Specifying Compositional Units for Correct Program Development in Computational Logic

Kung-Kiu Lau[1] and Mario Ornaghi[2]

[1] Department of Computer Science, University of Manchester
Manchester M13 9PL, United Kingdom
kung-kiu@cs.man.ac.uk

[2] Dipartimento di Scienze dell'Informazione, Universita' degli studi di Milano
Via Comelico 39/41, 20135 Milano, Italy
ornaghi@dsi.unimi.it

Abstract. In order to provide a formalism for defining program correctness and to reason about program development in Computational Logic, we believe that it is better to distinguish between specifications and programs. To this end, we have developed a general approach to specification that is based on a model-theoretic semantics. In our previous work, we have shown how to define specifications and program correctness for open logic programs. In particular we have defined a notion of correctness called *steadfastness*, that captures at once modularity, reusability and correctness. In this paper, we review our past work and we show how it can be used to define compositional units that can be correctly reused in modular or component-based software development.

1 Introduction

In software engineering, requirements analysis, design and implementation are distinctly separate phases of the development process [18], as they employ different methods and produce different artefacts. In requirements analysis and design, *specifications* play a central role, as a frame of reference capturing the requirements and the design decisions. By contrast, data and programs only appear in the implementation phase, towards the end of the development process. There is therefore a clear distinction between specifications and programs.

In Computational Logic, however, this distinction is usually not maintained. This is because there is a widely held view that logic programs are executable specifications and therefore there is no need to produce specifications before the implementation phase of the development process. We believe that undervaluing specifications in this manner is not an ideal platform for program development. If programs are indistinguishable from specifications, then how do we define program correctness, and how do we reason about program development? We hold the view that the meaning of correctness must be defined in terms of something other than logic programs themselves. We are not alone in this, see e.g., [17, p. 410]. In our view, the specification should axiomatise all our relevant knowledge of the problem context and the necessary data types, whereas,

M. Bruynooghe and K.-K. Lau (Eds.): Program Development in CL, LNCS 3049, pp. 1–29, 2004.
© Springer-Verlag Berlin Heidelberg 2004

for complexity reasons, programs rightly capture only what is strictly necessary for computing. In the process of extracting programs from specifications, a lot of knowledge is lost, making programs much weaker axiomatisations. This suggests that specifying and programming are different activities, involving different methodological aspects. Thus, we take the view that specifications should be clearly distinguished from programs, especially for the purpose of program development. Indeed, we have shown (in [28,29]) that in Computational Logic, not only can we maintain this distinction, but we can also define various kinds of specifications for different purposes. Moreover, we can also define correctness with respect to these specifications.

Our semantics for specification and correctness is model-theoretic. The declarative nature of such a semantics allows us to define *steadfastness* [34], a notion of correctness that captures at once modularity, reusability and correctness. *Open* programs are incomplete pieces of code that can be (re)used in many different *admissible* situations, by *closing* them (by adding the missing code) in many different ways. *Steadfastness* of an open program P is pre-proved correctness of the various closures of P, with respect to the different meanings that the specification of P assumes in the admissible situations. For correct reuse, we need to know when a situation is admissible. This knowledge is given by the *problem context*. We have formalised problem context as a *specification framework* [27], namely, a first-order theory that axiomatises the problem context, characterises the admissible situations as its (intended) models, and is used to write specifications and to reason about them.

In this paper, we review our work in *specification* and *correctness* of logic programs, including steadfastness. Our purpose is to discuss the role of steadfastness for correct software development. In particular, we are interested in modularity and reuse, which are key aspects of software development. Our work is centred on the notion of a *compositional unit*. A compositional unit is a *software component*, which is commonly defined as a unit of composition with contractually specified interfaces and context dependencies only [46]. The interfaces declare the imported and exported operations, and the context dependencies specify the constraints that must be satisfied in order to correctly (re)use them. Throughout the paper, we will not refer to compositional units as software components, however, for the simple reason that as yet there is no standard definition for the latter (although the one we used above [46] is widely accepted). So we prefer to avoid any unnecessary confusion. In our compositional units, the interfaces and the context dependencies are declaratively specified in the context of the specification framework $\mathcal{F}$ axiomatising the problem context. $\mathcal{F}$ gives a precise semantics to specifications and allows us to reason about the *correctness* of programs, as well as their *correct reuse*. Thus, in our formalisation, a compositional unit has a three-tier structure, with separate levels for framework, specifications and programs.

We introduce compositional units in Section 2, and consider the three levels separately. We focus on model-theoretic semantics of frameworks and specifications, and on steadfastness (i.e., open program correctness).

In Section 3, we show how the proposed formalisation of compositional units can be used to support correct reuse. Our aim is to highlight the aspects related to specifications, so we consider only the aspects related to the framework and the specification levels, while assuming the possibility of deriving (synthesising) steadfast programs from specifications.

