Toward a Classification Approach to Design

Douglas R. Smith

Kestrel Institute, 3260 Hillview Avenue, Palo Alto, California 94304, USA
smith@kestrel.edu
18 March 1996

Abstract. This paper addresses the problem of how to construct refinements of specifications formally and incrementally. The key idea is to use a taxonomy of abstract design concepts, each represented by a *design theory*. An abstract design concept is applied by constructing a specification morphism from its design theory to a requirement specification. Procedures for propagating constraints, computing colimits, and constructing specification morphisms provide computational support for this approach. Although we conjecture that classification generally applies to the incremental application of knowledge represented in a taxonomy of design theories, this paper mainly focuses on algorithm design theories and presents several examples of design by classification.

1 Introduction

Mathematically-based techniques for software construction will play an increasing, if not critical, role in the future of software engineering. This paper is part of a broader research program to explore a mechanizable model of software development based on algebraic specifications and specification morphisms. An algebraic specification (or simply a *specification* or *theory*) defines a language and constrains its possible meanings via axioms and inference rules. Specifications can be used to express many kinds of software-related artifacts, including application domain models [22], formal requirements [1, 5, 14], abstract data types [7, 10], abstract algorithms [17], and programming languages [3, 9, 11]. A *specification morphism* (or simply a *morphism*) translates the language of one specification into the language of another specification in a way that preserves theorems. Specification morphisms underlie several aspects of software development, including the binding of parameters in parameterized specifications [4, 9], specification refinement and implementation [2, 15, 24], and algorithm design [12, 17, 25].

Despite years of research on specification languages and specification refinement, there has been relatively little work on formal techniques for constructing refinements, as opposed to verifying refinements that have been written manually. This paper addresses the following overall problem: given a specification S_0, construct a specification morphism $J : S_0 \rightarrow S_1$ that refines S_0 by using preexisting knowledge about standard generic designs.

Software often can be explained in terms of a relatively small collection of abstract design concepts. Examples of design abstractions include divide-and-conquer as an algorithm abstraction, heaps as a data structure abstraction, and a standard tracking architecture as a software system abstraction. An abstract design concept corresponds to a class of artifacts and the common structure of the class can be represented as a specification, called an *artifact theory*. Consider a class of related artifacts $\mathcal{A}$. Elements of $\mathcal{A}$ can be described via normal-form expressions in an appropriate artifact description language. Then, abstracting out those sorts and operations that vary over the elements we obtain the language of a *design theory* DT for $\mathcal{A}$ and an artifact scheme with free sort and operator symbols. The axioms of DT arise as conditions under which the sorts and operations can be instantiated in the artifact scheme to yield a correct concrete artifact. A specification AT containing the artifact scheme and parameterized on the design theory is called an *artifact theory*.

Greedy algorithms provide a particularly clear and well-known example of these concepts. Many algorithm design texts give a program scheme for greedy algorithms. If the free operators of the scheme have matroid structure, then the corresponding instance of the greedy scheme is provably correct with respect to its optimization objective [13]. Here, matroid theory is the design theory, and the artifact theory is parameterized on matroid theory and contains the greedy scheme.

$$\begin{tikzcd} DT \arrow[r, "I"] \arrow[d] & S_0 \arrow[d, "J"] \\ AT \arrow[r] & S_1 \end{tikzcd}$$

The diagram to the left shows how a design and artifact theory can be used to construct a refinement of a requirement specification S_0. The hard work in design is constructing a *classification arrow* (a morphism or interpretation between theories) I from the design theory DT to S_0, which explicates the $\mathcal{A}$-structure of S_0, or *classifies* S_0 as an $\mathcal{A}$-structure. Given I and $DT \longrightarrow AT$, then the refinement $J : S_0 \rightarrow S_1$ is automatically generated via a colimit construction which instantiates the parameter to AT.

In Section 3 a design theory for the algorithmic concept of divide-and-conquer is presented. Any particular divide-and-conquer algorithm corresponds to an interpretation from divide-and-conquer theory to a specification of the particular problem being solved. In particular, various interpretations from divide-and-conquer theory to a sorting specification correspond to various sorting algorithms, such as quicksort, mergesort or Batcher's sort. Given such an interpretation, a concrete sorting algorithm is obtained by instantiating a divide-and-conquer scheme with the translations of the symbols in divide-and-conquer theory.

Design theories can be arranged in a refinement hierarchy with specification morphisms providing the refinement links; e.g. a hierarchy of algorithm theories is presented in Figure 4 (see [17, 25]). *The main technical focus of this paper is showing how a refinement hierarchy of design theories supports incremental construction of refinements.*

The concepts and procedures described below are intended to improve the practicality of machine support for formal software development. This work is based on experience with algorithm design and optimization using KIDS (Kestrel Interactive Development System) which has been used to design over 70 algorithms from formal specifications [18]. Currently in KIDS, algorithm design is carried out by specialized procedures for each class of algorithms, called *design tactics* [17]. Classification allows us to duplicate and extend the functionality of the KIDS algorithm design tactics. Classification is being implemented in the successor to KIDS, called Specware [23]. We conjecture that classification will also support a much broader range of design tasks, such as the design of data structures, user interfaces, and software systems.

After reviewing basic concepts and notation in Section 2, the classification method is presented in Section 3. Two examples are presented in Section 4.

2 Basic Concepts and Notations

2.1 Specifications

As much as possible we adhere to conventional concepts and notation for first-order algebraic specification [6, 8, 26]. A *signature* $\Sigma = \langle S, \Omega \rangle$ consists of a set of sort symbols S and a family $\Omega = \langle \Omega_{v,s} \rangle$ of finite disjoint sets indexed by $S^* \times S$, where $\Omega_{v,s}$ is the set of operation symbols of rank $\langle v, s \rangle$. We write $f : v \rightarrow s$ to denote $f \in \Omega_{v,s}$ for $v \in S^*$, $s \in S$ when the signature is clear from context. As far as possible in this paper we treat truth-values as any other sort. Letting *boolean* be the sort symbol for truth values, then $\Omega_{v,boolean}$ is a set of predicate symbols for each $v \in S^*$. The usual logical connectives $\wedge$, $\vee$, $\neg$, $\Longrightarrow$, and $\Longleftrightarrow$ are treated as boolean operations. For any signature Σ, the *Σ-terms* are defined inductively in the usual way as the well-sorted composition of operator symbols and variables. A *Σ-formula* is a boolean-valued term built from Σ-terms and the quantifiers $\forall$ and $\exists$. A *Σ-sentence* is a closed formula. The generic term *expression* is used to refer to a term, formula, or sentence. A *specification* $T = \langle S, \Omega, Ax \rangle$ comprises a signature $\Sigma = \langle S, \Omega \rangle$ and a set of Σ-sentences Ax called *axioms*. Specification $T' = \langle S', \Omega', Ax' \rangle$ *extends* specification $T = \langle S, \Omega, Ax \rangle$ if $S \subseteq S'$, $\Omega_{v,s} \subseteq \Omega'_{v,s}$ for each $v \in S^*$, $s \in S$, and $Ax \subseteq Ax'$. Alternatively, we say T' is an *extension* of T. A *model* for T is a structure for $\langle S, \Omega \rangle$ that satisfies the axioms. We shall use modus ponens, substitution of equals/equivalents and other natural deduction *rules of inference* in T. A sentence e is a *theorem* of T, written $\vdash_T e$, if e is in the closure of the axioms under the rules of inference.

