FROM REQUIREMENTS SPECIFICATION TO DESIGN SPECIFICATION

Valerie NOVITZKÁ, Ján KOLLÁR

Technical University Košice, Faculty of Electrical Engineering and Informatics,
Dept.of Computers and Informatics
e-mail: Valerie.Novitzka@tuke.sk, Jan.Kollar@tuke.sk

Abstract

The aim of program development process is to construct a correct program with respect to its requirements specification. Our paper deals with the first step of this process in the framework of the theory of correct programming. We construct requirements specification institution in terms of algebraic specifications, design specification institution in terms of functional programming and type theory, and the institution arrow from the first to the second one.

Keywords: requirements specification, design specification, institution

1 INTRODUCTION

The aim of a programmer’s work is to get a correct result that solves his problem by computer. The first task is to formulate a question, for which we would like to obtain a correct answer by using a computer. Because the logic of questions is not elaborated, it is necessary to formulate preconditions that have to be satisfied for obtaining correct answer. These preconditions together with used symbols are written in some specification language as requirements specification. Requirements specifications have to be formulated in the frame of some mathematical theory; more precisely in an axiomatic set theory or in some its conservative extensions. For dealing with this framework we use the notion of institution firstly formulated by Goguen and Burstall in [1]. Every requirements specification corresponds with a hierarchy of algebraic specifications, which components are from the requirements specification institution.

It is known that from a requirements specification more than one way leads to the result correct answer. For solving a problem, we can construct more programs that correctly answer our question, i.e. they are semantically equivalent. When we have formulated a requirements specification, we can proceed our work using our creativity and deduction in models by constructing a new specification, design specification that
contains more (implementation) details about solved problem. That means we have to construct also new signatures, and new institution that we call design specification institution. We map the requirements specification institution to design specification institution componentwise by mathematically defined morphisms, and the construction such morphism between institutions we call institution arrow. The program development process then proceeds by generating new institutions, for: specification of program text, specification of machine instructions sequence, and result specification after executing, which should provide the correct answer. The directed graph which nodes are institutions and edges are institution arrows we call development graph of a program. It is clear that a development graph has to end in one result institution. Certainly, in the construction of the result institution the correct loaders, linkers and operating systems play very important role. Institution arrows as morphisms enable us to prove the correctness of any development step in the program development process.

In this paper we describe the basic concepts in constructing the requirements specification institution, the design specification institution, institution arrow from the first to the second one, i.e. the first step in the program development process, and we illustrate this step by a simple example.

2 CONSTRUCTION OF REQUIREMENTS SPECIFICATION INSTITUTION

The most important role of a requirements specification is to establish as clearly as possible what has to be done. What has to be done is the rational solution of a (simple or very difficult) problem. Firstly, we formulate it as a question in a (natural) language. Because the truthness of a solution has to be proved by using a logic, and because in logic there are untractable interrogate sentences formulated in natural languages as questions, we has to

- find preconditions of the question as true sentences, called requirements of the problem;
- formulate them in some formal language with mathematically defined semantics called specification language.

A requirements specification is a text written in some specification language. It contains used symbols and preconditions that must be satisfied to get a correct answer to our question. In this paper we use simple algebraic specification language defined in [2] that enables to write requirements specifications as hierarchies of basic and/or structured specifications.

We illustrate our approach on a simple example of the structural parametrized requirements specification LIST_SELECTOR:
spec LIST[sort Elem] =
  sort List;
  op
    cons: Elem * List → List;
    concat: List * List → List, assoc, unit nil;
    head: List →? Elem;
    tail: List →? List;
  var e: Elem; l,l': List;
  axiom
    concat(cons(e,l), l') = cons(e, concat(l,l'));
    head(cons(e, l)) = e;
    tail(cons(e, l)) = l;
    not def head(nil);
    not def tail(nil);
end;

spec LIST_SELECTOR[sort Elem] =
  LIST[sort Elem] then
  op    select: List → List;
  pred  in: (Elem, List);
        prop: (Elem);
  var   x,y: Elem; l: List;
  axiom
    not in(x, nil);
    in(x, cons(y,l)) ⇔ ( (x=y) or in(x,l));
    in(x, select(l)) ⇔ select(l) if (in(x,l) and prop(x));
end.

