$.

Fact 4.4. Algorithm C transforms a binary, tree-structured width-one ordered CSP B into an equivalent CSP B′ with the DAC property.

Proof. Let c be a constraint in R with scope $\{x_i, x_j\}$ and $x_i < x_j$. Let a be a value in D_i', the domain of x_i generated by the C algorithm. I show that a has support in D_j'. First note that x_i is the parent of x_j in the width-one ordering. Hence, at step j of the C algorithm, all values in the domain of x_i without support in the domain of x_j are eliminated. The domain of x_i can only shrink or remain the same after step j; hence all its values when C terminates have support in the domain of x_j, since the latter never changes after step j of algorithm C.

Algorithm C uses subroutine g; the latter can be described as follows. It starts with the domains of two variables, D_i and D_j, that are linked by a constraint. It takes the first value in D_j, and checks whether there is a value in D_i that supports it; this step takes at most $|D_i|$ constraint checks. If the answer is yes, it declares this value supported, and proceeds to the second value of D_j. If the answer is no, it deletes this value from D_j and proceeds to its second value. It repeats until all the values of D_j have been either deleted or declared to be supported.

Algorithms C and g are in fact communication techniques; at each step j of g, someone responsible for decision x_j asks someone responsible for decision $x_{\text{parent}(j)}$ whether $x_j = a$ is consistent with some value of $x_{\text{parent}(j)}$. The answers are then codified by algorithm C, which deletes unsupported values. How costly is communication? This is answered by

Fact 4.5. A binary, tree-structured, width-one ordered CSP can be turned into a DAC CSP with at most nk^2 constraint checks.

Proof. Algorithm C goes through n stages; at each stage i it runs the subroutine g $(D_i, D_{\text{parent}(i)})$. Algorithm g checks each value of D_i for consistency with some value of $D_{\text{parent}(i)}$; at most k^2 such checks are needed to find support for every value of $D_{\text{parent}(i)}$ that has such support, and to delete the rest.

Facts 4.1 to 4.5 show that the total cost of solving a binary, tree-structured CSP without rework is $n + nk + nk^2$. Note that no problem-partitioning technique is needed to eliminate rework in this special case. In this sense, a binary, tree-structured CSP has optimal structure. The purpose of problem partitioning is to transform an arbitrary CSP into an equivalent binary, tree-structured CSP.

5. Problem partitioning

This section and the next deal with arbitrary CSPs. They show how arbitrary CSPs can be transformed into equivalent binary, tree-structured CSPs, so that the techniques of Section 4 can be applied to obtain a solution without rework. The peculiar feature of the transformed CSPs is that their 'variables' are subproblems of the original CSP; the domains of the 'variables' are the solution sets of the corresponding subproblems; and all the constraints are binary, equality ones that enforce that each variable of the original CSP is assigned the same value in any two subproblems. Recall that in order to solve a CSP with n variables, r constraints, and domains containing up to k values, one may need up to rk^n constraint checks. Hence, any problem decomposition seems likely to generate large time savings, since $rk^{n_1} + rk^{n_2}$ is smaller than $rk^{n_1+n_2}$ when n_1 and n_2 are close to each other. Problem decomposition, however, creates two new problems, namely extendability and coordination.

A solution of a subproblem may not be extendable to a full solution; and the solutions of two subproblems may be extendable to full solutions, but not to the same full solution. This is illustrated by the following example.

Example 5.1. Consider the CSP defined by $X = \{x_1, x_2, x_3, x_4, x_5\}$, $D_i = \{\alpha, \beta, \gamma, \delta\}$, $R = \{c_{124}, c_{13}, c_{45}, c_{234}\}$, the obvious scopes, i.e. $s(c_{12}) = \{x_1, x_2\}$, and extents given by $e(c_{124}) = \{\alpha\delta\delta, \beta\gamma\alpha, \gamma\beta\gamma, \delta\alpha\gamma\}$, $e(c_{13}) = \{\alpha\delta, \beta\gamma, \gamma\beta\}$, $e(c_{234}) = \{\delta\delta\delta, \gamma\gamma\alpha, \beta\beta\gamma, \alpha\alpha\gamma\}$, $e(c_{45}) = \{\delta\alpha, \beta\gamma, \alpha\delta\}$.

In what follows, I will represent each constraint by its scope, i.e. 12 will stand for c_{12}. The subproblem {124, 234} has four solutions, namely $x_1x_2x_3x_4 = \alpha\delta\delta\delta$, $\beta\gamma\gamma\alpha$, $\gamma\beta\beta\gamma$, $\delta\alpha\alpha\gamma$. The subproblem {124, 13, 234} is satisfied only by the first three of these. The original problem has two solutions, namely $x_1x_2x_3x_4x_5 = \alpha\delta\delta\delta\alpha$, $\beta\gamma\gamma\alpha\delta$. The partial solutions $\delta\alpha\alpha\gamma$ and $\gamma\beta\beta\gamma$ are not extendable to a full solution. The partial solution $\alpha\delta\delta\delta$ of {124, 234} and the partial solution $\beta\gamma\gamma\alpha$ of {124, 23, 234} are both extendable, but to different full solutions.

The example shows that arbitrary decompositions into subproblems will not eliminate rework. The basic tool for finding the right decomposition is the dual graph of a CSP; it records which constraints share which variables, i.e. the potential extendability and coordination problems. In the definition that follows, I use the notation $\langle V, E, l\rangle$ for a labelled graph; V is the set of nodes of the graph, $E \subseteq V \times V$ is its set of edges, and l is a function that assigns a label to each edge.

Definition 5.1. The dual graph of a CSP $\langle X, D, R, s, e\rangle$ is defined by

- $V = R$
- $E = \{(c_1, c_2) \in R \times R : s(c_1) \cap s(c_2) \neq 0\}$
- $l(c_1, c_2) = s(c_1) \cap s(c_2)$.

The dual graph of the CSP in Example 5.1 has nodes 124, 13, 45, 234; edges (124, 13), (124, 234), (13, 234), (45, 234), (124, 45); and labels of edges (in the same order) 1, 24, 3, 4, 4.

I will consider only CSPs with connected dual graphs; otherwise, an efficient algorithm can find the connected components of the (disconnected) dual graph, and each of them can be solved separately. I will consider only CSPs with reduced dual graphs, i.e. CSPs in which no constraint's scope is a subset of another constraint's scope; otherwise, if $s(c) \subset s(c')$, c can be eliminated and $e(c')$ can be restricted to contain only tuples consistent with $e(c)$ on $s(c)$. In a CSP with reduced dual graph, each constraint is uniquely represented by its scope; I will adopt this notation from now on.

What is the best criterion for problem-partitioning? In Example 5.1, consider the partitioning {124, 13} and {234, 45}. The two subproblems have to communicate to set the values of x_2, x_3, x_4 consistently; that is why they are linked with edges (124, 234) and (13, 234), labelled with 24 and 3 respectively. The set of variables {2, 3, 4} whose values need to be set consistently is not a subset of either 124 or 13. Hence, subproblem {234, 45} cannot communicate with {124, 13} through a single, representative constraint. Any 'agreement' with constraint 124 on the values of x_2, x_4 might conflict with an 'agreement' with constraint 13 on the value of x_3. We would like each subproblem to be 'coherent enough' to be able to communicate with other subproblems throughout a single, representative constraint. To formalize this idea, I introduce two definitions due to Gyssens et al. (1994).

Definition 5.2. Let $\langle R, E, l \rangle$ be the dual graph of some CSP; let $H \subseteq R$, $F \subseteq R - H$ be two sets of constraints. F is connected with respect to H if, for any two constraints c, c' in F, there exists a sequence of constraints $c = c_1, c_2, \ldots, c_n = c'$ in F such that for all $i = 1, \ldots, n-1$,

- (c_i, c_{i+1}) is an edge in E,
- $l(c_i, c_{i+1})$ contains a variable not in the scope of any of the constraints in H,

for each $i = 1, \ldots, n-1$.