2.2 Morphisms

A *signature morphism* $I : \langle S, \Omega \rangle \rightarrow \langle S', \Omega' \rangle$ maps S to S' and Ω to Ω' such that the ranks of operations are preserved: if $f : v \rightarrow s$ in Ω and $v = v_1, \ldots v_n$ then $I(f) : I(v_1) \ldots I(v_n) \rightarrow I(s)$ in Ω'. A signature morphism extends in a unique way to a translation of expressions (as a homomorphism between term algebras) or sets of expressions. For Σ-expression e, let $I(e)$ denote its translation to a Σ'-expression. For a set of Σ-expressions E, let $I(E)$ denote the set of Σ'-expressions $\{I(e) \mid e \in E\}$. The notion of a signature morphism can be extended to a specification morphism by requiring that the translation preserve theorems. Let $T = \langle S, \Omega, Ax \rangle$ and $T' = \langle S', \Omega', Ax' \rangle$ be specifications and let $I : \langle S, \Omega \rangle \rightarrow \langle S', \Omega' \rangle$ be a signature morphism between them. I is a *specification morphism* if for every axiom $A \in Ax$, $I(A)$ is a theorem of T': $\vdash_{T'} I(A)$. It is straightforward to show that a specification morphism translates theorems of the source specification to theorems of the target specification.

Specifications and specification morphisms form a category. Colimits exist in this category and are easily computed.

The semantics of a specification morphism is given by a model construction. If $I : T1 \rightarrow T2$ is a specification morphism, then every model $\mathcal{M}$ of $T2$ can be made into a model of $T1$ by simply "forgetting" some structure of $\mathcal{M}$.

It will be convenient to generalize the definition of signature morphism slightly so that the translations of operator symbols are allowed to be expressions in the target specification and the translations of sort symbols are allowed to be constructions (e.g. products) over the target sorts. A symbol-to-expression morphism is called an *interpretation between theories* (or simply, an *interpretation*). An interpretation, notated $I : A \implies B$ or $A \stackrel{I}{\implies} B$, can be represented as a morphism into an extension by definitions of the target specification; i.e. as a pair of morphisms $A \rightarrow A\text{-}B \leftarrow B$. The specification $A\text{-}B$ is called the *mediator* of the interpretation. Composition of interpretations is straightforward.

2.3 Examples of Specifications and Morphisms

A *problem* P consists of a set of possible inputs $x \in D$ such that *input condition* $I(x)$ holds, and a set of outputs (also called *feasible solutions*) $z \in R$ such that some *output condition* $O(x, z)$ holds. A *problem specification* can be presented in the following format

Spec *Problem-Theory*
  sorts D, R
  op $I : D \rightarrow Boolean$
  op $O : D \times R \rightarrow Boolean$
end-spec

A concrete problem can be presented via an interpretation from *Problem-Theory* into the specification of the problem. For example, consider the problem of sorting a bag of integers. A specification for sorting defines the concepts and laws necessary to support the definition of the sorting problem. The following domain specification is parameterized on a linear order — given any particular set S that is linearly ordered by $\leq$ we obtain a concrete sorting specification.

Spec *Sorting-Theory* $(\langle S, \leq \rangle :: \mathit{Linear\text{-}Order})$
    imports $\mathit{seq\text{-}over\text{-}linear\text{-}order}(\langle S, \leq \rangle)$,
        $\mathit{bag\text{-}over\text{-}linear\text{-}order}(\langle S, \leq \rangle)$
    op $\mathit{ordered} : \mathit{seq}(S) \to \mathit{boolean}$
    op $\mathit{bagify} : \mathit{seq}(S) \to \mathit{bag}(S)$
    op $\mathit{Sorting} \quad (x : \mathit{bag}(S) \mid \mathit{true})$
        returns $(z : \mathit{seq}(S) \mid \mathit{ordered}(z) \ \wedge \ x = \mathit{bagify}(z))$
    axioms ... axioms defining the operations ...
end-spec

Here a program-like format is used for specifying problems:

op $f \ (x : D \mid I(x))$
    returns $(z : R \mid O(x, z))$

which can be regarded as syntactic sugar for the following signature and axiom

op $f : D \to R$
ax $\forall (x : D)(I(x) \implies O(x, f(x)))$

We can present the sorting problem via an interpretation from *Problem-Theory* to *Sorting*:

$$\begin{array}{lcl} D & \longmapsto & \mathit{bag}(S) \\ I & \longmapsto & \lambda(x)\, \mathit{true} \\ R & \longmapsto & \mathit{seq}(S) \\ O & \longmapsto & \lambda(x, z)\, (\mathit{ordered}(z) \ \wedge \ x = \mathit{bagify}(z) \end{array}$$

Again this interpretation is represented as a pair of morphisms. The mediator is a definitional extension to *Sorting-Theory* with the new symbols and definitional axioms

op $I_m : \mathit{seq}(S) \to \mathit{boolean}$
op $O_m : \mathit{seq}(S) \times \mathit{bag}(S) \to \mathit{boolean}$
def $I_m(x) \ = \ \mathit{true}$
def $O_m(x, z) \ = \ \mathit{ordered}(z) \ \wedge \ x = \mathit{bagify}(z)$

The morphism from *problem-theory* to the mediator is

$$\begin{array}{lcl} D & \longmapsto & \mathit{bag}(S) \\ I & \longmapsto & I_m \\ R & \longmapsto & \mathit{seq}(S) \\ O & \longmapsto & O_m \end{array}$$

In the following we will present interpretations in the more convenient symbol-to-expression format.

3 Classification Approach to Design

The hard work in the classification approach to design lies in constructing interpretations from design theories to requirement specifications; that is, constructing classification arrows. There are two problems related to constructing classification arrows: (1) selecting an appropriate design theory, and (2) constructing an interpretation. Suppose that we have a refinement hierarchy of design theories with specification morphisms providing the refinement links. The refinement hierarchy provides a framework for solving the selection problem, and simultaneously providing a way to construct classification arrows incrementally. The stronger a theory is, the more structure that can be exploited in an artifact theory. Consequently, we want to construct an interpretation from the deepest possible theories in the hierarchy to the given requirement specification. This suggests an incremental procedure for accessing a hierarchic library of design theories and for constructing classification arrows. First, construct an interpretation from the root theory of the hierarchy. Then, iteratively, given an interpretation from design theory DT, try to construct an interpretation from DT's refinements in the hierarchy. If several succeed, then select one or keep several and repeat the process. If none succeed, then the current design theory exploits as much of the problem structure as possible (with respect to this classification hierarchy). A concrete design can be obtained by instantiating an artifact theory parameterized on the design theory.

$$
\begin{array}{ccc}
DT_0 & \xRightarrow{I_0} & Spec_0 \\
\downarrow & & \downarrow \\
DT_1 & \xRightarrow{I_1} & Spec_1 \\
\downarrow & & \downarrow \\
DT_2 & \xRightarrow{I_2} & Spec_2 \\
\downarrow & & \downarrow \\
\vdots & & \vdots \\
DT_n & \xRightarrow{I_n} & Spec_n
\end{array}
$$

Ladder Construction

The process of incrementally constructing an interpretation is illustrated in the *ladder construction* diagram to the left. The left-hand side of the ladder is a path in a refinement hierarchy of design theories starting at the root. The ladder is constructed a rung at a time from the top down. The initial interpretation from problem theory to $Spec_0$ may be simple to construct. Subsequent rungs of the ladder are constructed by a constraint solving process that involves user choices, the propagation of consistency constraints, calculation of colimits, and constructive theorem proving [20].