The specification LIST is parametrized (generic) specification with the parameter of the sort ‘Elem’. It contains one sort ‘List’, the construction operation symbol ‘cons’, which is associative and it has a unit ‘nil’, and obvious operation symbols for concatenation, head and tail of list. As it follows from operation symbols profiles, the operations ‘cons’ and ‘concat’ are total binary operations, while ‘head’ and ‘tail’ are partial unary operations. Axioms are described as closed formulae of the first-order predicate logic. The first three axioms are (implicitly) universally quantified equations, the last two describe that the head and tail of empty lists are not defined.

The specification LIST_SELECTOR is built from LIST by extension specification building operation, i.e. we add new operation symbol ‘select’ and two
predicate symbols: ‘in’ for membership operation and ‘prop’ for defining some property of elements of list. Naturally, we have to describe the properties of new operation and predicate symbols by additional axioms.

Every specification describes an abstract data type and has a corresponding algebraic specification of the form

\[
\text{Spec} = (\Sigma, \Psi_\Sigma)
\]

where \(\Sigma = (S, O, P)\) is a signature consisting of a set \(S\) of sorts, a set \(O\) of (total and/or partial) function symbols of the form \(f: (s_1, s_2, \ldots, s_n) \rightarrow s\) with their profiles, and a set \(P\) of predicate symbols of the form \(p: (s_1, s_2, \ldots, s_n)\) with their arities. We can define signature morphism \(\sigma: \Sigma_1 \rightarrow \Sigma_2\) between signatures of the requirements specifications componentwise. A class of all requirements specification signatures together with the signature morphisms between them form the category \(\text{Sign}\) of requirements specification signatures.

The second part of \(\text{Spec}\) is the set \(\Psi_\Sigma\) of \(\Sigma\)-axioms, i.e. closed \(\Sigma\)-formulae (\(\Sigma\)-sentences) written in the language of the first-order predicate logic. \(\Sigma\)-axioms are axioms of used mathematical theories, i.e. proved theorems of an axiomatic set theory [3]. They describe the preconditions of the requirements specification, i.e. some properties of functions corresponding with function symbols from the signature \(\Sigma\).

To every signature \(\Sigma\) we can construct \(\Sigma\)-algebras by assigning data sets, functions and true \(\Sigma\)-sentences to sorts, function symbols and predicate symbols, respectively, as it is defined in detail in [2]. If for a \(\Sigma\)-sentence \(\phi\) there exists a proof in a \(\Sigma\)-algebra \(M\), then we say, \(\phi\) is satisfied in \(M\), written

\[
M \models_\Sigma \phi
\]

and such \(\Sigma\)-algebra \(M\) with satisfied \(\Sigma\)-sentences we call \(\Sigma\)-model. To every signature \(\Sigma\) from \(\text{Sign}\) we construct the category \(\text{Mod}(\Sigma)\) consisting of \(\Sigma\)-models as category objects and reduce homomorphisms between them as category morphisms.

We need still two functors

\[
\text{Sen}: \text{Sign} \rightarrow \text{Set} \quad \text{and} \quad \text{Mod}: \text{Sign}^{\text{op}} \rightarrow \text{Cat}
\]

The functor \(\text{Sen}\) assigns to every signature \(\Sigma\) from \(\text{Sign}\) a set of true \(\Sigma\)-sentences from the category \(\text{Set}\) of sets. The functor \(\text{Mod}\) from dual category of signatures \(\text{Sign}^{\text{op}}\) to the category \(\text{Cat}\) of small categories assigns to every signature \(\Sigma\) a category \(\text{Mod}(\Sigma)\) of \(\Sigma\)-models. The quadruple

\[
\mathbf{I} = (\text{Sign}, \text{Sen}, \text{Mod}, (\models_\Sigma)_{\Sigma \in \text{Sign}})
\]
we call \textit{requirements specification institution}. It is clear from the construction above that a $\Sigma$-sentence $\varphi$ is satisfied in a $\Sigma_1$-model $M_1$ iff its image under functor $\text{Sen}$ is a $\Sigma_2$-sentence satisfied in the reduct $\Sigma_2$-model $M_2$.