Intuitively, subproblem F is connected with respect to subproblem H if any two of the constraints in F share, directly or indirectly, variables whose value cannot be set in subproblem H. Hence, even if H is solved and its (partial) solution imposed on F, there may still be extendability and coordination problems in F.

Example 5.1 (continued). Let $H = \{124\}$. Then $F = \{13, 234, 45\}$ is not connected wrt H, because the label 4 on the edge that links 234 to 45 belongs to the scope of a constraint in H. F consists of two connected components wrt H, namely {13, 234} and {45}.

The idea of the 'coherent enough' subproblem that can communicate with other subproblems through a single representative is captured in the following definition.

Definition 5.3. Let $\langle R, E, l \rangle$ be the dual graph of some CSP. A hinge is a set of constraints H that contains at least two elements and is equal to either R or a subset of R that satisfies: for each connected component C_i of $R - H$ wrt H, there is a constraint h_i in H such that $\text{scope}(C_i) \cap \text{scope}(H) \subseteq \text{scope}(h_i)$, where the scope of a subproblem is the union of the scopes of its constraints. A hinge is minimal if all its proper subsets are not hinges.

Example 5.1 (continued). The minimal hinges of the CSP defined in Example 5.1 are $H_1 = \{124, 13, 234\}$, $H_2 = \{45, 234\}$, $H_3 = \{45, 124\}$.

Why should CSPs be decomposed into minimal hinges? Two duality-type results provide an answer: decomposition into hinges is the only partitioning method that creates no extendability problems where none exist in the first place.

Definition 5.4. A problem decomposition method preserves extendability if any subproblem F it creates has the following property:

- if every solution of every constraint in R extends to a full solution, then every solution of F extends to a full solution.

The next example shows that this is not a trivial property.

Example 5.2. Consider the CSP given by $X = \{x_1, x_2, x_3, x_4\}$, $D_i = \{\alpha, \beta, \gamma\}$ $R = \{12, 23, 34, 14\}$, and $e(12) = \{\gamma\alpha, \gamma\beta\}$ $e(23) = \{\alpha\alpha, \beta\beta\}$, $e(34) = \{\alpha\gamma, \beta\gamma\}$, $e(14) = \{\gamma\gamma\}$. There are two full solutions, $\gamma\alpha\alpha\gamma$ and $\gamma\beta\beta\gamma$. Every solution of every constraint is extendable. The subproblem $\{12, 14, 34\}$, however, has the nonextendable solutions $\gamma\alpha\beta\gamma$, $\gamma\beta\alpha\gamma$; both violate constraint 23.

The next fact, due to Gyssens *et al.* (1994), shows that the hinge decomposition method preserves extendability. Its proof is in the appendix.

Fact 5.1. Let B be a CSP where every solution of every constraint is extendable. If H is a hinge of B, then every solution of H is extendable.

The next fact shows that the hinge decomposition method is the only one that preserves extendability in all CSPs. Its proof is in the Appendices.

Fact 5.2. Let $\langle X, R, s \rangle$ be a CSP skeleton, i.e. a CSP problem whose domains and extents have not been specified. Let F be a subset of R that is not a hinge. Then, there exist domains $D_1 \ldots D_n$ and extents $e(c)$ for each constraint c in R such that in the resulting CSP $\langle X, D, R, s, e \rangle$:

- *every solution of every constraint in R is extendable;*
- *F has a nonextendable solution.*

The last fact that makes hinge decomposition interesting is that the size of the largest minimal hinge is an invariant of the dual graph of a CSP; it is due to Gyssens (1986). The size of the largest subproblem of a CSP determines solution time. It makes sense, then, to classify NPD problems into classes representing their degree of difficulty (as measured by the size of their largest minimal hinge); call this parameter the degree of cyclicity of the dual graph of a CSP. The next example shows that there is no relationship between the number of variables of a CSP and its degree of cyclicity. It follows that the number of variables of a problem is not a very precise indicator of its degree of difficulty.

Example 5.3. Consider a sequence $A_2, \ldots, A_n$ of CSPs. Each A_k has n variables. The constraint set of each A_k, $k = 2, 3, \ldots, n$, is $R_k = \{12, 23, \ldots, (k-1)k, k1, k(k+1), \ldots, (n-1)n\}$. The largest minimal hinge of A_k is the set $\{12, 23, \ldots, k1\}$ for $k \geq 2$; its size is k. Hence problems with the same number of variables n can have degrees of cyclicity ranging from 2 to n.

This section has looked at individual subproblems and their properties. The next one discusses how subproblems should be connected with each other. Recall Clausing's (1994, p. 40) statement: "The formation of the best interlocking structure of teams is a key success factor."

6. The best interlocking structure of teams

This section builds on the ideas of Sections 4 and 5 to find a way of integrating subproblem solutions that minimizes rework. The basic ideas are three. First, as the results of Section 5 suggest, each subproblem should be a minimal hinge; so that any extendability present in the constraints of the original CSP is preserved. Secondly, as the results of Section 4 suggest, subproblems should be linked to each other to form a binary, tree-structured CSP; so that a solution can be discovered without rework after application of the decision-ordering and communication techniques. Thirdly, given that the size of the largest subproblem determines solution time, subproblems of maximum size should be used as sparingly as possible. I start formalizing these ideas with the notion of a hinge tree, due to Gyssens et al. (1994).

In the next definition, the scope of a hinge is the set of variables that appear in the scope of some constraint in the hinge.

Definition 6.1. Let $G = \langle R, E, l \rangle$ be the dual graph of a CSP. A hinge tree of G is a labelled tree with nodes N, edges A and labelling function λ such that
(1) each node in N is a minimal hinge of G;
(2) each constraint in R belongs to some node in N;
(3) if (H, H') is an edge in A, then $H \cap H'$ is a singleton; and $\lambda(H, H') = H \cap H'$;

(4) if (H, H') is an edge in A, then scope $(H) \cap$ scope$(H') =$ scope$(H \cap H')$;
(5) if $(H_1, H_2)(H_2, H_3), \ldots, (H_{n-1}, H_n)$ is a path of edges in A, then scope(H_1) $\cap$ scope$(H_n) \subseteq$ scope(H_i), $i = 2, 3, \ldots, n-1$.

Property 1 was justified in Section 5. Property 2 is obviously necessary to cover the original CSP. The next three examples motivate the three remaining properties. (Recall that hinges in the hinge tree correspond to subproblems to be solved independently of each other.)

Example 6.1. This example motivates property (3) of a hinge tree. Let B be the CSP defined by $X = \{x_1, x_2, x_3, x_4, x_5\}$, $R = \{12, 13, 234, 235\}$. Its dual graph has R as its node set; edges (12, 13), (12, 234), (12, 235), (13, 234), (13, 235), (234, 235); and labels of edges, respectively, 1, 2, 2, 3, 3, 23. This graph contains three minimal hinges, namely $H_1 = \{12, 13, 234\}$, $H_2 = \{12, 13, 235\}$, $H_3 = \{234, 235\}$. There are two hinge trees satisfying properties (1), (2) and (3), namely these with node sets $\{H_i, H_3\}$, $i = 1, 2$. There is one hinge tree that satisfies properties (1), (2) but violates property (3), namely $\{H_1, H_2\}$. Problem decomposition $\{H_1, H_2\}$ requires the solution of two subproblems of maximum size, whereas problem decompositions $\{H_i, H_3\}$, $i = 1, 2$, require the solution of only one subproblem of maximum size.