Once we have constructed a classification arrow, then constructing a refinement of $Spec_0$ is straightforward. Elaborating a little on the presentation in the introduction, if we have a classification arrow $DT_n \xRightarrow{I_n} Spec_n$ represented by the pair of morphisms $DT_n \xrightarrow{M_n} \text{DT-Spec}_n \xleftarrow{d_n} S_n$, and an artifact theory AT_n that is parameterized on DT_n, then we can mechanically calculate a morphism that refines $Spec_n$ (by computing the pushout and then composing J_n and d_n):

$$
\begin{tikzcd}
DT_n \arrow[r, "M_n"] \arrow[d] \arrow[dr, phantom, "p.o."] & DT\text{-}Spec_n \arrow[d, "J_n"'] & Spec_n \arrow[l, "d_n"'] \arrow[dl, "J_n \circ d_n"] \\
AT_n \arrow[r] & AT\text{-}Spec_n &
\end{tikzcd}
$$

Finally, by composition we construct the refinement morphism

$$Spec_0 \longrightarrow AT\text{-}Spec_n$$

whose codomain contains the artifact specified in $Spec_0$.

The incremental situation in rung construction is this: given I_i and $DT_i \longrightarrow DT_{i+1}$

$$
\begin{tikzcd}
DT_i \arrow[r, "M_i"] \arrow[d] & DT\text{-}Spec_i \arrow[d, dotted] & Spec_i \arrow[l, "d_i"'] \arrow[d, dotted] \\
DT_{i+1} \arrow[r, dotted, "M_{i+1}"] & DT\text{-}Spec_{i+1} & Spec_{i+1} \arrow[l, dotted]
\end{tikzcd}
$$

construct specifications $DT\text{-}Spec_{i+1}$ and $Spec_{i+1}$, and the dotted-line morphisms.

It is unlikely that a general automated method exists for constructing rungs. At present it seems that users must be involved in guiding the rung construction process. Our intent is to build an interface providing the user with various generic automated operations and libraries of standard components. The user applies various operators with the goal of filling out partial morphisms and specifications until the rung is complete. After each user-directed operation, various constraint propagation rules are automatically invoked to perform sound extensions to the partial morphisms and specifications in the rung diagram.

Strategies for rung construction vary according to the particular refinement $DT_i \rightarrow DT_{i+1}$. Different design theories call for different methods for constructing the morphism M_{i+1}. The construction of $Spec_{i+1}$ is mainly driven by the construction M_{i+1}. Typically $Spec_{i+1}$ is a conservative extension of $Spec_i$; however, in some cases arising during algorithm design (global search in particular), $Spec_{i+1}$ is a (conservative) extension of a colimit involving a refinement of $Spec_i$.

There are several obvious user-directed actions, including supplying translations for symbols and invoking a prover to verify that axioms translate to theorems. The user may translate a symbol into existing or new symbols, in which case the codomain specification $Spec_{i+1}$ is extended. The user may also elect to translate imported specifications via preexisting interpretations.

Constraint propagation rules are based on constraints that are generic to rung construction, such as commutativity of the rung diagram and preservation of operator rank under a morphism. Initially, we can propagate the symbol translations in I_i through $DT_i \xrightarrow{K_i} DT_{i+1}$. For example, if symbol a in DT_i is translated to b ($I_i : a \mapsto b$), then let $I_{i+1} : K_i(a) \mapsto b$. If two symbols of DT_i translate to the same symbol in DT_{i+1}, then they are effectively equated by the morphism, so the translation of either one will suffice.

The constraint that morphisms must preserve signatures under translations yields various propagation rules. If $I_i : f \mapsto g$ where $f : D \to R$ and $g : E \to S$, then propagation rules can add the translations $I_i : D \mapsto E$ and $I_i : R \mapsto S$.

There are several mechanizable techniques for constructing morphisms [20], one of which, called *unskolemization*, will be discussed here. The key idea in unskolemization is to use the axioms of the source specification as constraints on the translations of source symbols. Theorem-proving techniques are used to deduce symbol translations such that the source axioms are properly translated.

Skolemization is the process of replacing an existentially quantified variable z with a Skolem function over the universally quantified variables whose scope includes z. For example, the formula

$$\exists(w)\forall(x, y)\exists(z)\forall(u)H(w, x, y, z, u) \tag{1}$$

is skolemized to

$$\forall(x, y)\forall(u)H(a, x, y, f(x, y), u) \tag{2}$$

where f is a Skolem function of x and y and a is a Skolem function of no arguments – a Skolem constant. A *simple occurrence* of an operation symbol $g : v \to s$ in a sentence G is a subexpression of G of the form $g(x)$ where $x : v$ is a sequence of distinct variables that are universally quantified in G. Skolemization always replaces an existentially quantified variable with simple occurrences of a fresh operation symbol.

We are interested in the inverse process, *unskolemization*: given a sentence (such as (2)) containing identical simple occurrences of operation symbol g, say $g(x)$, replace g by a fresh existentially quantified variable in the scope of x (such as (1)).

Suppose that we are trying to complete a partial specification morphism σ from specification T to specification S. Let $f : v \to s$ be a function symbol of T that has no translation yet under σ. Suppose that F is a prenex normal form axiom in which all occurrences of f are identical and simple, and suppose that all other symbols in F are translatable under σ (i.e. the domain of σ includes all of the sort and operator symbols of F except for the function symbol f). To obtain a candidate translation for a function symbol f, we proceed as follows.

(1) *Unskolemize f in F yielding F'.* Since this has the effect of replacing each occurrence of f by a variable, each symbol in F' can be translated via σ.
(2) *Translate F'.* The translated sentence $\sigma(F')$ need not be an axiom of S. In order for σ to become a specification morphism we need an expression defining the translation of f in S. $\sigma(F')$ can be viewed as a constraint on the possible translations of f.
(3) *Attempt to prove $\sigma(F')$ in S.* A constructive proof will yield a (witness) expression $t(x)$ for f that depends only on the variables x. If the proof involves induction (resulting in a recursively defined witness), then we extend

the target specification with a fresh operator symbol and an axiom stating its recursive definition.

(4) *Extend the partial morphism σ by defining $\sigma(f)$ to be $t(x)$.* By construction this translation for f guarantees that σ properly translates the axiom F.

Other axioms that involve f may now be translatable, and if so, then we can attempt to prove that they translate to theorems. In this manner constructive theorem-proving can help to construct a specification morphism.

4 Examples

A partial taxonomy of algorithm classes that we have developed over the years is shown in Figure 4 [17]. Each algorithm theory is a refinement of *Problem-Theory*. The classification approach to algorithm design starts with the construction of an interpretation from *Problem-Theory* to the requirement specification; this interpretation makes explicit what problem is to be solved. Given the format for expressing requirements, this task is simple since it amounts to little more than parsing.

Ladder construction has the effect of making more structure explicit in the requirement specification so that appropriate problem-solving methods can be applied. For efficiency we want to find the strongest problem structure exhibited by the problem, so that the strongest problem-solving method can be applied.

Two examples are presented: Sorting and a Goods Distribution Problem (GDP). These are intended to clearly illustrate the concepts of this paper. They are not intended as complex or particularly challenging problems, nor are they presented in full detail. Sorting illustrates the use of classification in generating a new (divide-and-conquer) algorithm. The GDP example illustrates the synthesis of code to solve a problem by reducing it to an existing library code – it is necessary to set up the appropriate call to the code and to back-translate its results.

4.1 Sorting

The principle of divide-and-conquer is to solve small problem instances by some direct means, and to solve larger problem instances by decomposing them, solving the pieces, and composing the resulting solutions. Part of a specification for a simple divide-and-conquer design theory is given next. It provides the structure for a binary decomposition operator and corresponding composition operator. A general scheme for problem reduction theories (including divide-and-conquer) is given in [19].