3 CONSTRUCTING DESIGN SPECIFICATION INSTITUTION

The software life cycle involves several levels of specifications, from the most abstract requirements specifications to the more concrete ones: design specifications, program text specifications, executable program specifications and result specifications. This process which starts from a requirements specification and goes incrementally more and more into details of implementation is often called \textit{stepwise refinement} of specifications.

As well as a requirements specification, also a design specification has to be formulated in the framework of some conservative extension of the mathematical theory for requirements specification. Therefore, we have to construct a new institution $\mathbf{I}'$ for design specification. We think that the most available mode how to construct it is to define its components by the notions of some pure functional language such as e.g. Haskell [4] because the sets corresponding with sorts can be quite simply refined to types and operations on them to functions.

3.1 Design specification signatures

The first component of the institution $\mathbf{I}'$ is the category of design specification signatures. Every design specification signature is of the form

$$\Sigma' = (\text{TO}, \varepsilon, \subseteq, v^+, v^-, C).$$

In the following text we describe the components of it in detail.

\text{TO} is a set of type operators. Type operators can be type constructors or type synonyms (aliases). Type values in functional languages are built from type constructors. Type expressions are classified into different kinds. All nullary type constructors has a kind denoted by the symbol \texttt{"*"}. A sort is a nullary type constructor. If $\kappa_1$ and $\kappa_2$ are kinds, then $\kappa_1 \rightarrow \kappa_2$ is the kind of types that take a type of the kind $\kappa_1$ and return a type of the kind $\kappa_2$. We introduce three built-in type constructors:

- \textbf{products} \hspace{1cm} _\ast : \kappa \rightarrow \kappa \rightarrow \kappa
- \textbf{partial functions} \hspace{1cm} \rightarrow?: \kappa \rightarrow \kappa \rightarrow \kappa
- \textbf{singleton type} \hspace{1cm} \text{unit} : \kappa
where $\kappa$ is the kind of all types. Functional languages enable type synonyms, or aliases, i.e. names for commonly used types. Type synonyms do not define new types, but simply give new names for existing types. To any type operator $F$ from $TO$ a kind of the form $\kappa \rightarrow \kappa \rightarrow ... \rightarrow \kappa$ is assigned.

An alias-free mapping $\varepsilon$ assigns to any type synonym a pseudotype. A pseudotype is a $\lambda$-term of the form

$$\lambda a_1,a_2,...a_n : \kappa \bullet t$$

where $a_1,a_2,...a_n$ are type variables and the type $t$ contains at most $a_1,a_2,...a_n$ as type variables. Because recursive expansion for type synonyms are forbidden, every type $t$ has a unique alias-free expansion denoted by $\varepsilon(t)$. Types associated to the signature $\Sigma'$ can be type variables or type operators of the form $F(t_1,t_2,...t_n)$.

We enable type polymorphism with subtyping constraints [5], thus type schemes are added to the types. A type scheme is of the form

$$\forall a_1,a_2,...a_n : \kappa \bullet s_1 \sqsubseteq t_1, ... s_m \sqsubseteq t_m \Rightarrow t$$

The subtyping constraint $s_i \sqsubseteq t_i$, for $i=1,2,...m$ indicates that the interpretation of the type variables is restricted to such types for which this subtype relation is valid. It is trivial to prove that the subtyping relation $\sqsubseteq$ is a preorder, i.e. it is reflexive and transitive.