At the end of each section we briefly discuss and compare our results with related work, and finally in the conclusion we comment on future developments.

2 Compositional Units

In our approach, compositional units represent correctly reusable units of *specifications and correct open programs*. Our view is that specifications and programs are not stand-alone entities, but are always to be considered in the light of a problem context. The latter plays a central role: it is the *semantic context* in which specifications and program correctness assume their appropriate meaning, and it contains the necessary knowledge for *reasoning* about correctness and correct reuse. This is reflected in the three-tier structure (with model-theoretic semantics) of a compositional unit, as illustrated in Figure 1.

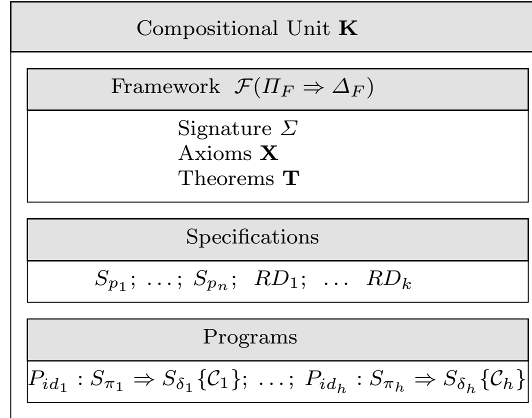

Fig. 1. A three-tier formalism.

At the top level of a compositional unit **K**, we have a *specification framework* $\mathcal{F}$, or *framework* for short, that embodies an axiomatisation of the problem context. $\mathcal{F}$ has a signature Σ, a set **X** of *axioms*, a set **T** of *theorems*, a list Π_F of *open symbols*, and a list Δ_F of *defined symbols*. The syntax $\Pi_F \Rightarrow \Delta_F$ indicates that the axioms of $\mathcal{F}$ fix (the meaning of) the symbols Δ_F whenever $\mathcal{F}$ is composed with frameworks that fix Π_F. The defined and open symbols belong to the signature Σ, which may also contain *closed symbols*, namely symbols

defined completely by the axioms (i.e., independently from Π_F). Frameworks are explained in Section 2.1, and framework composition is explained in Section 3.1.

In the middle, we have the *specification section*. Its role is to bridge the gap between the framework $\mathcal{F}$ and the chosen *programming language*. So far, we have considered only logic programs, and the corresponding specification formalism is explained in Section 2.2. The specification section contains the specifications $S_{p_1}, \ldots, S_{p_n}$ of the program predicates occurring in the program section. It may also contain a set of *specification reduction theorems* theorems $RD_1, \ldots, RD_k$, that are useful to reason about correct reuse. Specification reduction is explained in Section 3.2.

At the bottom, we have the *program section*. Programs are open logic (or constraint logic) programs. An open program $P_{id_i} : S_{\pi_i} \Rightarrow S_{\delta_i}\ \{\mathcal{C}_i\}$ $(1 \leq i \leq h)$ has an identifier id_i, an *interface specification* $S_{\pi_i} \Rightarrow S_{\delta_i}$ and a set $\{\mathcal{C}_i\}$ of *implementation clauses*. S_{π_i} and S_{δ_i} are lists of specifications defined in the specification section. An interface specification contains all the information needed to *correctly reuse* a *correct* program. Programs and correctness are explained in Section 2.3. Correct reuse is explained in Section 3.3.

2.1 Specification Frameworks

A specification framework $\mathcal{F}$ is defined in the context of first-order logic, and contains the *relevant knowledge* of the necessary concepts and data types for building a model of the application at hand.

We distinguish between *closed* and *open* frameworks. A *closed framework* $\mathcal{F} = \langle \Sigma, \mathbf{X}, \mathbf{T} \rangle$ has a signature Σ, a set $\mathbf{X}$ of axioms, and a set $\mathbf{T}$ of theorems. It has no open and defined symbols, that is, all the symbols of Σ are closed.

Example 1. An example of closed framework is first-order arithmetic $\mathcal{NAT} = \langle \Sigma_{Nat}, \mathbf{X}_{Nat}, \mathbf{T}_{Nat} \rangle$, introduced by the following syntax:[3]

Framework $\mathcal{NAT}$;

DECLS: $Nat : sort;$
$0 : [\,] \rightarrow Nat;$
$s : [Nat] \rightarrow Nat;$
$_+_, _*_ : [Nat, Nat] \rightarrow Nat;$

AXS: $Nat : construct(0, s : Nat);$
$+ :\quad i + 0 = i;$
$i + s(j) = s(i + j);$
$:\quad i * 0 = 0;$
$i * s(j) = i * j + i;$

THMS: $i + j = j + i;$
$\ldots$

[3] In all the examples, we will omit the outermost universal quantifiers, but their omnipresence should be implicitly understood.