Example 6.2. This example motivates property (4) of hinge trees. Consider the CSP B with $X = \{x_1, x_2, \ldots, x_7\}$ and $R = \{23, 124, 127, 135, 136\}$. Its dual graph has nodes R, and there is an edge connecting any two nodes. The minimal hinges are $H_1 = \{23, 124, 135\}$, $H_2 = \{23, 124, 136\}$, $H_3 = \{23, 127, 135\}$, $H_4 = \{23, 127, 136\}$, $H_5 = \{124, 127\}$ and $H_6 = \{135, 136\}$. There are two trees that satisfy properties 1, 2 and 3 but violate 4, with node sets $\{H_1, H_4\}$ and $\{H_2, H_3\}$. To see the violation of property 4 by the first of these trees, note that scope$(H_1) \cap$ scope$(H_4) = \{1, 2, 3\}$, while scope$(H_1 \cap H_4) =$ scope$(23) = \{2, 3\}$. The trees that satisfy properties 1 through 4 are those with node sets $\{H_i, H_5, H_6\}$ for $i = 1, 2, 3, 4$. Every problem decomposition of the latter form contains only *one* hinge with three elements; every problem decomposition that violates property 4 contains *two* hinges with three elements each. Again, a violation of one of the properties of a hinge tree results in larger than necessary subproblem size.

Example 6.3. This example motivates property (5) of hinge trees. Let B be a CSP with $X = \{x_1, \ldots, x_7\}$ and $R = \{14, 36, 47, 139, 456, 123, 58\}$. The minimal hinges are $H_1 = \{14, 36, 139, 456\}$, $H_2 = \{14, 36, 123, 456\}$, $H_3 = \{139, 123\}$, $H_4 = \{47, 456\}$, $H_5 = \{58, 456\}$. The trees of hinges that satisfy properties 1 through 4 but violate property 5 have node sets $\{H_i, H_m, H_j, H_n\}$ where $i \neq j$, $i, j = 1, 2$ and $m \neq n$, $m, n = 4, 5$; and edges (H_i, H_m), (H_m, H_j), (H_j, H_n), all labelled with 456. The trees of hinges that satisfy all five properties have node sets $\{H_i, H_3, H_4, H_5\}$, $i = 1, 2$; and edges (H_i, H_3), (H_i, H_4), (H_i, H_5), the first edge labelled with 123 and the other two with 456. Again, the trees of hinges that satisfy all five properties involve only

one hinge of maximum size; while all those trees of hinges that satisfy only the first four properties involve *two* hinges of maximum size, resulting in larger than necessary subproblem size.

I now describe exactly how a hinge tree is used to solve a CSP. There are three ideas involved: problem decomposition into minimal hinges; subproblem communication through the edges of the hinge tree; and solving not the original CSP (by picking values for each individual variable out of its domain), but an equivalent, dual CSP (by picking tuples of values out of each subproblem's solution set). The basic construction is described by the next definition.

Definition 6.2. Let $B = \langle X, D, R, s, e\rangle$ be a CSP, $G = \langle R, E, l\rangle$ its dual graph, and $\langle N, A, \lambda\rangle$ a hinge tree of G. The dual CSP of B is defined to be a binary tree-structured CSP $B' = \langle X', D', R', s', e'\rangle$, as follows:

- $X' = N$; $R' = A$; $s'(H_1, H_2) = \{H_1, H_2\}$ for each (H_1, H_2) in R';
- for each hinge H in N, D'_H = solution set of H;
- for each edge (H_1, H_2) in A, $e'(H_1, H_2)$ = all pairs (u^1, u^2) where u^i is a solution of H_i, $i = 1, 2$; and where u^1 and u^2 assign the same values to the variables they share, i.e. to the variables in scope$(H_1) \cap$ scope(H_2).

The dual of a CSP, being binary and tree structured, can be solved without rework after the methods of Section 4 have been applied. The next example takes us through the steps of the solution process, assuming that a hinge tree has already been constructed and the solution sets of the hinges computed.

Example 6.4. The CSP B is defined by $X = \{x_1, x_2, \ldots, x_9\}$; $D_i = \{\alpha, \beta, \gamma, \delta\}$, $i = 1, 2, \ldots, 9$; $R = \{12, 1345, 238, 456, 57, 67, 89\}$; $e(12) = \{\alpha\alpha, \beta\beta, \gamma\gamma, \delta\delta, \alpha\beta, \beta\alpha\}$, $e(1345) = \{\alpha\beta\beta\alpha, \beta\gamma\gamma\beta, \gamma\delta\delta\gamma, \delta\delta\delta\delta\}$, $e(238) = \{\alpha\gamma\delta, \alpha\beta\alpha, \beta\gamma\beta, \gamma\delta\gamma, \delta\delta\delta\}$, $e(456) = \{\gamma\gamma\alpha, \beta\alpha\alpha, \delta\gamma\beta, \delta\delta\delta\}$, $e(57) = \{\alpha\alpha, \beta\beta, \gamma\gamma, \delta\delta\}$, $e(67) = \{\alpha\alpha, \beta\beta, \gamma\gamma\}$, $e(89) = \{\alpha\alpha, \delta\delta\}$.

A hinge tree of B has nodes H_1, H_2, H_3, H_4 where $H_1 = \{12, 1345, 238\}$, $H_2 = \{1345, 456\}$, $H_3 = \{456, 57, 67\}$, $H_4 = \{238, 89\}$; and edges (H_1H_2), (H_2H_3), (H_1H_4), labelled, respectively, with 1345, 456, 238. The solution sets of each hinge are given by

$D'_{H1} = \{x_1x_2x_3x_4x_5x_8 = \beta\beta\gamma\gamma\beta\alpha, \gamma\gamma\delta\delta\gamma\gamma, \delta\delta\delta\delta\delta\delta, \alpha\alpha\beta\beta\alpha\alpha, \beta\alpha\gamma\gamma\beta\alpha\}$,
$D'_{H2} = \{x_1x_3x_4x_5x_6 = \gamma\delta\delta\gamma\beta, \delta\delta\delta\delta\delta, \alpha\beta\beta\alpha\alpha\}$,
$D'_{H3} = \{x_4x_5x_6x_7 = \gamma\gamma\alpha\alpha, \beta\alpha\alpha\alpha, \delta\delta\delta\delta\delta\}$,
$D'_{H4} = \{x_2x_3x_8x_9 = \alpha\gamma\delta\delta, \delta\delta\delta\delta, \alpha\beta\alpha\alpha\}$.

A width-one ordering of the hinge-tree is $H_1 < H_2 < H_3 < H_4$. The parent of H_4 and H_2 is H_1, and the parent of H_3 is H_2. The DAC algorithm starts from H_4, and eliminates from D'_{H1} all tuples inconsistent with all values in D'_{H4}; since H_1 and H_4 share the variables 238, every solution of H_1 that does not agree with any solution of H_4 on the variables 238 is eliminated. We obtain $D''_{H1} =$

$\{x_1x_2x_3x_4x_5x_8 = \delta\delta\delta\delta\delta\delta,\ \alpha\alpha\beta\beta\alpha\alpha,\ \beta\alpha\gamma\gamma\beta\alpha\}$, i.e. the first two solutions of H_1 are eliminated by the DAC algorithm.

The DAC algorithm now proceeds to the hinge immediately preceding H_4 in the width-one ordering, namely H_3; and eliminates from D'_{H2} (the domain of the parent of H_3) all tuples that do not agree on the variables 456 with any tuple in D'_{H3}. The resulting domain of H_2 is $D''_{H2} = \{x_1x_3x_4x_5x_6 = \delta\delta\delta\delta\delta,\ \alpha\beta\beta\alpha\alpha\}$; i.e., H_2 has lost one solution. Finally, the DAC algorithm proceeds to H_2, and eliminates from D''_{H1} (the domain of the parent of H_2) all tuples that do not agree on the variables 1345 with any tuple in D''_{H2}. The resulting domain of H_1 is $D'''_{H1} = \{x_1x_2x_3x_4x_5x_8 = \delta\delta\delta\delta\delta\delta,\ \alpha\alpha\beta\beta\alpha\alpha\}$; i.e., H_1 has lost one more solution. The BT algorithm now takes over from the DAC algorithm.