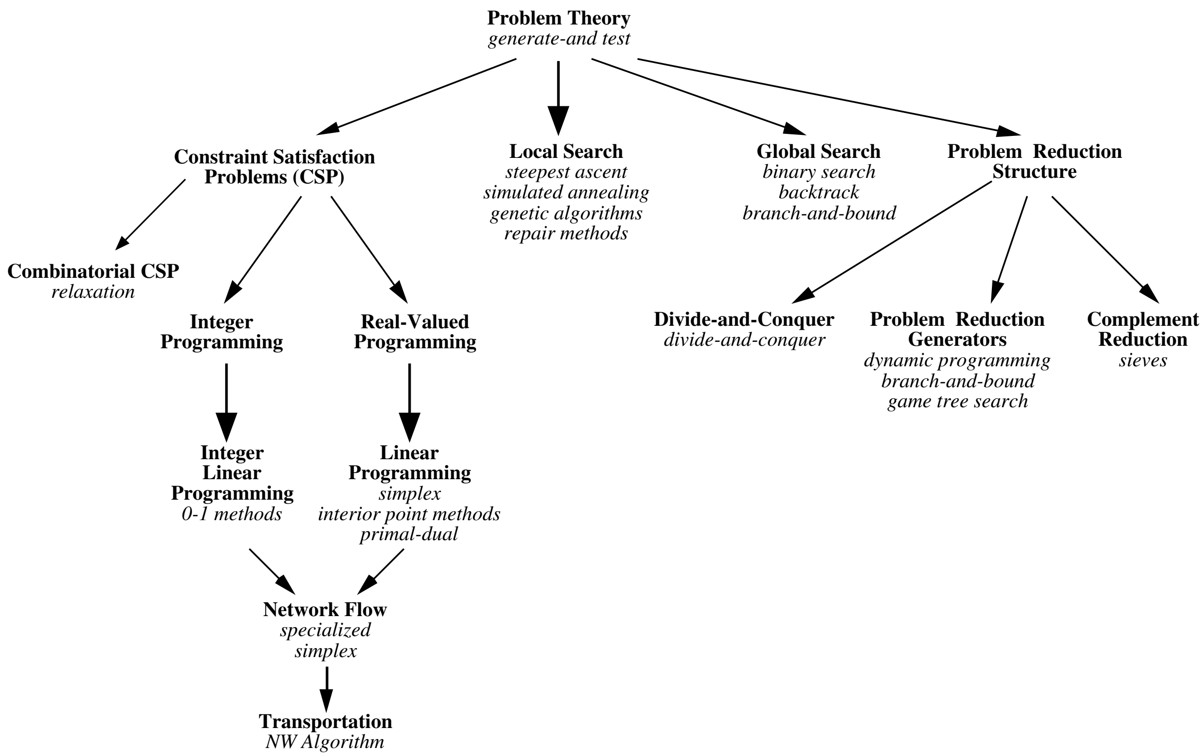

Fig. 1. Refinement Hierarchy of Algorithm Theories

Spec *Divide-and-Conquer*
Sorts

D	*input domain*
R	*output domain*

Operations

$I : D \rightarrow boolean$	*input condition*
$O : D \times R \rightarrow boolean$	*output condition*
$O_{Decompose} : D \times D \times D \rightarrow boolean$	*decomposition relation*
$O_{Compose} : R \times R \times R \rightarrow boolean$	*composition relation*
$primitive : D \rightarrow boolean$	*primitive predicate*

Axioms

$$\begin{array}{l} \forall(x_0, x_1, x_2 : D)\ \forall(z_0, z_1, z_2 : R) \\ \quad (I(x_0)\ \wedge\ O_{Decompose}(x_0, x_1, x_2) \\ \qquad \wedge\ O(x_1, z_1)\ \wedge\ O(x_2, z_2) \\ \qquad \wedge\ O_{Compose}(z_0, z_1, z_2) \\ \qquad \Longrightarrow\ O(x_0, z_0)) \end{array}$$

Soundness axiom

...

end spec

The Soundness axiom relates O, $O_{Decompose}$, and $O_{Compose}$. It asserts that if (1) nonprimitive problem instance x_0 can decompose into two subproblem instances x_1 and x_2, (2) subproblem instances x_1 and x_2 have feasible solutions z_1 and z_2 respectively, (3) z_1 and z_2 can compose to form z_0, then z_0 is a feasible solution to input x_0. We omit the remaining axioms.

An artifact theory for divide-and-conquer is parameterized on divide-and-conquer (design) theory and contains a schematic definition for the top-level divide-and-conquer functions and schematic requirement specifications for subalgorithms *Directly-Solve*, *Decompose*, and *Compose*. For simplicity we omit well-foundedness constraints that ensure termination. With well-foundedness constraints in place, this theory can be shown to be consistent. That is, given any interpretation from *Divide-and-Conquer* and functions that satisfy the requirement specifications for the subalgorithms, the corresponding instance of the divide-and-conquer function satisfies its requirement specification (see [16]).

Spec *Divide-and-Conquer-Program (T :: Divide-and-Conquer)*

op *Directly-Solve* $(x : D \mid I(x)\ \wedge\ prim(x))$
returns $(z : R \mid O(x, z))$

op *Decompose* $(x_0 : D \mid I(x_0)\ \wedge\ \neg prim(x_0))$
returns $(\langle x_1, x_2\rangle : D \times D \mid O_{Decompose}(x_0, x_1, x_2)\ \wedge\ I(x_1)\ \wedge\ I(x_2))$

op *Compose* $(z_1 : R,\ z_2 : R \mid I_{Compose}(z_1, z_2))$
returns $(z_0 : R \mid O_{Compose}(z_0, z_1, z_2))$

$$
\begin{array}{l}
\textbf{def } F(x_0 : D \mid I(x_0)) \\
\quad \textbf{returns } (z : R \mid O(x, z)) \\
\quad = \textbf{if } prim(x_0) \\
\qquad \textbf{then } \mathit{Directly\text{-}Solve}(x_0) \\
\qquad \textbf{else } let(\langle x_1, x_2 \rangle : D \times D \; = \; Decompose(x_0)) \\
\qquad\qquad Compose(F(x_1),\; F(x_2))
\end{array}
$$

end spec

The development by classification of a divide-and-conquer algorithm for sorting begins with the construction of an interpretation from *Problem-theory* to *Sorting* theory (as described earlier):

$$
\begin{array}{rcl}
D & \longmapsto & bag(S) \\
I & \longmapsto & \lambda(x)\; true \\
R & \longmapsto & seq(S) \\
O & \longmapsto & \lambda(x, z)\; ordered(z) \;\wedge\; x = bagify(z)
\end{array}
$$

Since the morphism from *Problem-theory* to *Divide-and-Conquer* is an inclusion, we can use straightforward propagation to obtain translations for the components of *Problem-theory* in *Divide-and-Conquer*:

$$
\begin{array}{rcl}
D & \longmapsto & bag(S) \\
I & \longmapsto & \lambda(x)\; true \\
R & \longmapsto & seq(S) \\
O & \longmapsto & \lambda(x, z)\; ordered(z) \;\wedge\; x = bagify(z) \\
Primitive & \longmapsto & ? \\
O_{Decompose} & \longmapsto & ? \\
O_{Compose} & \longmapsto & ?
\end{array}
$$

To complete the classification arrow we attempt to translate the remaining operators into expressions of *Sorting*. Alternative translations give rise to different sorting algorithms.

There are several ways to proceed. One tactic in KIDS is based on the choice of a standard decomposition operator from a library. The tactic then uses unskolemization on the soundness axiom to derive a specification for a composition operator. This approach allows the derivation of insertion sort, mergesort, and various parallel sorting algorithms [16, 21]. A dual approach is to choose a simple composition relation and use the Soundness axiom to derive a decomposition operator.