The signature Σ_{Nat}, introduced in the declaration section DECLS, is the signature of Peano's arithmetic. The axioms $\mathbf{X}_{Nat}$, introduced in the AXS section, are the usual ones of first-order arithmetic. 0 and s are the *constructors* of *Nat* and their axioms, which we call the *constructor axioms* for *Nat*, are collectively indicated by $construct(0, s : Nat)$. The latter contains Clark's equality theory [35] for 0 and s, as well as all the instances of the first-order induction schema. $\mathcal{NAT}$ has been widely studied, and there are a lot of known theorems (in section THMS), including for example the associative, commutative and distributive laws.

Theorems are an important part of a framework. However, they are not relevant in the definitions that follow, so we will not refer to them explicitly here.

For closed frameworks we adopt isoinitial semantics, that is, we choose the intended model of $\mathcal{F} = \langle \Sigma, \mathbf{X} \rangle$ to be a *reachable isoinitial model*, defined as follows:

Definition 1 (Reachable Isoinitial Model [5]). *Let* $\mathbf{X}$ *be a set of* Σ*-axioms. A* Σ*-structure* $\textsc{i}$ *is an* isoinitial model *of* $\mathbf{X}$ *iff, for every model* $\textsc{m}$ *of* $\mathbf{X}$*, there is a unique isomorphic embedding* $i : \textsc{i} \rightarrow \textsc{m}$*.*

A model $\textsc{i}$ *is* reachable *if its elements can be represented by ground terms.*

Definition 2 (Adequate Closed Frameworks [30]). *A closed framework* $\mathcal{F} = \langle \Sigma, \mathbf{X} \rangle$ *is* adequate *iff there is a reachable isoinitial model* $\textsc{i}$ *of* $\mathbf{X}$ *that we call 'the' intended model of* $\mathcal{F}$*.*

In fact $\textsc{i}$ is one of many intended models of $\mathcal{F}$, all of which are isomorphic. So $\textsc{i}$ is unique up to isomorphism, and hence our (ab)use of 'the'.

As shown in [5], adequacy entails the computability of the operations and predicates of the signature.

Example 2. $\mathcal{NAT}$ is an adequate closed framework. Its intended model is the standard structure $\mathcal{N}$ of natural numbers ($\mathcal{N}$ is a reachable isoinitial model of $\mathbf{X}_{Nat}$). $\mathcal{N}$ interprets *Nat* as the *set of natural numbers*, and $s, +$ and $*$ as the *successor*, *sum* and *product* function, respectively.

The adequacy of a closed framework is not a decidable property. We have the following useful proof-theoretic characterisation, which can be seen as a "richness requirement" implicit in isoinitial semantics [31]:

Definition 3 (Atomic Completeness). *A framework* $\mathcal{F} = \langle \Sigma, \mathbf{X} \rangle$ *is* atomically complete *iff, for every ground atomic formula* A*, either* $\mathbf{X} \vdash A$ *or* $\mathbf{X} \vdash \neg A$*.*

Theorem 1 (Adequacy Condition [38]). *A closed framework* $\mathcal{F} = \langle \Sigma, \mathbf{X} \rangle$ *is adequate iff it has at least one reachable model and is atomically complete.*

Closed adequate frameworks can be built incrementally, starting from a closed adequate kernel, by means of *adequate extensions*.

Definition 4 (Adequate Extensions [30]). *An* adequate extension *of an adequate closed framework* $\mathcal{F} = \langle \Sigma, \mathbf{X} \rangle$ *is an adequate closed framework* $\mathcal{F}_\delta = \langle \Sigma \cup \delta, \mathbf{X} \cup D_\delta \rangle$ *such that:*

a) D_δ *is a set of* $(\Sigma \cup \delta)$*-axioms, axiomatising a set of* new *(i.e., not in* Σ*) symbols* δ*;*
b) the Σ*-reduct* $\text{I}|\Sigma$ *of the intended model* I *of* $\mathcal{F}_\delta$ *is the intended model of* $\mathcal{F}$*.*

The notions of *reduct* and *expansion* are standard in logic [4]. The Σ-reduct $\text{I}' = \text{I}|\Sigma$ forgets the interpretation of the symbols not in Σ, in our case the new symbols δ. Conversely, I is said to be a $(\Sigma \cup \delta)$-*expansion* of I', that is, a $(\Sigma \cup \delta)$-expansion is a $(\Sigma \cup \delta)$-interpretation that preserves the meaning of the old Σ-symbols, and interprets the new δ arbitrarily.

In Definition 4, by b), the intended model I of an adequate extension is an expansion of the old intended model, that is, adequacy entails that the meaning of the old symbols is preserved.

If the axioms D_δ of an *adequate* extension are explicit definitions, we say that they are *adequate explicit definitions*. Since they are important in our approach, we briefly recall them.