It starts from the first value of the first variable in the dual CSP, i.e. from the tuple $x_1x_2x_3x_4x_5x_8 = \delta\delta\delta\delta\delta\delta$ in D'''_{H1}. It proceeds to the next dual variable, i.e. H_2, and discovers that $x_1x_3x_4x_5x_6 = \delta\delta\delta\delta\delta$ is consistent with its previous assignment. It then proceeds to H_3, H_4, in that order, and discovers that an assignment of δ to all variables in the scope of each hinge is consistent with all previous assignments. The BT algorithm terminates with the solution $x_i = \delta$, all i, without having performed any rework, since no assignment of δs to each hinge's scope ever had to be undone. If the BT algorithm had started from the other value in D'''_{H1}, namely $\alpha\alpha\beta\beta\alpha\alpha$, it would have extended it without rework to the other full solution, namely $\alpha\alpha\beta\beta\alpha\alpha\alpha\alpha$.

At first sight, it seems grossly inefficient to calculate the solution *sets* of each subproblem in the hinge tree in order to compute just *one* solution of the overall design problem. As is shown in Section 7, this impression is misleading. There is, in fact, an intriguing similarity between the solution method just described and Toyota's product development process, as described by Ward *et al.* (1995, p. 49). "Toyota's development process looks expensive, clumsy and inefficient to outsiders (including other Japanese companies). Toyota does not conduct a point-to-point search but follows a version of set-based concurrent engineering: 1. The team defines a *set* of solutions at the system level, rather than a single solution. 2. It defines sets of possible solutions for various subsystems. 3. It explores these possible subsystems in parallel using analysis, design rules and experiments to characterize a set of possible solutions. 4. It uses the analysis to gradually narrow the sets of solutions, converging slowly toward a single solution. 5. Once the team establishes the single solution for any part of the design, it does not change it unless absolutely necessary. In particular, the single solution is *not* changed to gain improvements, to climb the hill." Note that point 5 claims no backtracking between subproblems. In another passage, Ward *et al.* (1995, p. 44) identify a Toyota practice similar in effect to our DAC technique: "Toyota uses engineering checklists to represent sets of manufacturable designs in order to constrain the styling process—for example requiring the stylists to avoid some extreme body curves. Each functional group in Toyota has identified infeasible designs and systematically records these in the checklists."

7. Costs and benefits of organizing for fast NPD

The previous sections have shown that a decision-ordering, a communication, and a problem-partitioning technique will solve a CSP without any rework between subproblems. What is the overall cost of applying these methods, and how does it compare with the cost of solving a CSP without them? Given that typical NPD problems are large (recall the figures cited in point six of the Introduction), it makes sense to adopt rate of growth as our measure of cost. To see what this means, recall that in Section 4 it was seen that to solve binary, tree-structured problems we need to perform n constraint checks to obtain a width-one ordering; nk^2 constraint checks to obtain arc consistency, and nk constraint checks to solve the resulting CSP. The overall cost is thus $n + nk + nk^2$. Considering only the rate of growth means replacing this expression by its leading term, ignoring its coefficient. Hence

Fact 7.1. The cost of solving a binary, tree-structured CSP is k^2, where k is the size of its largest domain.

This is standard computer science practice. Cormen *et al.* (1990, p. 10) state: "It is the rate of growth of the running time that really interests us. We therefore consider only the leading term of a formula since the lower order terms are relatively insignificant for large n. We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs."

Recall also that to solve an arbitrary CSP we need rk^n constraint checks. Hence

Fact 7.2. The cost of finding a solution of an arbitrary CSP with the (unaided) BT algorithm is k^n.

To apply the methods of Section 6, we need to compute the solution sets of subproblems of the original CSP, as opposed to finding a single solution. How is this done? If B is the CSP whose solution set we seek, apply the BT algorithm to its dual CSP, with one difference; once the first solution is identified, instead of stopping, discard the portion of the search tree already explored by the BT algorithm, and apply the BT algorithm to the rest. Stop when the whole search tree has been explored; any time a new solution is identified, add it to the solution set. If the size of the extents of the constraints in the original CSP has maximum l, and if there are r constraints in the original CSP, the solution set can be computed after at most l^r constraint checks. Hence, if d is the degree of cyclicity of the dual graph of a CSP, computing the solution set of any subproblem that is a node of a hinge tree takes at most l^d constraint checks, since d is the maximum number of constraints any such subproblem may contain.

Fact 7.3. The cost of computing the solution sets of the subproblems that are the nodes of a hinge tree is l^d.

To apply the methods of Section 6 we need an algorithm that computes a hinge tree. Here is what the algorithm basically does. It picks some constraint c in R and computes the connected components $C_1, \ldots, C_m$ of $R - \{c\}$wrt c. It then forms the hinges $H_i = C_i \cup \{c\}$; and the tree with nodes $H_1, \ldots H_m$, edges $(H_i H_{i+1})i = 1, \ldots, m-1$, and labels of all edges c. It then marks c as used, and repeats the same process with some H_i that contains more than two elements and at least one unused element. It stops when all constraints have been marked used, or when all nodes in the constructed tree contain only used elements. In case $R - \{c\}$, or some $H_i - \{c\}$, contains only one connected component wrt c, c is marked used, another, unused constraint is picked, and the process repeated. The cost of this algorithm is actually the cost of computing connected components; connected components are computed by standard graph algorithms (Cormen *et al.*, 1990, p. 488).

Fact 7.4. (Gyssens et al., 1994). The cost of computing a hinge tree is r^2, where r is the number of constraints of the original CSP.

Note also that the number of nodes in a hinge tree is bounded by r, since hinges are constructed as connected components of subsets of R until R is covered.

Finally, the methods of Section 6 require the discovery of a solution of a binary, tree-structured CSP using the methods of Section 4. The variables of this CSP are the nodes of the hinge tree. We know from Fact 7.1 that the cost of solving such a problem is determined by the size of the largest domain of the CSP, i.e., in our case, by the size of the largest solution set pertaining to a node of the hinge tree. This is no larger than l^d, where l is the size of the largest extent of a constraint and d the number of constraints in the largest minimal hinge. Hence, by Fact 7.1,

Fact 7.5. The cost of finding a solution to the dual CSP defined by the hinge tree of the original CSP is l^{2d}.

By Facts 7.3, 7.4 and 7.5 we have that the sum of costs of applying the methods of Section 6 is $r^2 + l^d + l^{2d}$. Hence, neglecting all but the leading terms, we obtain

Fact 7.6. The cost of finding a solution to a CSP using the decision-ordering, communication and problem-partitioning techniques is l^{2d}.

Comparing l^{2d} with k^n, we conclude that the organizational methods that eliminate rework between subproblems are worth their cost if the degree of cyclicity d of a CSP is small relative to the number of variables n of the same CSP.

8. Conclusion

This paper considers technological change, and in particular new product and process development, from a new perspective, that of complex problem solving. Technological change is made possible by organizational techniques that reduce the complexity of problem solving; differences in the technological performance of firms are attributed to their different problem-solving styles. More specifically, this paper proposes the formalism of constraint satisfaction problems (CSP) as (a) expressive enough to capture design problems, including design for manufacturing, listening to the voice of the customer, and robust design; and (b) tractable enough to be fully decidable, i.e. to possess an algorithm that will always correctly generate a solution, or the set of all solutions. This algorithm produces too much rework; in fact, rework is the main reason for and symptom of slow new product development, both on theoretical and empirical grounds. To minimize rework, this paper proposes three distinct techniques: a technique to order decisions; a technique to make more specific the goals of design by communication prior to decision making; and a problem-partitioning technique that minimizes subproblem size subject to the constraint of not creating extra extendability or coordination problems. The three techniques together provably minimize rework and thus accelerate decision making. The problem-partitioning technique in particular associates with each design problem a tree of subproblems; the size of the largest subproblem is an index of the degree of difficulty of the design problem. The nature of the technical and marketing constraints faced by product designers determines both the organizational form (the tree of subproblems) appropriate for the design problem and the speed of reaching a solution (an exponential term whose exponent is the size of the largest subproblem). This organizational form is reusable because it depends only on structural information, i.e. on which variables enter which constraints, not on the values that satisfy each constraint. The cost of building this organizational form is quadratic in the size of the design problem; i.e., other things being equal, larger firms with many interdependent products face disproportionately larger costs of adjusting their organizational forms to changes in the underlying structural information. If the underlying structure remains the same, though, an incumbent firm that has already computed its organizational form can amortize it over many different NPD efforts, and will develop new products faster than a new entrant who still has to figure out how to organize for new product development.