Suppose that we choose concatenation as a simple composition relation on the output domain $seq(integer)$. This gives us the partial signature morphism

$$
\begin{array}{rcl}
D & \longmapsto & bag(S) \\
I & \longmapsto & \lambda(x)\ true \\
R & \longmapsto & seq(S) \\
O & \longmapsto & \lambda(x,z)\ ordered(z)\ \wedge\ x = bagify(z) \\
Primitive & \longmapsto & ? \\
O_{Decompose} & \longmapsto & ? \\
O_{Compose} & \longmapsto & \lambda(z_0,z_1,z_2)\ z_0 = concat(z_1,z_2)
\end{array}
$$

The soundness axiom

$$
\begin{array}{l}
\forall(x_0,x_1,x_2 : D)\ \forall(z_0,z_1,z_2 : R) \\
\quad (I(x_0)\ \wedge\ O_{Decompose}(x_0,x_1,x_2) \\
\quad \wedge\ O(x_1,z_1)\ \wedge\ O(x_2,z_2) \\
\quad \wedge\ O_{Compose}(z_0,z_1,z_2) \\
\quad\ \Longrightarrow\ O(x_0,z_0))
\end{array}
$$

cannot yet be translated into *Sorting* because of $O_{Decompose}$. However, unskolemization on operator symbol $O_{Decompose}$ replaces the occurrence of $O_{Decompose}$ by a variable:

$$
\begin{array}{l}
\forall(x_0,x_1,x_2 : D)\ \exists(y : boolean)\ \forall(z_0,z_1,z_2 : R) \\
\quad (I(x_0)\ \wedge\ y\ \wedge\ O(x_1,z_1)\ \wedge\ O(x_2,z_2)\ \wedge\ O_{Compose}(z_0,z_1,z_2) \\
\quad\ \Longrightarrow\ O(x_0,z_0)).
\end{array}
$$

This formula can be translated via the partial signature morphism yielding:

$$
\begin{array}{l}
\forall(x_0,x_1,x_2 : bag(integer))\ \exists(y : boolean)\ \forall(z_0,z_1,z_2 : seq(integer)) \\
\quad (true\ \wedge\ y \\
\quad\quad \wedge\ ordered(z_1)\ \wedge\ x_1 = bagify(z_1) \\
\quad\quad \wedge\ ordered(z_2)\ \wedge\ x_2 = bagify(z_2) \\
\quad\quad \wedge\ z_0 = concat(z_1,z_2) \\
\quad\ \Longrightarrow\ ordered(z_0)\ \wedge\ x_0 = bagify(z_0))
\end{array}
$$

A straightforward proof of this formula in *Sorting* results in a constructive definition of $O_{Decompose}$ (see [16]):

$$x_0 = x_1 \uplus x_2\ \wedge\ x_1 \le x_2$$

where $\uplus$ is bag-union and $x_1 \le x_2$ means that each element of bag x_1 is less-than-or-equal to each element of bag x_2. This is, of course, a specification for a partition operation in a Quicksort. If we take this as the translation of $O_{Decompose}$, then we know that the soundness axiom translates to a theorem in *Sorting-theory* by construction.

The remaining steps in constructing this classification arrow include unskolemizing another axiom to obtain a translation for the *prim* predicate, and translating and proving other axioms. The resulting algorithm is a variant of Quicksort. Once the classification arrow is complete, *divide-and-conquer-program-theory* can be instantiated to obtain concrete code; see [16, 19] for details.

4.2 Distribution of Goods

$$\begin{tikzcd}
\text{Problem Theory} \arrow[r, Rightarrow, "I_0"] \arrow[d] & GDP_0 \arrow[d] \\
\text{Constraint Satisfaction} \arrow[r, Rightarrow, "I_1"] \arrow[d] & GDP_1 \arrow[d] \\
\text{Integer Programming} \arrow[r, Rightarrow, "I_2"] \arrow[d] & GDP_2 \arrow[d] \\
\text{Integer Linear Programming} \arrow[r, Rightarrow, "I_3"] \arrow[d] & GDP_3 \arrow[d] \\
\text{Network Flow} \arrow[r, Rightarrow, "I_4"] & GDP_4
\end{tikzcd}$$

GDP Ladder Construction

Suppose that a large organization needs to rationalize its distribution system and to lower its shipping costs. The distribution system comprises factories that produce goods, warehouses that store the goods, and outlets that sell the goods. There is a known cost for shipping goods from factory to warehouse and from warehouse to outlet. The problem is to find a least cost flow of goods from the factories through the warehouses to the outlets. The figure to the left shows the ladder construction that enables us to classify this problem as a network flow problem and generate code to solve it by invoking a pre-existing network flow code.

The natural first approach to modeling this problem domain is to introduce three sorts called *Factory*, *Warehouse*, and *Retail-Outlet*. A little further work on the nature of the problem constraints leads to the conclusion that these distinctions are cumbersome. The shipping routes from *Factory* to *Warehouse* are a distinct type from routes from *Warehouse* to *Retail-Outlet*. More seriously, factories and warehouses may coincide, similarly warehouses and retail outlets, and even factories and retail outlets. These observations suggest that we abstract a little and use a generalized sort called *depot*. Associated with each depot is a *supply* of goods. Depots that represent factories have a positive supply; retail outlets consume goods, so their supply is negative; and warehouses have zero supply since goods flow through them. A specification for this goods distribution problem appears in Figure 2. For simplicity we focus on the problem of finding a feasible distribution flow:

$$\begin{array}{l}
GDP\ (dpts : set(Depot),\ supply : map(Depot,\ Supply\text{-}quant) \\
\qquad\qquad \mid domain(supply) = dpts) \\
\quad \mathbf{returns}\ (flow : map(Channel,\ Flow\text{-}quant) \\
\qquad\qquad \mid domain(flow) = cart\text{-}product(dpts,\ dpts) \\
\qquad\qquad \wedge \forall(ch : Channel)\ (ch \in domain(flow) \implies 0 \leq flow(ch)) \\
\qquad\qquad \wedge\ balanced\text{-}flow(dpts,\ supply,\ flow))
\end{array}$$

This problem specification is easily expressed as an interpretation from *Problem-Theory* to *DISTRIBUTION-SYSTEM*:

Spec *GDP*

imports *arithmetic, map, set, tuple*

sorts *Depot*,
$Channel = tuple(Depot, Depot)$,
$Supply\text{-}quant = integer$,
$Flow\text{-}quant = integer$

def *FLOW-OUT*
$(d : Depot,\ dpts : set(Depot),\ flow : map(Channel,\ Flow\text{-}quant)$
$\mid d \in dpts\ \wedge\ domain(flow) = cart\text{-}product(dpts,\ dpts)) : Flow\text{-}quant$
$= reduce(+,\ image(\lambda(e : Depot)\ flow(\langle d,\ e\rangle),\ dpts))$

def *FLOW-IN*
$(d : Depot,\ dpts : set(Depot),\ flow : map(Channel,\ Flow\text{-}quant)$
$\mid d \in dpts\ \wedge\ domain(flow) = cart\text{-}product(dpts,\ dpts)) : Flow\text{-}quant$
$= reduce(+,\ image(\lambda(e : Depot)\ flow(\langle e,\ d\rangle),\ dpts))$

def *BALANCED-FLOW*
$(dpts : set(Depot),$
$supply : map(Depot,\ Supply\text{-}quant),$
$flow : map(Channel,\ Flow\text{-}quant)$
$\mid domain(supply) = dpts$
$\wedge\ domain(flow) = cart\text{-}product(dpts,\ dpts)) : boolean$
$= \forall(d : Depot)$
$(d \in dpts$
$\implies flow\text{-}out(d,\ dpts,\ flow) - flow\text{-}in(d,\ dpts,\ flow) = supply(d))$