An explicit definition of a new relation r has the form $\forall \underline{x} \bullet r(\underline{x}) \leftrightarrow R(\underline{x})$, where $\underline{x}$ indicates a tuple of variables and (as usual) "$\bullet$" extends the scope of a quantifier to the longest subformula next to it. The explicit definition of a new function f has the form $\forall \underline{x} \bullet F(\underline{x}, f(\underline{x}))$, where $R(\underline{x})$ and $F(\underline{x}, y)$ are formulas of the framework that contain free only the indicated variables. The explicit definition of f has the *proof obligation* $\mathbf{X} \vdash \forall \underline{x} \bullet \exists! y \bullet F(\underline{x}, y)$, where $\mathbf{X}$ are the framework axioms (as usual, $\exists! y$ means unique existence). $R(\underline{x})$ is called the *definens* (or *defining formula*) of r, and $F(\underline{x}, y)$ the *definiens* (or *defining formula*) of f.

Explicit definitions have nice properties. They are *purely* declarative, in the following sense: they define the new symbols *purely in terms of the old ones*, that is, in a *non-recursive* way. This declarative character is reflected by the following *eliminability* property, where Σ is the signature of the framework and δ are the new explicitly defined symbols: the extension is conservative (i.e., no new Σ-theorem is added) and every formula of $\Sigma + \delta$ is provably equivalent to a corresponding formula of the old signature Σ. Moreover, if we start from a sufficiently expressive kernel, most of the relevant relations and functions can be explicitly defined. Finally, we can prove:

Proposition 1. *If the definiens of an explicit definition is quantifier-free, then the definition is adequate.*

If the definiens is not quantifier-free, adequacy must be checked. To state the adequacy of closed frameworks and of explicit definitions, we can apply proof methods based on logic program synthesis [26,27] or constructive logic [38].

Example 3. The kernel $\mathcal{NAT}$ of Example 1 is sufficiently expressive in the following sense. Every recursively enumerable relation r can be introduced by an

explicit definition.[4] For example, we can define the ordering relations $\leq$ and $<$ by the explicit definitions:

$$D_{\leq} : i \leq j \leftrightarrow \exists k \bullet i + k = j;$$
$$D_{<} : i < j \leftrightarrow i \leq j \land \neg i = j.$$

Since the outermost universal quantifiers are implicitly present, $D_{\leq}$ is the closed formula $\forall i, j \bullet i \leq j \leftrightarrow \exists k \bullet i + k = j$ (similarly, $D_{<}$ is understood to be universally closed).

Since the definiens $\exists k \bullet i + k = j$ of $\leq$ is quantified, adequacy of $D_{\leq}$ must be checked. It can be proved by logic program synthesis, as follows.

(a) We derive the following clauses in $\mathcal{NAT} + D_{\leq}$:

$$\begin{aligned} P_{\leq} : \quad\quad 0 \leq i &\leftarrow \\ s(i) \leq s(j) &\leftarrow i \leq j. \end{aligned}$$

(b) In $\mathcal{NAT} + D_{\leq}$ we prove the *only-if part* of the completed definition [35] of $\leq$ in $P_{\leq}$ (the *if part* is guaranteed by a)).
(c) Finally, we prove that $P_{\leq}$ existentially terminates, i.e., for every ground atom A, the goal $\leftarrow A$ finitely fails or has at least one successful derivation (with program $P_{\leq}$).

By (a), (b) and (c) we get ([27], Theorem 11) that the extension by $D_{\leq}$ is adequate. By the way, adequacy entails that the new predicate $\leq$ is computable. We do not have to check the adequacy of $D_{<}$, because its definiens is quantifier free. $D_{<}$ uses $\leq$. However, an explicit definition of $<$ and a proof of its adequacy can be given directly in $\mathcal{NAT}$, by the eliminability of explicit definitions. Thus we could define $<$ first, prove its adequacy, and then define $\leq$ on top of $<$. That is, the order of explicit definitions is not relevant.

We can explicitly define functions, for example the integer square root *sqrt*:

$$D_{sqrt} : sqrt(i) * sqrt(i) \leq i \land i < s(sqrt(i)) * s(sqrt(i)).$$

The *proof obligation* $\forall i \bullet \exists! j \bullet j * j \leq i \land i < s(j) * s(j)$ can be proved in $\mathcal{NAT}$ by induction. Adequacy follows from the fact that the definiens $j * j \leq i \land i < s(j) * s(j)$ is quantifier free.