The empirical literature on new product development stresses the importance of design in determining operational efficiency. Baily and Gersbach (1995, p. 347), summarizing their findings on the determinants of operating efficiency, state: "Traditional determinants, such as capital intensity and scale were found to play a role. But innovations such as design for manufacturing and workplace organization turned out to be be even more important." The empirical literature by itself, however, cannot pinpoint exactly how fast product developers differ from slow

product developers; a result of this is that it is not clear to slow developers what they should be imitating to become fast product developers. Ward *et al.* (1995) make this point clearly in their discussion of Toyota's NPD organization, which they call set-based concurrent engineering: "We do not know enough about how set-based concurrent engineering is or should be performed. Toyota's approach is not well-defined or documented"; and "Since there is no proven formal methodology, learning the process will be slow and error-prone".

Furthermore, the empirical literature has not established *causal* relationships between time and problem-solving techniques; only associations have been documented. Ward *et al.* (1995) state: "Each advantage of set-based concurrent engineering described earlier represents a hypothesis—that there is a causal relationship between Toyota's success and its use of set-based concurrent engineering. An important task for further research is therefore to demonstrate this causal link more carefully. Unfortunately such causes are difficult to show in complex organizations." Seen in this light, this paper contributes a well-defined set of techniques that provably accelerate NPD; these techniques could, therefore, provide a starting point for filling this gap in the literature.

Acknowledgements

I would like to thank Mark Salmon and Alan Kirman for the opportunity to present earlier versions of this work in their workshops; Ed Green for a useful conversation; B. Visser, A. Savkov, H. Sabourian and seminar participants at CORE, EUI and the University of Pittsburgh for useful comments. All errors are mine.

Appendix A

This appendix is a prerequisite for reading the proofs of facts 5.1 and 5.2 in appendices B and C, respectively. Its purpose is to find a small set of properties (two in fact) that fully characterize the solution set of a CSP. The first definition captures the structure of a CSP by a directed graph.

Definition A.1. The directed graph G_B generated by a CSP B has nodes $N_B = X \cup R$; there is an edge $\pi_x^c : c \to x$ for each c in R and each x in $s(c)$.

The second definition will describe the domains and extents of a CSP, and the relations between them, in terms of a single map T. Recall that each $e(c)$ is a subset of $\prod_{x \in s(c)} D_x$; the map $m_c : e(c) \to \prod_{x \in s(c)} D_x$ is then the unique embedding defined by $m_c(u) = u$, u in $e(c)$. Let $P_x : \prod_{x=1}^n D_x \to D_x$, $p^c : \prod_{x=1}^n D_x \to \prod_{x \in s(c)} D_x$, $p_x^c : \prod_{x \in s(c)} D_x \to D_x$ be projection functions; they obviously satisfy

$$P_x = p_x^c p^c \tag{A1}$$

where fg denotes the composition of functions f, g.

Definition A.2. The diagram of shape G_B defined by a CSP B in a map T such that
- $T(x) = D_x$, x in X,
- $T(c) = e(c)$, c in R,
- $T(\pi_x^c): T(c) \to T(x)$, ($x$ in $s(c)$, c in R)

is the function that assigns to each vector in $T(c)$ its x-component, i.e.

$$T(\pi_x^c) = p_x^c m_c. \tag{A2}$$

Recall that the solution set L of a CSP is a subset of $\prod_{x=1}^{n} T(x)$; the map $m_L: L \to \prod_{x=1}^{n} T(x)$, $m_L(u) = u$, is needed in the next definition. Our purpose now is to describe L uniquely in terms of T only.

Definition A.3. The family of maps $\langle L, \lambda_a : a \in \text{nodes}(G_B)\rangle$ is defined by
- $\lambda_x: L \to T(x)$ (for each x in X)

is the map that assigns to each u in L its x-component, i.e.

$$\lambda_x = p_x m_L. \tag{A3}$$

- $\lambda_c: L \to T(c)$ (for each c in R)

is the map that assigns to each u in L its component in $s(c)$, i.e.

$$m_c \lambda_c = p^c m_L. \tag{A4}$$

This family of maps satisfies a commutativity property that constitutes half of the definition of L in terms of T.

Fact A.1. The family of maps in definition A.3 satisfies, for each edge $\pi_x^c : c \to x$ in G_B,

$$\lambda_x = T(\pi_x^c)\lambda_c. \tag{A5}$$

Proof: $\lambda_x =^{(A3)} p_x m_L =^{(A1)} p_x^c p^c m_L =^{(A4)} p_x^c m_c \lambda_c =^{(A2)} T(\pi_x^c)\lambda_c$.

The next definition gives a (standard) name to this property of the solution set.

Definition A.4. A commutative cone with base T is a finite set L' and a family of maps $\lambda'_a: L' \to T(a)$, one for each node a of G_B, such that for each edge $\pi_x^c : c \to x$ in G_B

$$\lambda'_x = T(\pi_x^c)\lambda'_c. \tag{A6}$$

The next property defines the solution set L (together with the maps λ_a) as the 'largest' commutative cone with base T.

Fact A.2. Let (L', λ'_a) be a commutative cone with base T. Then there is a unique function $f: L' \to L$ such that

$$\lambda'_a = \lambda_a f \text{ for all nodes } a \text{ of } G_B, \text{ i.e.,}$$

$$\lambda'_x = \lambda_x f \text{ for all } x \text{ in } X, \text{ and} \tag{A7}$$

$$\lambda'_c = \lambda_c f \text{ for all } c \text{ in } R. \tag{A8}$$

Proof: Uniqueness of f is easy. If $h: L' \to L$ also satisfies (A7) and (A8), then by (A7) $\lambda_x f = \lambda_x h$, and by (A3) $p_x m_L f = p_x m_L h$, for all x. Then $m_L f = m_L h$ because the maps p_x are projections, and $f = h$ because m_L is one-to-one (being an embedding). To show existence, note that by (A7) the only possible definition of f is $f(u') = (\lambda'_1(u'), \ldots, \lambda'_n(u'))$, u' in L, or equivalently

$$p_x f = \lambda'_x, \ x \text{ in } X. \tag{A9}$$

It still has to be shown, though, that f maps into the solution set L, because (A9) defines it as a function from L' into $\prod_{x=1}^{n} T(x)$. A sufficient condition for this is that, for each constraint c, $m_c \lambda'_c = p^c f$, because λ'_c does map into $e(c)$. It suffices to show, then, that $p_x^c m_c \lambda'_c = p_x^c p^c f$ for all x in $s(c)$. In fact, $p_x^c p^c f = {}^{(A1)} p_x f = {}^{(A9)} \lambda'_x = {}^{(A6)} T(\pi_x^c) \lambda'_c = {}^{(A2)} p_x^c m_c \lambda'_c$.

The next definition provides a (standard) name for the properties of the solution set L shown in Facts A.1 and A.2.

Definition A.5. A limit of T is a finite set M, and a family of maps $\mu_a: M \to T(a)$, $a \in \text{nodes}(G_B)$, that satisfy two properties

- *Commutativity:* (M, μ_a) is a commutative cone with base T.
- *Universality:* If (M', μ'_a) is a commutative cone with base T, there is a unique map $f: M' \to M$ that satisfies $\mu'_a = \mu_a f$ for all nodes a of G_B.

Facts (A.1) and (A.2) have shown that the solution set L of a CSP B (together with the functions λ_x, λ_c given by (A7), (A8) resp.) is a limit of T. The next two facts show that T has essentially only one limit. In other words, L is uniquely characterized by its property of being a limit of T. (This property suffices to show Facts 5.1 and 5.2.)

Fact A.3. Let (M, μ_a) be a limit of T. The family of maps $\langle \mu_a \rangle$ is jointly monic in the sense that if $h_i: N \to M$, $i = 1, 2$ are two functions that satisfy $\mu_a h_1 = \mu_a h_2$ for all nodes a of G_B, then $h_1 = h_2$.