op $GDP(dpts : set(Depot),\ supply : map(Depot,\ Supply\text{-}quant)$
$\mid domain(supply) = dpts)$
returns $(flow : map(Channel,\ Flow\text{-}quant)$
$\mid domain(flow) = cart\text{-}product(dpts,\ dpts)$
$\wedge\ \forall(ch : Channel)\ (ch \in domain(flow) \implies 0 \le flow(ch))$
$\wedge\ balanced\text{-}flow(dpts,\ supply,\ flow))$

end-spec

Fig. 2. GDP Specification

$$
\begin{array}{lcl}
D & \longmapsto & set(Depot) \times map(Depot, Supply\text{-}quant) \\
I & \longmapsto & \lambda(dpts, supply)\ domain(supply) = dpts \\
R & \longmapsto & map(Channel, Flow\text{-}quant) \\
O & \longmapsto & \lambda(dpts, supply, flow) \\
 & & \quad domain(flow) = cart\text{-}product(dpts, dpts) \\
 & & \quad \wedge\ \forall(ch : Channel)\ (ch \in domain(flow) \implies 0 \le flow(ch)) \\
 & & \quad \wedge\ balanced\text{-}flow(dpts, supply, flow)
\end{array}
$$

4.3 Constraint Satisfaction Problems

The goal now is to classify the structure of the goods-distribution problem (GDP) so that we can design a good algorithm for it. We try to classify the goods-distribution problem towards the classes of Operations Research algorithms in Figure 1.

The first step is to see if GDP can be classified as a Constraint Satisfaction Problem (CSP). Intuitively, the goal of a CSP is to produce an assignment of values to some finite set of variables subject to contraints on the assignments. A specification of CSP is:

Spec *Constraint-Satisfaction-Problem*
imports *Boolean, map, set*
sorts D, *Var*, *Val*
op $I : D \longrightarrow boolean$
op $Variables : D \longrightarrow set(Var)$
op $Legal\text{-}Val : D,\ Var,\ map(Var,\ Val) \longrightarrow boolean$
op $O\text{-}Constraint : D,\ map(Var,\ Val) \longrightarrow Boolean$
op $O\text{-}CSP : D,\ map(Var,\ Val) \longrightarrow Boolean$
def $O\text{-}CSP(x,\ vm) = (domain(vm) = Variables(x)$
$\quad\quad \wedge\ \forall(v : Var)(v \in domain(vm)$
$\quad\quad\quad\quad \Longrightarrow\ Legal\text{-}Val(x,\ v,\ vm))$
$\quad\quad \wedge\ O\text{-}constraint(x,\ vm))$
end-spec

The intention is that sort D provides input data constrained by input condition I. Var and Val are the sorts of variables and values respectively. *Variables* computes the set of variables for a given input and *Legal-Val* constrains the values that can be assigned to a particular variable. The output sort is *map(Var, Val)* and is subject to the further condition *O-constraint*. The view of CSPs as problems is carried by the refinement morphism from *Problem-Theory* to *Constraint-Satisfaction-Problem*:

$$\begin{array}{ccl} D & \longmapsto & D \\ I & \longmapsto & I \\ R & \longmapsto & map(Var, Val) \\ O & \longmapsto & O\text{-}CSP \end{array}$$

To classify GDP as a CSP we perform propagation of the symbol translations from *Problem-Theory* $\stackrel{I_0}{\Longrightarrow}$ *GDP*, yielding the main translations in Figure 3. Identification of the *map* subspecification in *CSP* with the corresponding map in *GDP* triggers propagation rules that infer

$$\begin{array}{l} Var \mapsto Depot \times Depot \\ Val \mapsto Flow\text{-}Quant \end{array}$$

i.e. that the variables are pairs of depots and the values being assigned to the variables are flow quantities. These translations are shown in small font inside

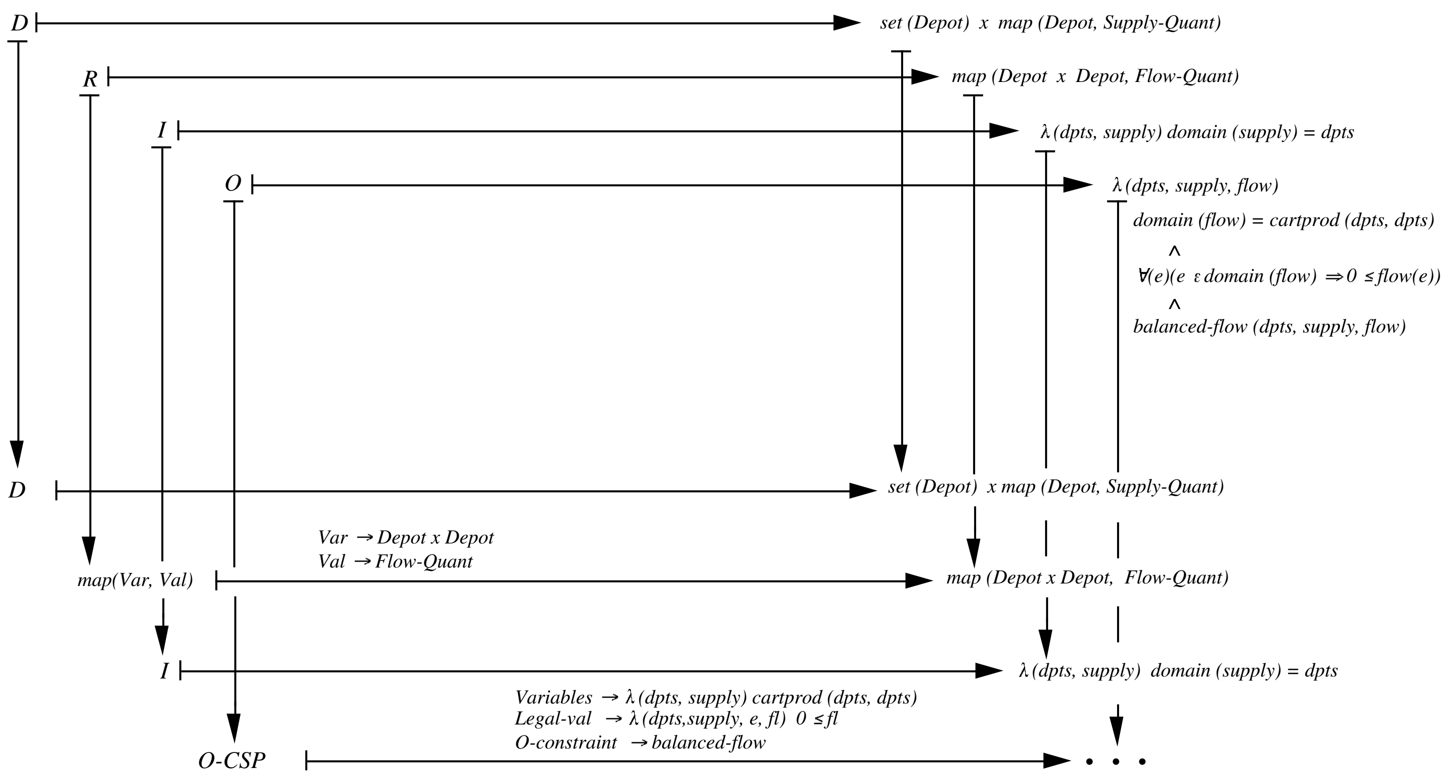

Fig. 3. Classifying GDP as a CSP

the main translations in Figure 3. Unskolemization of the definition of *O-CSP* on the operator symbols *Variables*, *Legal-val*, and *O-constraint* easily yields the remaining translations via unification with the output condition in *GDP* (See figure).