An *open framework* $\mathcal{F}(\Pi \Rightarrow \Delta) = \langle \Sigma, \mathbf{X} \rangle$ represents an incomplete axiomatisation. It has a non-empty import list Π, containing the symbols left *open* by the axioms, and a (possibly empty) disjoint export list Δ, containing the symbols that are *defined* by the axioms, in terms of the open ones. The *closed* symbols are the symbols of the signature that are not in $\Pi \cup \Delta$, and their meaning is fixed in a unique way by the axioms. We distinguish three sets of axioms, where Σ_K is the sub-signature of the closed symbols:

[4] Every recursively enumerable relation is Diophantine (Matijacevic theorem [37]).

- the *kernel axioms* $\mathbf{X}_K = \mathbf{X}|\Sigma_K$ ($\ldots|\Sigma_K$ is the subset of the axioms with symbols from Σ_k); the kernel axioms axiomatise the closed symbols, that is, $\mathcal{F}_K = \langle \Sigma_K, \mathbf{X}_K \rangle$ must be an adequate closed framework, that we call the *closed kernel*;
- the *constraints* $\mathbf{X}_C = (\mathbf{X}|(\Sigma_K \cup \Pi)) \setminus \mathbf{X}_K$, which constrain the possible interpretations of the open symbols Π;
- and the *definition axioms* $\mathbf{X}_D = \mathbf{X} \setminus (\mathbf{X}_K \cup \mathbf{X}_C)$, which fix the meaning of the defined symbols Δ, in terms of the open and closed symbols.

Example 4. The following open framework axiomatises lists with generic elements X and a generic total ordering $\lhd$ on X. From now on, in the examples, the variables of sort X will begin with x, y, z, w, those of sort *Nat* with i, j, h, k, and those of sort *ListX* with l, m, n, o.

Framework $\mathcal{LIST}(X, \lhd \;\Rightarrow\; ListX, nil, ., @, nocc)$;

KERNEL: $\mathcal{NAT}$;

DECLS:
$$\begin{aligned}
X &: sort;\\
ListX &: sort;\\
_ \lhd _ &: [X, X];\\
nil &: [\,] \to ListX;\\
_._ &: [X, ListX] \to ListX;\\
_@(_,_) &: [X, Nat, ListX];\\
nocc &: [X, ListX] \to Nat;
\end{aligned}$$

DEFAXS: $ListX : construct(nil, . : ListX)$;
$$\begin{aligned}
@ :\quad & x@(0,l) \leftrightarrow \exists y, m \,\bullet\, l = y.m \wedge x = y;\\
& x@(s(i),l) \leftrightarrow \exists y, m \,\bullet\, l = y.m \wedge x@(i,m);\\
nocc :\quad & nocc(x, nil) = 0;\\
& x = y \to nocc(x, y.l) = nocc(x,l) + 1;\\
& \neg x = y \to nocc(x, y.l) = nocc(x,l);
\end{aligned}$$

CONSTRS: $\lhd : TotalOrdering(\lhd)$.

The signature Σ_{Nat}, the axioms $\mathbf{X}_{Nat}$ and the theorems $\mathbf{T}_{Nat}$ of the imported *kernel* $\mathcal{NAT}$ are automatically included. In the *definition axioms* DEFAXS, *nil* and "." are the *list constructors*, as indicated by $construct(nil, . : ListX)$, which contains Clark's equality theory and structural induction on constructors; $x@(i,l)$ means that the element x occurs at position i in the list l, where positions start from 0; $nocc(x,l)$ is the number of occurrences of the element x in the list l; by the *constraint axioms* CONSTRS, $\lhd$ is a total ordering relation.

To specify the basic operations on the ADT of lists, the closed kernel $\mathcal{NAT}$ is not necessary. We have imported it for specification and reasoning purposes. Indeed, by using natural numbers we can introduce @ and *nocc*. The resulting language and axiomatic system give a rich starting framework, which allows us to explicitly define the usual operations on lists and ordered lists, and to reason about them (see Example 6).

An open framework $\mathcal{F}$ has a class of not necessarily isomorphic intended models, since $\mathbf{X}_K \cup \mathbf{X}_C$ allows many $(\Sigma_K \cup \Pi)$-interpretations, that we call *pre-models*. The semantics considered here is a variant of the one presented in [30]. A pre-model is an expansion of the intended model of the kernel that satisfies the constraints $\mathbf{X}_C$. For every pre-model P, the axioms of $\mathcal{F}$ fix a corresponding intended P-model I_P, defined as follows.

A P-model of $\mathcal{F}$ is a Σ-model M of $\mathbf{X}$ such that $\mathrm{M}|(\Sigma_K \cup \Pi) = \mathrm{P}$, that is, M coincides with P over the closed and open symbols. Since Π may contain open sorts, we consider Π-reachable models, where Π-reachability is reachability in an expansion containing a new constant for each element of each open sort.