To every type constructor $F$ we add variance predicates $v^+$ and $v^-$ [6,7]. They indicate which arguments of a type constructor $F$ can be used in the subtype mechanism as covariant ($v^+$) or contravariant ($v^-$). Because also functions can be passed as arguments to other functions, we must give the following subtyping rule for (function) types, i.e. we must specify under which circumstances it is safe to use a function of one type in a context where a different type is expected:

$$t_1 \sqsubseteq s_1, s_2 \sqsubseteq t_2$$

$$----------------------------------$$

$$s_1 \rightarrow s_2 \sqsubseteq t_1 \rightarrow t_2$$

If $t_1$ is a subtype of $s_1$, $t_1 \sqsubseteq s_1$, and $s_2$ is a subtype of $t_2$, $s_2 \sqsubseteq t_2$, then the type $s_1 \rightarrow s_2$ is a subtype of the type $t_1 \rightarrow t_2$. The sense of the subtype relation is reversed (contravariant) for the argument types in the left-hand premise of the rule, while it runs in the same direction (covariant) for the result types as for the function types themselves. If we have a function $f$ of a type $s_1 \rightarrow s_2$, then we know that $f$ accepts elements of type $s_1$; clearly, $f$ will also accept elements of type $t_1$. We can also view that result belongs to any supertype $t_2$ of $s_2$. That is, any function $f$ of a type $s_1 \rightarrow s_2$
can also be viewed as having a type \( t_1 \rightarrow t_2 \). An alternative view is that it is safe to allow a function of a type \( s_1 \rightarrow s_2 \) to be used in a context where another type \( t_1 \rightarrow t_2 \) is expected as long as none of the arguments that may be passed to the function in this context will surprise it (\( t_1 \sqsubseteq s_1 \)), and none of the results that it returns will surprise the context (\( s_2 \sqsubseteq t_2 \)). For instance, variance predicates for built-in type constructors for product and partial functions are:

\[
\begin{align*}
v^+(\_ \times \_) &= \{ 1, 2 \} \\
v^-(\_ \times \_) &= \emptyset \\
v^+(\_ \rightarrow \_) &= \{ 2 \} \\
v^-(\_ \rightarrow \_) &= \{ 1 \}
\end{align*}
\]

where numbers (elements of sets) are serial numbers of arguments. Let \( s_1, s_2, \ldots, s_n, t_1, t_2, \ldots, t_n \) be (alias-free) types and \( F \) be an \( n \)-ary type constructor. Then the relation

\[ F(s_1, s_2, \ldots, s_n) \sqsubseteq F(t_1, t_2, \ldots, t_n) \]

is valid, if for every \( i = 1, \ldots, n \)

\[ i \in v^+(F) \text{ and } s_i \sqsubseteq t_i \]

or \( i \in v^-(F) \text{ and } t_i \sqsubseteq s_i \)

The set \( C \) is a set of constant symbols with assigned type schemes. A constant symbol with a name \( c \) and profile \( t \) is written by \( c : t \). \( C \) contains also a distinguished constant

\[
( : \text{unit})
\]

needed for mapping predicate symbols from the requirements specification signature \( \Sigma \).

Now we have to define morphisms between signatures of design specifications. A signature morphism

\[ \sigma' : \Sigma_1' \rightarrow \Sigma_2' \]

we define simply by morphisms between their components:

- type constructors from \( \Sigma_1' \) are mapped to type operators in \( \Sigma_2' \) saving profiles and variance predicates; type synonyms are not mapped at all;
- constant symbols from \( \Sigma_1' \) are mapped to constant symbols in \( \Sigma_2' \) respecting profiles and subsorting.

We denote by \( \text{Sign}' \) the structure consisting of
• a class of design specification signatures;
• a set of morphisms between them.

It is trivial to prove that the composition of these morphisms is associative, i.e. \( \text{Sign}' \) is a category and we call it the category of design specification signatures.