4.4 Integer Programming Problems

Integer Programming Problems (IPPs) are one possible refinement of CSPs, in that they further constrain *Vals* to be *integers*, and restrict the constraints to be a set of equations of a certain form. A specification of IPP follows:

Spec *Integer-Programming-Problem*
imports *Boolean, integer, map, set*
sorts *D, Constraint, Var*
op $I : D \rightarrow boolean$
op $Variables : D \rightarrow set(Var)$
op $Legal\text{-}Val : D,\ Var,\ integer \rightarrow boolean$
op $Constraints : D \rightarrow set(Constraint)$
op $H : D,\ Constraint,\ map(Var,\ integer) \rightarrow integer$
op $b : D,\ Constraint \rightarrow integer$
op $O\text{-}IPP\text{-}constraint : Dmap(Var,\ integer) \rightarrow boolean$
def $O\text{-}IPP\text{-}constraint(x,\ vm)$
$$= \forall(c : Constraint)(c \in Constraints(x) \implies H(x,\ c,\ vm) = b(x,\ c))$$
end-spec

A new sort called *Constraint* is introduced and is used to index two new functions H and b which serve to define a system of equational constraints that are encapsulated in the definition of *O-IPP-constraint*. Again, note that *Integer-Programming-Problem* only provides the basic sorts and operations needed to state IPPs. No more is needed at this point. The view of IPPs as problems is carried by the refinement morphism from *Constraint-Satisfaction-Problem* to *Integer-Programming-Problem*:

$$\begin{array}{rcl}
D & \longmapsto & D \\
I & \longmapsto & I \\
Var & \longmapsto & Var \\
Val & \longmapsto & integer \\
Legal\text{-}Val & \longmapsto & Legal\text{-}Val \\
Variables & \longmapsto & Variables \\
O\text{-}constraint & \longmapsto & O\text{-}IPP\text{-}constraint
\end{array}$$

To classify GDP as an IPP we perform propagation of the symbol translations from *Constraint-Satisfaction-Problem* $\stackrel{I_1}{\Longrightarrow}$ *GDP*, yielding the main translations in Figure 4 (which only shows the nonobvious translations). Unskolemization of the definition of *O-IPP-constraint* yields the remaining translations via unification with the balanced flow constraint in *GDP* (See figure). The classification arrow shows that the constraints correspond to depots and that the constraints check whether the flows in and out match the supply for each depot.

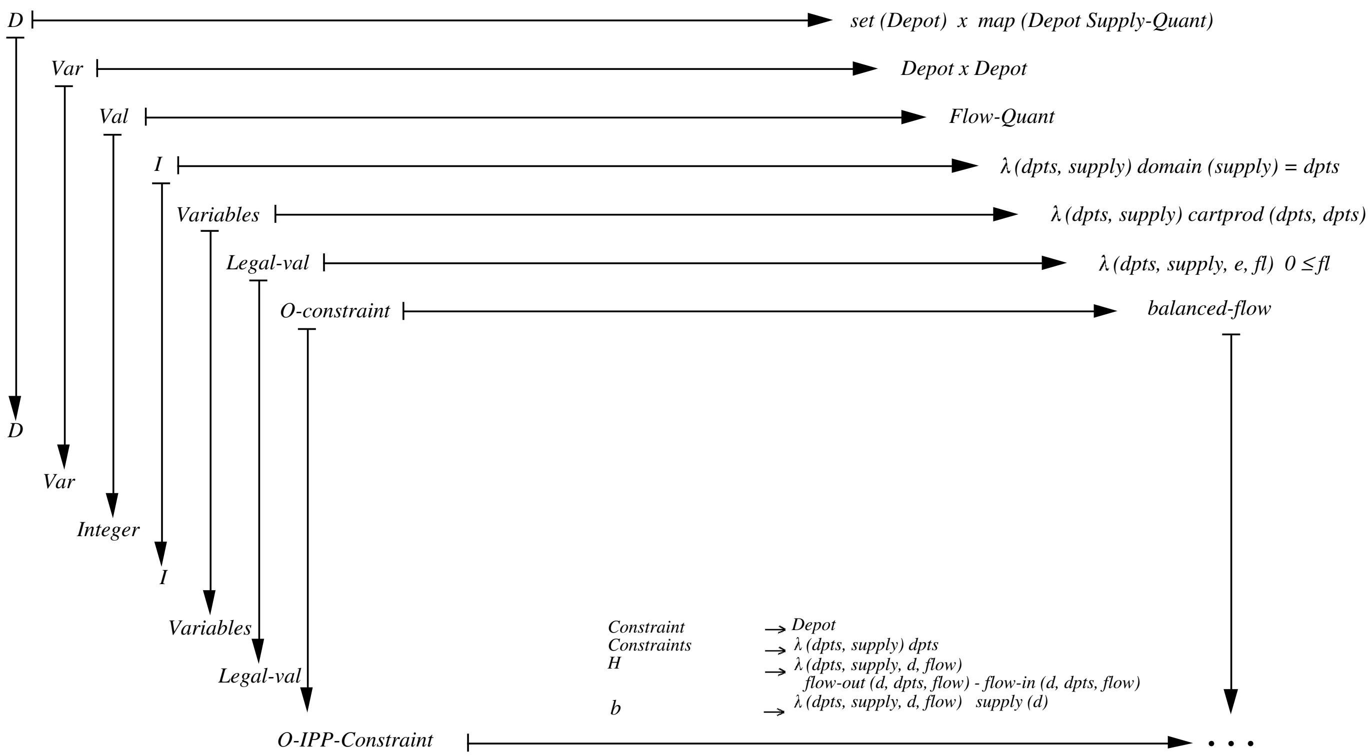

Fig. 4. Classifying GDP as an IPP

Spec *MODFLO-Solver* (*NF* :: *Network-Flow*)

def *NF-solve*($x : D \mid I(x)$)
returns (*flow* : *map*(*ARC*, *integer*) | *domain*(*flow*) = *Arcs*(x) $\wedge$ ...)
= ...code to compute a feasible network flow
expressed over the symbols of *Network-Flow*
and programming language L ...
end-spec

Fig. 5. Artifact Theory for Network Flow

Spec *MODFLO-Solver-for-GDP*
imports *DISTRIBUTION-SYSTEM*

def *GDP*(*dpts* : *set*(*Depot*), *supply* : *map*(*Depot*, *Supply-quant*)
| *domain*(*supply*) = *dpts*)
returns (*flow* : *map*(*Channel*, *integer*)
| *domain*(*flow*) = *cart-product*(*dpts*, *dpts*) $\wedge$ ...)
= ...code to compute a feasible network flow
expressed over the symbols of *DISTRIBUTION-SYSTEM*
and programming language L ...
end-spec

Fig. 6. Goods Distribution Program

4.5 Network Flow Problems

We skip two rungs down the ladder at this point. Network Flow Problems (NFPs) are a refinement of Integer Linear Programming (ILP) which in turn is a refinement of Integer Programming Problem (IPP). The input to a NFP is a directed graph with upper and lower bounds on arc capacity and a supply value at each node (positive for sources, negative for sinks). The goal is to assign a flow to each arc such that the flow is balanced and each flow is within the capacity bounds on the arc. The optimization version assigns a cost to each unit of flow through each arc and asks for a minimal cost flow over all feasible flows. A specification of the feasibility form of NFP is:

Spec *Network-Flow*
imports *Boolean*, *integer*, *tuple*, *set*
sorts D, *Node*, *Arc* = *tuple*(*Node*, *Node*)
op $I : D \rightarrow boolean$
op $Nodes : D \rightarrow set(Node)$

op $Arcs : D \rightarrow set(Arc)$
op $Arc\text{-}lb\text{-}capacity : D,\ Node,\ Node,\ integer \rightarrow boolean$
op $Arc\text{-}ub\text{-}capacity : D,\ Node,\ Node,\ integer \rightarrow boolean$
op $Supply : D,\ Node \rightarrow integer$
end-spec