Definition 5 (P-isoinitial Models). *A* P*-model* I *is a* P-isoinitial model of $\mathcal{F}$ *iff, for every* P*-model* M*, there is a unique isomorphic embedding* $i : \mathrm{I} \rightarrow \mathrm{M}$ *such that* i *is the identity over the open sorts.*

Definition 6 (Adequate Open Frameworks and Intended Models). *An open framework* $\mathcal{F}$ *is* adequate *iff, for every pre-model* P *of* $\mathcal{F}$*, there is a* Π*-reachable* P*-isoinitial model* I_P*, that we call the intended* P*-model of* $\mathcal{F}$*.*

M *is an* intended model *of* $\mathcal{F}$ *iff there is a pre-model* P *of* $\mathcal{F}$ *such that* M *is the intended* P*-model of* $\mathcal{F}$*.*

For every pre-model P, the intended P-model is unique up to isomorphism. Intended models with non-isomorphic pre-models are, of course, non-isomorphic. We consider closed frameworks as a limiting case, where the kernel coincides with the whole framework and the unique intended model coincides with the unique pre-model.

Example 5. $\mathcal{LIST}$ is an adequate open framework. In it, a pre-model P coincides with $\mathcal{N}$ for the kernel signature Σ_{Nat} and interprets X as any set with a total ordering $\lhd$. The intended P-model of $\mathcal{LIST}$ interprets $ListX$ as the set of the finite lists with elements from X, and the other defined symbols in the way already explained in Example 4.

Adequate open frameworks can be built incrementally, by *adequate extensions*, where the intended models of an adequate extension $\mathcal{F}'$ of a framework $\mathcal{F}$ are expansions of intended models of $\mathcal{F}$.

Definition 7 (Adequate Extensions). *A framework* $\mathcal{F}'$ *is an* adequate extension *of an adequate open framework* $\mathcal{F}$ *iff* $\mathcal{F}'$ *is an adequate open or closed framework, the signature and the axioms of* $\mathcal{F}'$ *contain those of* $\mathcal{F}$*, the kernel signature of* $\mathcal{F}'$ *contains the kernel signature of* $\mathcal{F}$*, and for every intended model* I' *of* $\mathcal{F}'$*, the reduct* $\mathrm{I}'|\Sigma$ *is an intended model of* $\mathcal{F}$*.*

In the limiting case, an adequate extension $\mathcal{F}'$ of an open framework $\mathcal{F}$ may be a closed framework. In this case, we say that $\mathcal{F}'$ is an *instance* of $\mathcal{F}$, and the axioms that "instantiate" (i.e., close) the open symbols are called *closure axioms*. A set of closure axioms is called a *closure*. Closures will be considered in Section 3.1, together with other framework operations.

In general, the adequacy of an extension is not decidable, but we may have different kinds of extensions, with different adequacy conditions. In particular, we distinguish:

- Parameter extensions. In this case new parameters and/or new constraints are added. Parameter extensions are adequate iff, adding new constraints, consistency is preserved.
- Defined symbol extensions. In this case new defined symbols, together with the corresponding definition axioms, are added. Adequate explicit definitions are still useful for introducing new defined symbols, and adequacy can be stated in a way similar to those mentioned before for closed framework extensions (by program synthesis or constructive logic [27,38]). Proposition 1 still holds.
- Kernel extensions. In this case the closed kernel is extended by new closed symbols, as already shown for closed frameworks.

Example 6. The framework $\mathcal{LIST}(X, \lhd \;\Rightarrow\; ListX, nil, ., @, nocc)$ can be obtained by extending the framework $\mathcal{LIST}(X \;\Rightarrow\; ListX, nil, ., @, nocc)$, without $\lhd$ and without constraint axioms, by the *parameter* $\lhd : [X, X]$ constrained by $TotalOrdering(\lhd)$. The *kernel* $\mathcal{NAT}$ can be extended by explicitly defining the most useful operations and predicates on natural numbers. The *defined symbols* can be extended by the relevant operations on lists, by means of explicit definitions. For example, the definitions of list membership, length, concatenation and permutation are:[5]

$$
\begin{array}{ll}
D_\in & : x \in l \leftrightarrow nocc(x,l) > 0 \\
D_{len} & : \forall i \,\bullet\, (\exists x \,\bullet\, x@(i,l)) \leftrightarrow i < len(l) \\
D_| & : \forall i, x \,\bullet\, (i < len(l) \rightarrow (x@(i,l) \leftrightarrow x@(i, l|m))) \quad \wedge \\
 & \qquad\quad (len(l) \leq i \rightarrow (x@(i,m) \leftrightarrow x@(i + len(l), l|m))) \\
D_{perm} & : perm(l,m) \leftrightarrow \forall x \,\bullet\, nocc(x,l) = nocc(x,m)
\end{array}
$$

$D_\in$ gives rise to an adequate extension, because its definiens is quantifier free. The definiens of D_{len} is $\forall i \,\bullet\, (\exists x \,\bullet\, x@(i,l)) \leftrightarrow i < k$ and the proof obligation requires a proof of $\forall l \,\bullet\, \exists! k \,\bullet\, \forall i \,\bullet\, (\exists x \,\bullet\, x@(i,l)) \leftrightarrow i < k$. Since the definiens is quantified, adequacy must be checked (and can be proved), by constructive proofs or by program synthesis. Adequacy must be checked (and can be proved) also for $D_|$ and D_{perm}.