Proof: Consider the family of maps (N, υ_a) where $\upsilon_a = \mu_a h_1 = \mu_a h_2$; it is a commutative cone with base T, since, for each edge $\pi_x^c: c \to x$ in G, $T(\pi_x^c)\upsilon_c = T(\pi_x^c)\mu_c h_i = \mu_x h_i = \upsilon_x$. Hence, by the defining property of limits, there is a unique $f: N \to M$ such that $\upsilon_a = \mu_a f$. But then $\upsilon_a = \mu_a h_1 = \mu_a h_2$ for all a, and the uniqueness property of f, imply $h_i = f$, $i = 1, 2$. But then $h_1 = h_2$.

Fact A.4. If (M, μ_a) and (N, υ_a) are limits of T, then M and N are isomorphic, i.e. there exist functions $f: M \to N$ and $g: N \to M$ such that

$$f g = id_N, \ g f = id_M$$

where $id_A: A \to A$ is the identity function on set A.

Proof: (M, μ_a) is a limit of T, and (N, υ_a) a commutative cone with base T. There is then a unique $g: N \to M$ such that $\upsilon_a = \mu_a g$. Similarly, (N, υ_a) is a limit of T, and (M, μ_a) is a commutative cone with base T. Hence, there is a unique

$f: M \to N$ such that $\mu_a = \upsilon_a f$. Then $\upsilon_a fg = \mu_a g = \upsilon_a = \upsilon_a id_N$, all a. Then, by Fact A.3, $fg = id_N$. Similarly, $\mu_a gf = \upsilon_a f = \mu_a = \mu_a id_M$, all a; then, by Fact A.3, $gf = id_M$.

Appendix B

Fact 5.1. Let B be a CSP where every solution of every constraint is extendable. If H is a hinge of B, then every solution of H is extendable.

Proof: Let L be the solution set of B, and let $\lambda_a: L \to T(a)$ be the family of maps in Definition A.3. Recalling that $T(c) = e(c)$ for all c in R, extendability of every solution of every constraint means that each λ_c is onto. This implies that each λ_c has at least one one-sided inverse, i.e. a map $\bar{\lambda}_c: T(c) \to L$ such that

$$\lambda_c \bar{\lambda}_c = id_{T(c)}. \tag{B1}$$

$\bar{\lambda}_c$ is in fact an extension map that assigns to each solution u of the constraint c one of the full solutions extending u.

The solution set of H can be defined, as Appendix A has shown, as the limit (L_H, λ_a^H) of a diagram T_H of shape G_H. G_H is the graph with nodes scope $(H) \cup H$ and edges $\pi_x^c: c \to x$, c in H, x in $s(c)$. T_H is the restriction of T on G_H. Note that $(L, \lambda_a: a \in s(H) \cup H)$ is a commutative cone with base T_H. Hence, by the limit property of (L_H, λ_a^H), there is a unique $f: L \to L_H$ such that

$$\lambda_c = \lambda_c^H f, \ c \text{ in } H, \tag{B2}$$

$$\lambda_x = \lambda_x^H f, \ x \text{ in } s(H). \tag{B3}$$

The map f restricts each solution of the CSP B to a solution of the subproblem H.

I now need to show that each solution of H (i.e. each element of L_H) is extendable to a solution of B (i.e. to an element of L). Let $g: L_H \to L$ be the extension (if it exists). It has to satisfy three properties to be in fact an extension. First, $fg = id_{L_H}$: if u solves H, and $g(u)$ is its extension to a full solution, then the restriction $f(g(u))$ of $g(u)$ on H should be u itself. Secondly, for each x in $s(H)$, $\lambda_x^H = \lambda_x g$: if u solves H and $g(u)$ is its extension to a full solution, then u and $g(u)$ should agree on the variables of H. Thirdly, for each c in H, $\lambda_c^H = \lambda_c g$: if u solves H and $g(u)$ is its extension to a full solution, then u and $g(u)$ should agree on the values they assign to the variables in the scope of each constraint in c.

The rest of this appendix shows the existence of such a g. The main idea is that, exactly because H is a hinge, the limit (L^H, λ_a^H) of T_H can be extended to a commutative cone with base T. Since (L, λ_a) is a limit of T, the function $g: L^H \to L$ will be defined uniquely by the universality property of limits. I start by defining $\lambda_a^H: L_H \to T(a)$ for a not in $H \cup s(H)$. Let $C_1, \ldots, C_m$ be the

connected components of $R - H$ with respect to H. Since H is a hinge, for each i there is a constraint h_i in H such that

$$s(C_i) \cap s(H) \subseteq s(h_i),\ i = 1, \ldots, m. \tag{B4}$$

Fact B.1. If x is a variable not in $s(H)$, then x belongs to exactly one connected component C_i.

Proof: Suppose for contradiction that $x \notin s(H)$ and $x \in s(C_i) \cap s(C_j)$. Then $x \in s(c) \cap s(c')$ for some $c \in C_i$, $c' \in C_j$. The fact that $s(c) \cap s(c') \neq \emptyset$ then implies that (c, c') is an edge in the dual graph of B, and $x \in l(c, c')$. But then $l(c, c') \notin s(H)$, and therefore C_i, C_j are connected; i.e., they are not two distinct connected components of $R - H$ with respect to H.

Fact B.1 allows us to define $\lambda_x^H : L_H \to T(x)$ for $x \in X - s(H)$ by

$$\lambda_x^H = \lambda_x \bar{\lambda}_{h_i} \lambda_{h_i}^H,\ x \in X - s(H),\ x \in s(C_i) \tag{B5}$$

where h_i is given by (B4). Similarly, if c is a constraint in $R - H$, c belongs to a unique C_i, hence define $\lambda_c^H : L_H \to T(c)$ by

$$\lambda_c^H = \lambda_c \bar{\lambda}_{h_i} \lambda_{h_i}^H,\ c \in R - H,\ c \in C_i. \tag{B6}$$

Fact B.2. The family of maps (L_H, λ_a^H), as extended by (B5) and (B6) to all nodes a in G_B, is a commutative cone with base T, i.e.

$$\lambda_x^H = T(\pi_x^c)\, \lambda_c^H \text{ for each } \pi_x^c : c \to x \text{ in } G_B. \tag{B7}$$

Proof: (B7) is true by definition for c in H. For c in $R - H$, there are two cases for each x in $s(c)$: $x \in s(H)$, or $x \notin s(H)$.

Case 1. $c \in R - H$, $x \in s(H)$.

Since $x \in s(c)$, and since c belongs to a unique C_i, we have

$$x \in s(c) \cap s(H) \subseteq s(C_i) \cap s(H) \overset{(B4)}{\subseteq} s(h_i),\ h_i \in H.$$

Hence, by definition of $\langle L, \lambda_a^H \rangle$ as the limit of T_H,

$$\lambda_x^H = T(\pi_x^{h_i})\, \lambda_{h_i}^H, \text{ since } \pi_x^{h_i} : h_i \to x \text{ is in } G_H.$$

On the other hand, by (B6),

$$T(\pi_x^c)\, \lambda_c^H = T(\pi_x^c)\, \lambda_c \bar{\lambda}_{h_i} \lambda_{h_i}^H.$$

To show (B7), therefore, it suffices to show

$$T(\pi_x^{h_i}) = T(\pi_x^c)\, \lambda_c \bar{\lambda}_{h_i}, \text{ or, recalling that } \lambda_{h_i} \bar{\lambda}_{h_i} = id,$$

$$T(\pi_x^{h_i})\, \lambda_{h_i} = T(\pi_x^c)\, \lambda_c.$$

This last equality is true by the commutativity property of (L, λ_a), since both sides equal λ_x.

Case 2. $c \in R - H$, $x \notin s(H)$.