Suppose that we have continued the ladder construction and finally managed to classify GDP as a Network Flow problem:

$$\begin{array}{rcl}
D & \longmapsto & set(Depot) \times map(Depot, Supply\text{-}quant) \\
I & \longmapsto & \lambda(dpts, supply)\ domain(supply) = dpts \\
Node & \longmapsto & Depot \\
Nodes & \longmapsto & \lambda(dpts, supply)\ dpts \\
Arc & \longmapsto & Channel \\
Arcs & \longmapsto & \lambda(dpts, supply)\ cart\text{-}product(dpts, dpts) \\
Arc\text{-}lb\text{-}capacity & \longmapsto & \lambda(dpts, supply, d1, d2, i)\ 0 \leq i \\
Arc\text{-}ub\text{-}capacity & \longmapsto & \lambda(dpts, supply, d1, d2, i)\ true \\
Supply & \longmapsto & \lambda(dpts, supply, d : Depot)\ supply(d)
\end{array}$$

Once we have a problem classified as a *NFP*, then the artifact theory (called a program theory) shown in Figure 5 can be used to obtain concrete code. The classification arrow from *NFP* to *GDP* binds the parameter in the program theory and a pushout calculation carries out the instantiation. The effect is to extend the GDP specification with the translated program scheme as shown in Figure 6.

The actual program theory in our system sets up a foreign function call to a FORTRAN network flow solver called MODFLO. In effect MODFLO has been wrapped in a well-defined interface, providing an example of how "legacy" code can be made to work in a formal software development process. An alternative approach to solving *GDP* using a network flow algorithm would be via problem reduction [16] which is a special case of a connection between specifications [20]. The necessary inferences would be difficult in general, whereas here we have achieved a similar result with a sequence of relatively easy propagation and unskolemization steps.

5 Concluding Remarks

The examples in this paper have shown how a refinement hierarchy of algorithm theories can be used to support algorithm design. We believe that the classification approach can usefully support a much broader range of design tasks, such as the design of data structures, user interfaces, and software systems. data structure design and software system design. For example, we conjecture that a hierarchy of software architecture theories could be used to support software system design. An architecture theory is parameterized on the interfaces to its component modules and its body specifies exported services, interconnections between components, system invariants, etc. More generally, we conjecture that classification is applicable to the design of any kind of artifact whose requirements can be specified and for which standard design abstractions can be expressed in a refinement hierarchy.

The ladder construction technique is being implemented in Specware [23], a specification-based development system at Kestrel Institute. We believe that this system will be much more flexible and easy to use than KIDS because a few general mechanisms support design directly from a taxonomic library of design theories without relying on complex, hard-to-write design tactics.

Acknowledgements

Discussions with Richard Jüllig, Junbo Liu, and Y.V. Srinivas helped to clarify the ideas presented above. This research was supported by the Air Force Office of Scientific Research under Contract F49620-91-C-0073, by the Office of Naval Research under Grant N00014-93-C-0056, and by ARPA and Rome Laboratories under Contracts F30602-91-C-0043 and F30602-95-10018.

References

1. Astesiano, E., and Wirsing, M. An introduction to ASL. In *Program Specification and Transformation*, L. Meertens, Ed. North-Holland, Amsterdam, 1987, pp. 343–365.
2. Blaine, L., and Goldberg, A. DTRE – a semi-automatic transformation system. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 165–204.
3. Broy, M., Wirsing, M., and Pepper, P. On the algebraic definition of programming languages. *ACM Transactions on Programming Languages and Systems 9*, 1 (January 1987), 54–99.
4. Ehrig, H., Kreowski, H. J., Thatcher, J., Wagner, E., and Wright, J. Parameter passing in algebraic specification languages. In *Proceedings, Workshop on Program Specification* (Aarhus, Denmark, Aug. 1981), vol. 134, pp. 322–369.
5. Ehrig, H., and Mahr, B. *Fundamentals of Algebraic Specification, vol. 2: Module Specifications and Constraints.* Springer-Verlag, Berlin, 1990.
6. Enderton, H. B. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.
7. Goguen, J. A., Thatcher, J. W., and Wagner, E. G. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology, Vol. 4: Data Structuring*, R. Yeh, Ed. Prentice-Hall, Englewood Cliffs, NJ, 1978.
8. Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B. Initial algebra semantics and continuous algebras. *Journal of the ACM 24*, 1 (January 1977), 68–95.
9. Goguen, J. A., and Winkler, T. Introducing OBJ3. Tech. Rep. SRI-CSL-88-09, SRI International, Menlo Park, California, 1988.
10. Guttag, J. V., and Horning, J. J. The algebraic specification of abstract data types. *Acta Inf. 10* (1978), 27–52.
11. Hoare, C. Varieties of programming languages. Tech. rep., Programming Research Group, University of Oxford, Oxford, UK, 1989.

12. Lowry, M. R. Algorithm synthesis through problem reformulation. In *Proceedings of the 1987 National Conference on Artificial Intelligence* (Seattle, WA, July 13–17, 1987).
13. Papadimitriou, C. H., and Steiglitz, K. *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, NJ, 1982.
14. Partsch, H. *Specification and Transformation of Programs: A Formal Approach to Software Development.* Springer-Verlag, New York, 1990.
15. Sannella, D., and Tarlecki, A. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica 25*, 3 (1988), 233–281.
16. Smith, D. R. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence 27*, 1 (September 1985), 43–96. (Reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds., Los Altos, CA, Morgan Kaufmann, 1986.).
17. Smith, D. R., and Lowry, M. R. Algorithm theories and design tactics. In *Proceedings of the International Conference on Mathematics of Program Construction, LNCS 375*, L. van de Snepscheut, Ed. Springer-Verlag, Berlin, 1989, pp. 379–398. (reprinted in *Science of Computer Programming*, 14(2-3), October 1990, pp. 305–321).
18. Smith, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (September 1990), 1024–1043.
19. Smith, D. R. Structure and design of problem reduction generators. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 91–124.
20. Smith, D. R. Constructing specification morphisms. *Journal of Symbolic Computation, Special Issue on Automatic Programming 15*, 5-6 (May-June 1993), 571–606.
21. Smith, D. R. Derivation of parallel sorting algorithms. In *Parallel Algorithm Derivation and Program Transformation*, R. Paige, J. Reif, and R. Wachter, Eds. Kluwer Academic Publishers, New York, 1993, pp. 55–69.
22. Srinivas, Y. V. Algebraic specification for domains. In *Domain Analysis: Acquisition of Reusable Information for Software Construction*, R. Prieto-Diaz and G. Arango, Eds. IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 90–124.
23. Srinivas, Y. V., and Jüllig, R. Specware:$^{\mathrm{tm}}$ formal support for composing software. In *Proceedings of the Conference on Mathematics of Program Construction*, B. Moeller, Ed. Springer-Verlag, Berlin, 1995. Lecture Notes in Computer Science, Vol. 947.
24. Turski, W. M., and Maibaum, T. E. *The Specification of Computer Programs.* Addison-Wesley, Wokingham, England, 1987.
25. Veloso, P. A. Problem solving by interpretation of theories. In *Contemporary Mathematics*, vol. 69. American Mathematical Society, Providence, Rhode Island, 1988, pp. 241–250.
26. Wirsing, M. Algebraic specification. In *Formal Models and Semantics*, J. van Leeuwen, Ed., vol. B of *Handbook of Theoretical Computer Science.* MIT Press/Elsevier, 1990, pp. 675–788.

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