### 3.2 Design specification formulae

To construct the second component of the design specification institution, i.e. the functor 

\[ \text{Sen}': \text{Sign}' \rightarrow \text{Set} \]

that provides for every design specification signature \( \Sigma' \) a set of \( \Sigma' \)-sentences, we have to define rules for forming sentences of design specification. They slightly differ from the requirements sentences. We define rules for formulating terms that extend the possible forms of terms by e.g. \( \lambda \)-abstractions:

- a variable \( x:t \) of a type \( t \) is a term;
- if \( \alpha : s \) is a term of a type \( s \) that is a subtype of a type \( t \), then \( \alpha : t \) is also a term of the type \( t \);
- if \( \alpha_1, \alpha_2, \ldots, \alpha_n \) are terms of types \( t_1, t_2, \ldots, t_n \), respectively, then \( (\alpha_1, \alpha_2, \ldots, \alpha_n) : t_1 \times t_2 \times \ldots \times t_n \) is a term of a product type \( t_1 \times t_2 \times \ldots \times t_n \);
- if \( \alpha : s \rightarrow ?t \) is a term of a partial function type and \( \beta : s \) is a term of a type \( s \), then also an application \( \alpha(\beta) : t \) is a term of the type \( t \);
- if \( \alpha : t \) is a term and \( x_1:s_1, x_2:s_2, \ldots, x_n:s_n \) are variables of the corresponding types, the also

\[ \hat{\lambda} x_1:s_1, x_2:s_2, \ldots, x_n:s_n \bullet \alpha : s_1 \times s_2 \times \ldots \times s_n \rightarrow ?t \]

is a term.

Sentences allow that universal and existential quantifier can range also over type variables \( a:k \) and can contain subtyping constraints in the same form as for type schemes. But the quantification over type schemes is forbidden. Predicates in sentences are replaced by terms of the type \( \text{unit} \). Although there is no direct mode to use logical connectives in \( \lambda \)-terms, we can write for example the constant true predicate on a type \( a \) just as \( \hat{\lambda} x : a \bullet () \); and the conjunction as \( \hat{\lambda} x_1, x_2 : \text{unit} \bullet () \).

### 3.3 Design specification models

The third component of the design specification institution \( I' \) is the functor \( \text{Mod}' : \text{Sign}' \rightarrow \text{Cat} \) that assigns to every design signature \( \Sigma' \) a category \( \text{Mod}(\Sigma') \) of \( \Sigma' \)-models. It seems to be most appropriate to choose as design specification model \( M' \) an intensional Henkin model where function types are interpreted by arbitrary sets equipped with an application operation of the appropriate type. Such models enable
better analysis of function spaces and they directly correspond with the class of requirements specification models.

Let \( \Sigma' \) be a design specification signature. A design specification model \( M' \) is an algebra consisting of the sets \( M'_t \) for every alias-free type \( t \) generated by \( \Sigma' \) in such a way, that

- \( \text{unit} \) is interpreted as singleton set;
- product types \( t_1 \times t_2 \times \cdots \times t_n \) are interpreted as Cartesian products together with an assignment of a partial interpretation function
  \[
  M'_{t_1} \times M'_{t_2} \times \cdots \times M'_{t_n} \rightarrow \models M'\_t
  \]
  to every term of the type \( t \) with variables \( x_1:t_1, x_2:t_2, \ldots, x_n:t_n \).

These interpretation functions must respect deducible equality of terms, the usual equality rules for existential equality and for \( \beta-, \eta-, \text{and}\ \xi\)-rules with multiple arguments in the case of simultaneous \( \lambda\)-abstraction over several variables. Substitution has to be modeled as a composition of partial functions. Terms of the form

\[
\lambda x: s \cdot x: t
\]

where \( s \subseteq t \) are internally total functions.

Morphisms between \( \Sigma'\)-models are usual homomorphisms, i.e. the set of homomorphisms for every type \( t \). For any design specification signature \( \Sigma' \) the class of \( \Sigma'\)-models together with the corresponding homomorphisms between them form the category \( \text{Mod}(\Sigma') \) of design specification models.