Using $\lhd : [X, X]$, we can also define operations on ordered lists, like $l \lhd_L m$ (lexicographic ordering on lists), $ord(l)$ (l is an ordered list), and so on. Their properties can be proved using the total ordering constraints. For example, we can prove that the lexicographic ordering $\lhd_L$ is, in turn, a total ordering.

2.2 Specifications

In a compositional unit **K**, specifications assume their proper meaning only in the context of the framework $\mathcal{F}$. In this section we define formally what we

[5] In D_{len} and $D_|$, the universal quantifiers of the definiens have not been omitted.

mean by specifications in $\mathcal{F}$ and we show some examples. We maintain a strict distinction between specification frameworks and (program) specifications and, to distinguish the function and relation symbols of the framework from those computed by programs, the latter will be called (program) *operations*.

Definition 8 (Specifications and S-expansions). *Let* $\mathcal{F}(\Pi \Rightarrow \Delta) = \langle \Sigma, \mathbf{X} \rangle$ *be a framework. A* specification S_ω in (the context of) $\mathcal{F}$ *is a set of closed* $(\Sigma + \omega)$*-formulas, that define a set of operations* ω *in terms of* $\mathcal{F}$.

An S_ω-expansion *of a model* M *of* $\mathcal{F}$ *is a* $(\Sigma + \omega)$*-expansion* M' *of* M *such that* $\mathrm{M}' \models S_\omega$.

That is, S_ω can be interpreted as an *expansion operator* that associates with every intended model of $\mathcal{F}$ the corresponding S_ω-expansions, namely the expansions that interpret the specified operations according to S_ω.

Definition 9 (Strict Specifications). *A specification* S_ω *is* strict *in a framework* $\mathcal{F}$*, if, for every model* M *of* $\mathcal{F}$*, there is only one* S_ω*-expansion. It is* non-strict *otherwise.*

Now we list different kinds of strict and non-strict specifications considered in [28], essentially based on explicit definitions. The specification formalism considered here is tailored to logic programs with definite clauses in a many-sorted signature. Program semantics is based on *minimum Herbrand models*, where *program data* (those used in programs) coincide with ground terms. We assume that the signature Σ_D of program data is *pre-defined* by the framework $\mathcal{F}$, and that, for every closed or defined sort s of Σ_D, $\mathcal{F}$ contains the axioms $construct(c_1, \ldots, c_n : s)$, where $c_1, \ldots, c_n$ are the constructors of s. They are the unique operations of sort s that can be used in logic programs. This assumption concerns Herbrand models of standard logic programs, where $construct(\ldots)$ holds, but our treatment readily extends to the specification formalism for constraint logic programs by assuming that Σ_D is the constraint signature, and is pre-defined by the framework.

Since in logic programs only *program predicates* are not pre-defined, we have to specify only them. There are different forms of specifications.

If-and-Only-if Specifications. An *if-and-only-if specification* in a framework $\mathcal{F}$ is an explicit definition of a new predicate r:

$$S_r : \quad \forall \underline{x} \bullet r(\underline{x}) \leftrightarrow R(\underline{x})$$

By the well known properties of explicit definitions, for every model M of the framework $\mathcal{F}$, there is only one S_r-expansion of M, that is, S_r is strict.

Example 7. In $\mathcal{NAT}$ we can specify, for example, the following predicates:

$$\begin{array}{ll} S_{div} & : div(i,j,h,k) \leftrightarrow i = j * h + k \wedge k < j; \\ S_{divides} & : divides(i,j) \leftrightarrow \exists h \bullet div(j,i,h,0); \\ S_{prime} & : prime(i) \leftrightarrow \forall j \bullet divides(j,i) \rightarrow j = 1 \vee j = i; \end{array}$$

Super-and-sub Specifications. A *super-and-sub specification* in a framework $\mathcal{F}$ is of the form

$$S_r : \quad \forall \underline{x} \bullet (R_{sub}(\underline{x}) \rightarrow r(\underline{x})) \wedge (r(\underline{x}) \rightarrow R_{super}(\underline{x}))$$

where $R_{sub}(\underline{x})$ and $R_{super}(\underline{x})$ are two formulas of $\mathcal{F}$ such that $\mathcal{F} \vdash \forall \underline{x} \bullet R_{sub}(\underline{x}) \rightarrow R_{super}(\underline{x})$.