Since $x \in s(c)$ and $c \notin H$, x must belong to $s(C_i)$, where C_i is the (unique) connected component of $R - H$ w.r.t. H that contains c. Then, λ_x^H and λ_c^H are defined by (B5) and (B6), respectively, and

$$\lambda_x^H \overset{(B5)}{=} \lambda_x \bar{\lambda}_{h_i} \lambda_{h_i}^H \overset{(A2)}{=} T(\pi_x^c)\, \lambda_c \bar{\lambda}_{h_i} \lambda_{h_i}^H \overset{(B6)}{=} T(\pi_x^c)\, \lambda_c^H.$$

Fact B.3. There is unique $g : L_H \to L$ *such that*

$$\lambda_x^H = \lambda_x g, \quad x \text{ in } X \tag{B8}$$

$$\lambda_c^H = \lambda_c g, \quad c \text{ in } R. \tag{B9}$$

Proof: (L, λ_a) is the limit of T, and, by Fact B.2, (L_H, λ_a^H) is a commutative cone with base T.

Fact B.4. $f g = id_{L_H}$.

Proof: Recall that $(L_H, \lambda_a^H : a \in s(H) \cup H)$ is a limit of T_H, the restriction of T on G_H. By the joint monic property of limit cones (Fact A.3), to show $f g = id_{L_H}$ it suffices to show

$$\lambda_x^H f g = \lambda_x^H, \quad \text{all } x \text{ in } s(H), \tag{B10}$$

$$\lambda_c^H f g = \lambda_c^H, \quad \text{all } c \text{ in } H. \tag{B11}$$

(B10) follows from (B8) and (B3), whereas (B11) follows from (B9) and (B2).

Facts B.3 and B.4 show that every solution of H can be extended to a solution of B; the (constructed) function g provides such an extension.

Appendix C

To prove Fact 5.2, the following fact is useful.

Fact C.1. Let $f : A \to B$ *be a function, and let* $f[A]$ *be its range in B. Then there exists an onto function* $e : A \to f[A]$ *and a one-to-one function* $m : f[A] \to B$ *such that* $f = me$.

Proof: Let $e(a) = f(a)$, $m(b) = b$, $(a \in A,\ b \in f[A])$. The pair (e, m) is called an image factorization of f.

Proof of Fact 5.2. Since H is not a hinge, there exists a connected component, say C_1, of $R - H$ with respect to H such that $s(C_1) \cap s(H) \subseteq s(h)$ for all h in H.

Let $Y = s(C_1) \cap s(H)$; Y is clearly non-empty. The domains of the CSP to be defined are taken to be

$$T(x) = Y \text{ if } x \in s(C_1),\ T(x) = \{b\} \text{ if } x \in X - s(C_1) \tag{C1}$$

where b is any element not in Y.

To define the extent of each constraint c, let $\pi : Y \to Y$ be any isomorphism such that $\pi(y) \neq y$ for all y in Y; and let $\delta : Y \to \prod_{x=1}^{n} T(x)$ be defined by

$$\delta(y)(x) = x \text{ if } x \in Y,\ x \neq y; \tag{C2}$$

$$\delta(y)(x) = \pi(y) \text{ if } x = y \text{ or } x \in s(C_1) - s(H);$$

$$\delta(y)(x) = b \text{ if } x \in X - s(C_1).$$

Let $L = \delta[Y]$, and take (e, λ) to be an image factorization of δ. Hence, by Fact C.1,

$$\delta = \lambda e, \tag{C3}$$

$$id_L = e\bar{e} \tag{C3'}$$

($\bar{e}$ is a partial inverse of the onto function e).

Let $p_x : \prod_{x=1}^{n} T(x) \to T(x)$, $p^c : \prod_{x=1}^{n} T(x) \to \prod_{x \in s(c)} T(x)$, and $p_x^c : \prod_{x \in s(c)} T(x) \to T(x)$ be projections. Then define

$$T(c) = p^c\lambda[L] = \text{projection of } L \text{ on } s(c), \tag{C4}$$

$$(\lambda_c, m_c) \text{ an image factorization of } p^c\lambda, \text{ so that } \lambda_c : L \to T(c),$$

$$m_c : T(c) \to \prod_{x \in s(c)} T(x) \text{ and} \tag{C5}$$

$$p^c\lambda = m_c\lambda_c, \tag{C5'}$$

$$\lambda_c\bar{\lambda}_c = id_{T(c)}, \text{ where } \bar{\lambda}_c \text{ is a partial inverse of } \lambda_c. \tag{C5''}$$

$$\lambda_x : L \to T(x),\ \lambda_x = p_x\lambda, \tag{C6}$$

$$T(\pi_x^c) : T(c) \to T(x),\ T(\pi_x^c) = p_x^c m_c,\ x \in s(c). \tag{C7}$$

The CSP defined by T has the property that any solution $u \in T(c)$ of any constraint c is extendable to a member of L, namely $\bar{\lambda}_c(u)$. The proof of Fact 5.2 consists of two parts. First, $(L, \lambda_c, \lambda_x)$ is shown to be a limit of T; this implies that L is the solution set of the CSP defined by T, and that every solution of every constraint is extendable (via $\bar{\lambda}_c$) to a full solution. Secondly, it is shown that the subproblem defined by H has a solution not extendable to any element of L.

Fact C.2. $(L, \lambda_c, \lambda_x)$ *is a commutative cone with base* T.

Proof: $T(\pi_x^c)\lambda_c =^{(C7)} p_x^c m_c\lambda_c =^{(C5')} p_x^c p^c\lambda = p_x\lambda =^{(C6)} \lambda_x$.

Fact C.3. $(L, \lambda_c, \lambda_x)$ *is a limit of* T.

Proof: Let (M, μ_c, μ_x) be a commutative cone with base T. Then, by definition of commutativity,

$$\mu_x = T(\pi_x^c)\mu_c. \tag{C8}$$

I have to show that there is a unique function $f: M \to L$ such that $\mu_x = \lambda_x f$, $\mu_c = \lambda_c f$. Uniqueness is easy: suppose that another function h also satisfies these equations. Then $\lambda_x f = \mu_x = \lambda_x h$, all x, or by (C6) $p_x \lambda f = p_x \lambda h$, all x. Hence, $\lambda f = \lambda h$ because the p_x are projections, and $f = h$ because λ is one-to-one. To show the existence of f, let $\mu: M \to \prod_{x=1}^{n} T(x)$ be uniquely defined by

$$p_x \mu = \mu_x, \ x \text{ in } X. \tag{C9}$$

Fact C.4 will show that there is a function $g: M \to Y$ such that $\delta g = \mu$. The function $f =^{\text{def.}} eg$, then, satisfies $\mu_x = \lambda_x f$, $\mu_c = \lambda_c f$. To see the second equation, for example, let $x \in s(c)$. Then $p_x^c m_c \lambda_c f =^{\text{def}} p_x^c m_c \lambda_c eg =^{(C5')} p_x^c p^c \lambda eg = p_x \lambda eg =^{(C3)} p_x \delta g = p_x \mu =^{(C9)} \mu_x =^{(C8)} T(\pi_x^c) \mu_c =^{(C7)} p_x^c m_c \mu_c$. The fact that $\lambda_c f = \mu_c$ then follows from the facts that p_x^c are projections and m_c is one-to-one. Similarly for $\mu_x = \lambda_x f$.

I now show the existence of g.

Fact C.4. There is a function $g: M \to Y$ such that $\delta g = \mu$.

Proof: There are two cases to be distinguished. The first is $s(C_1) - s(H) \neq \emptyset$, and the second $s(C_1) - s(H) = \emptyset$. The proof for the case $s(C_1) - s(H) \neq \emptyset$ consists of several lemmas.

Lemma 1. If $s(C_1) - s(H) \neq \emptyset$, then $s(c) - s(H) \neq \emptyset$ for each constraint c in C_1.