The satisfaction relation \( \models_{\Sigma'} \) is essentially the same as for requirements specification institution. Formulae that are universally quantified over type variables with subtype constraints are satisfied iff the enclosed formula is satisfied for all valuations of the type variables that satisfy the constraints. If the result of a predicate application is defined, then it is satisfied as an atomic formula.

We say, that the quadruple

\[
I' = (\text{Sign}', \text{Sen}', \text{Mod}', (\models_{\Sigma'}))_{\Sigma' \in \text{Sign}'}
\]

is the design specification institution.

4 DEVELOPMENT STEP FROM REQUIREMENTS TO DESIGN SPECIFICATION INSTITUTION

Now we construct the first step of the program development process in the framework of institutions, i.e. the institution arrow from the requirements specification institution \( I \) to the design specification institution \( I' \). This arrow consists of the
following mappings. The first is a functor $\Phi$ that maps the requirements specification category $\text{Sign}$ to the design specification category $\text{Sign}'$

$$\Phi: \text{Sign} \rightarrow \text{Sign}'$$

such that for any requirements specification signature $\Sigma = (S, O, P)$ the functor $\Phi$ maps its components as follows:

- sorts from $S$ are mapped to the nullary type constructors in $TO$;
- constant function symbols are mapped to constants in $C$ which contains also a special constant $(())$: \text{unit};
- function symbols from $O$ are mapped to the type operators in $TO$. If the profile of a function symbol contains a subtype, we must to define variance predicates for it. To the name of a type we add the corresponding $\lambda$-term defining it;
- predicate symbols from $P$ are mapped to the type operators in $TO$ with the result type \text{unit}.

The second component of the institution arrow is a natural transformation (i.e. mapping between functors)

$$\mu^\text{Sen}: \text{Sen}' \circ \Phi \rightarrow \text{Sen}$$

which is a set of morphisms for every signature $\Sigma$ from $\text{Sign}$. $\Sigma$-axioms are mapped to appropriate (mostly) $\lambda$-terms mentioned above.

The last component of the institution arrow is a natural transformation

$$\mu^\text{Mod}: \text{Mod} \rightarrow \text{Mod}' \circ \Phi$$

assigning to requirements specification models the appropriate design specification models. Then the institution arrow is the triple

$$(\Phi, \mu^\text{Sen}, \mu^\text{Mod}): I \rightarrow I'.$$

To illustrate this development step, we present a possible design specification $\text{DESIGN\_LIST\_SELECTOR}$ corresponding to the requirements specification $\text{LIST\_SELECTOR}$ above. Clearly, this design specification is already easy rewriteable to satisfy the concrete syntax of a functional language [8].

```
\text{design spec} \text{DESIGN\_LIST\_SELECTOR} [a] =

\text{LIST}[a] \text{ then}

\text{op} \text{ select: List}(a) \rightarrow \text{List}(a) = \text{rec}
```
\[ \lambda \text{prop: } a \rightarrow \text{unit} \cdot \lambda \text{xs: } \text{List}(a) \cdot \\
\text{case } \text{xs of} \\
[ ] \rightarrow [ ] \\
(y::ys) \rightarrow \text{case } \text{prop } y \text{ of} \\
\text{true } \rightarrow y:: \text{select } \text{prop } ys; \\
\text{false } \rightarrow \text{select } \text{prop } ys \\
\text{end.} \]

5 CONCLUSION

In our paper we have shown a step in the formal development of programs based on mathematically precise mapping between different kinds of specifications that is necessary for constructing correct programs from specifications. Because it is only the first step in program development process we follow our research by extending mathematical reasoning about the whole programming process.

This paper was supported by the Grant No.1/8134/01 “Binding the Process Functional Language to MPI” of the Slovak Scientific Grant Agency VEGA.

REFERENCES


Referee: Vitaly Levashenko