To see this, suppose (for contradiction) that $s(c) - s(H) = \emptyset$ for some c in C_1. Then there is (at least) another constraint c' in C_1 with $s(c') - s(H) \neq \emptyset$. Recall that C_1 is a connected component of $R - H$ w.r.t. H. There is, then, a sequence $c = c_1, c_2, \ldots, c_n = c'$ of constraints in C_1 such that $s(c_i) \cap s(c_{i+1}) - s(H) \neq \emptyset$ (see Def. 5.2 in the text). But then $\emptyset \neq s(c_1) \cap s(c_2) - s(H) = s(c) \cap s(c_2) - s(H) \subseteq s(c) - s(H) = \emptyset$, a contradiction.

For each c in C_1, then, define $g_c: M \to Y$ by

$$g_c = \bar{e} \bar{\lambda}_c \mu_c, \ c \text{ in } C_1. \tag{C10}$$

I show that $g_c = g_q$ for any two constraints q, c in C_1, and that by setting $g = g_c$ for some c in C_1, $\delta g = m$.

Lemma 2. Let v be in M and $y_c = g_c(v)$. For each c in C_1 and each x in $s(c)$, $\mu_x(v) = x$ if $x \in Y - \{y_c\}$, $\mu_x(v) = \pi(y_c)$ if $x = y_c$ or $x \in s(C_1) - s(H)$, and $\mu_x(v) = b$ if $x \notin s(C_1)$. In other words, $\mu_x(v) = p_x \delta(y_c)$.

Proof: $\mu_x(v) =^{(C8)} T(\pi_x^c) \mu_c(v) =^{(C7)} p_x^c m_c \mu_c(v) =^{(C5'')} p_x^c m_c \lambda_c \bar{\lambda}_c \mu_c(v) =^{(C3')} p_x^c m_c \lambda_c e \bar{e} \bar{\lambda}_c \mu_c(v) =^{(C10)} p_x^c m_c \lambda_c e g_c(v) =^{\text{def.}} p_x^c m_c \lambda_c e(y_c) =^{(C5')} p_x^c p^c \lambda e(y_c) = p_x \lambda e(y_c) =^{(C3)} p_x \delta(y_c)$. The lemma then follows from (C2).

Lemma 3. If c, q are in C_1 and $x \in s(c) \cap s(q) - s(H)$, then $y_c = y_q$.

Proof: By Lemma 2, $\pi(y_c) = \mu_x(v) = \pi(y_q)$. Since π is an isomorphism, $y_c = y_q$.

Lemma 4. If c, q are any two constraints in C_1, then $y_c = y_q$ and therfore $g_c = g_q$.

Proof: Since C_1 is a connected component of $R - H$ w.r.t. H, there is a sequence $c = c_1, c_2, \ldots, c_n = q$ such that $s(c_i) \cap s(c_{i+1}) - s(H) \neq \emptyset$. Let $x_i \in s(c_i) \cap s(c_{i+1}) - s(H)$, $i = 1, 2, \ldots, n-1$. By Lemma 3, $y_{c_i} = y_{c_{i+1}}$, $i = 1, 2, \ldots, n, -1$. Then, $y_c =^{\text{def.}} y_{c_1} = y_{c_2} = \ldots = y_{c_n} =^{\text{def.}} y_q$.

Lemma 5. Let $g : M \to Y$ be defined by $g = g_c$, c in C_1. Then $\mu = g\delta$.

Proof: I show $p_x \mu = p_x g \delta$ for all x in X. If $x \in X - s(C_1)$, then $p_x \mu(v) =^{(C9)} \mu_x(v) = b = p_x g \delta(v)$ because the domain of x is $\{b\}$ in this case. If $x \in s(C_1)$, then $x \in s(c)$ for some c in C_1. Then, $p_x \mu =^{(C9)} \mu_x =^{(C8)} T(\pi_x^c)\mu_c =^{(C7)} p_x^c m_c \mu_c$
$=^{(C5'')} p_x^c m_c \lambda_c \bar{\lambda}_c \mu_c$

$$\overset{(C3')}{=} p_x^c m_c \lambda_c e \tilde{e} \bar{\lambda}_c \mu_c \overset{(C10)}{=} p_x^c m_c \lambda_c e g_c \overset{(C5')}{=} p_x^c p^c \lambda e g_c$$

$$= p_x \lambda e g_c \overset{(C3)}{=} p_x \delta g_c = p_x \delta g.$$

This concludes the proof of Fact C.4, and therefore of Fact 5.2 as well in the case $s(C_1) - s(H) \neq \emptyset$. The case $s(C_1) - s(H) = \emptyset$ is easier, due to the following lemma.

Lemma 6. If $s(C_1) - s(H) = \emptyset$, then C_1 contains only one constraint, i.e. $C_1 = \{c\}$.

Proof: Suppose C_1 contained more than one constraint. For any c, q in C_1, then, there is a sequence $c = c_1, c_2, \ldots, c_n = q$ in C_1 such that $s(c_i) \cap s(c_{i+1}) - s(H) \neq \emptyset$, all i. This implies $\emptyset \neq s(c_i) \cap s(c_{i+1}) - s(H) \subseteq s(C_1) - s(H) = \emptyset$, a contradiction.

The proof of Fact C.4 is now completed by first setting $g = g_c$, where $C_1 = \{c\}$ by Lemma 6, and then repeating the steps in Lemma 5 to show $\mu = g\delta$.

To conclude the proof of Fact 5.2, I show that H has a nonextendable solution, namely the tuple u defined by $u_x = x$ for $x \in Y \equiv s(C_1) \cap s(H)$ and $u_x = b$ for x in $s(H) - s(C_1)$. Since $u_x = x$ for all x in Y, u is not extendable to a point in L; hence, given that L is by construction the solution set of the CSP, u is nonextendable to a full solution. To see that u solves H, note that for each h in H, $Y \equiv s(C_1) \cap s(H) \not\subset s(h)$; hence $Y - s(h) \neq \emptyset$. Consider the vector $\delta(y)$, for

$y \in Y - s(h)$. It follows that for each h in H $u_x = \delta(y)(x)$ for all x in $s(h)$, because $u_x = \delta(y)(x) = x$ for $x \in s(h) \cap Y$ and $u_x = \delta(y)(x) = b$ for each $x \in s(h) - Y$. Hence, by construction of $T(h)$, u restricted to $s(h)$ belongs to $T(h)$ for all h in H; i.e., u solves H.

References

Baily, M. and H. Gersbach, 1995, Efficiency in manufacturing and the need for global competition, Brookings Papers: Microeconomics 307–358.

Clark, K. and T. Fujimoto, 1991, Product Development Performance (Harvard Business School Press, Boston, MA).

Clausing, D., 1994, Total Quality Development (ASME Press, New York).

Cormen, T., C. Leiserson and R. Rivest, 1990, An introduction to algorithms (MIT Press, Boston, MA).

Dechter, R. and J. Pearl, 1989, Tree clustering for constraint networks, Artificial Intelligence 38, 353–366.

Gomory, M., 1989, From the product cycle to the ladder of science, Harvard Business Review 89, (6) 99–105.

Gyssens, M., 1986, On the complexity of join dependencies, ACM Transactions on Database Systems 11, (1) 81–108.

Gyssens, M., P. Jeavons and D. Cohen, 1994, Decomposing constraint satisfaction problems using database techniques, Artificial Intelligence 66, 57–89.

Hauser, J. and D. Clausing, 1988, The house of quality, Harvard Business Review (May–June) 63–73.

Mackworth, A., 1992, The logic of constraint satisfaction, Artificial Intelligence 58, 3–20.

Solow, R., 1994, Perspectives on growth theory, Journal of Economic Perspectives 8, (1) 45–54.

Taguchi, K. and D. Clausing, 1990, Robust quality, Harvard Business Review 90, 65–75.

Ulrich, K. and S. Eppinger, 1995, Product Design and Development (McGraw Hill, New York).

Ward, A., D. Sobek, J. Cristiano and J. Liker, 1995, The second Toyota paradox, Sloan Management Review Spring issue 43–61.

Wheelwright, R. and M. Clark, 1992, Revolutionizing Product Development (Harvard Business School Press, Boston, MA).

Womack, R., D. Jones and D. Roos, 1989, The Machine that Changed the World (Rawson Associates, New York).