The implication $\forall \underline{x} \bullet R_{sub}(\underline{x}) \rightarrow R_{super}(\underline{x})$ is satisfied by the models of $\mathcal{F}$. Therefore, in every intended model I, the relation R_{sub} in I, i.e., the set of values $\underline{\mathbf{x}}$ such that $\mathrm{I} \models R_{sub}(\underline{\mathbf{x}})$, is a sub-relation of the relation R_{super}, and the specified relation r is any relation that is a super-relation of R_{sub} but is a sub-relation of R_{super}.

Conditional Specifications. A *conditional specification* of a new relation r in a framework $\mathcal{F}$ has the form:

$$\forall \underline{x}, \underline{y} \bullet IC(\underline{x}) \rightarrow (r(\underline{x}, \underline{y}) \leftrightarrow R(\underline{x}, \underline{y})) \tag{1}$$

where $IC(\underline{x})$ is the *input condition*, and $R(\underline{x}, \underline{y})$ is the *input-output condition.* Both $IC(\underline{x})$ and $R(\underline{x}, \underline{y})$ are formulas of $\mathcal{F}$. (1) specifies $r(\underline{x}, \underline{y})$ only when the input condition $IC(\underline{x})$ is true, while nothing is required if the input condition is false. That is, $IC(\underline{x})$ states that $r(\underline{x}, \underline{y})$ is to be called *only* in contexts that make it true. This fact allows us to assume $IC(\underline{x})$ when reasoning about correct reuse, as shown in Section 3.2.

(1) is equivalent to the following super-and-sub specification, which allows us to apply the results of [34] in correctness proofs:

$$\forall \underline{x}, \underline{y} \bullet (IC(\underline{x}) \wedge R(\underline{x}, \underline{y}) \rightarrow r(\underline{x}, \underline{y})) \wedge (r(\underline{x}, \underline{y}) \rightarrow \neg IC(\underline{x}) \vee R(\underline{x}, \underline{y}))$$

Example 8. In the open framework $\mathcal{LIST}(X, \lhd \Rightarrow ListX, nil, ., @, nocc)$, we have for example the following specification:

$$\begin{array}{ll} S_{sort} & : sort(l, m) \leftrightarrow perm(l, m) \wedge ord(m); \\ S_{merge} & : ord(l) \wedge ord(m) \rightarrow (merge(l, m, o) \leftrightarrow ord(o) \wedge perm(l|m, o)); \\ S_{split} & : (len(l) > 1 \wedge split(l, m, n) \rightarrow perm(l, m|n) \wedge len(m) < len(l) \wedge \\ & \quad len(n) < len(l)) \ \wedge \ (len(l) > 1 \rightarrow \exists m, n \bullet split(l, m, n)); \end{array}$$

S_{sort} is an if-and-only-if specification, S_{merge} is a conditional specification. By the input condition, $merge(l, m)$ is to be called only in contexts where the input lists are ordered. If they are not, o is not required to be ordered. S_{split} is an example of another form of non-strict specification (called a *selector specification*) that we do not discuss here (see [28]).

2.3 Interface Specifications, Programs, and Correctness

Here we consider correctness of (logic) programs with respect to interface specifications. In Section 3.2, we will consider the role of interface specifications in correct reuse. We start by introducing some terminology.

The *signature* Σ_P of a program P contains the declarations of its predicate, constant and function symbols, and the sorts occurring in such declarations. The *data signature* of P is the subsignature of its sort, constant and function symbols. According to the previous section, the data signature belongs to the framework signature. We will distinguish *open* and *closed* programs, as follows.

The *defined predicates* of a program P are those that occur in the head of at least one clause of P, while the (possible) *open predicates* of P are those that occur only in the body. A program P is *open* if its signature contains at least one open sort or predicate. It is *closed* if no open symbol belongs to its signature.

A *interface specification* for an *open* program P is of the form $S_\pi \Rightarrow S_\delta$, where S_π are specifications of a set π of predicates that includes all the open predicates of P, and S_δ are specifications of a set δ of predicates that are included in the defined predicates of P. We will write $P : S_\pi \Rightarrow S_\delta$ to indicate that P has specification $S_\pi \Rightarrow S_\delta$. If P has no open predicates, then S_π will be empty. In this case, we write $P : \Rightarrow S_\delta$.

Example 9. In a compositional unit with open framework $\mathcal{LIST}$, we can declare the following open sorting program (where S_{split}, S_{merge} and S_{sort} are as shown in Example 8):

$$
\begin{array}{l}
\text{Program } P_{sort} : S_{split}, S_{merge} \Rightarrow S_{sort} \\
\{ \\
\quad sort(nil, nil) \leftarrow \\
sort(x.nil, x.nil) \leftarrow \\
\quad sort(x.y.l, o) \leftarrow split(x.y.l), m, n), \\
\qquad\qquad\qquad\quad sort(m, m_1), sort(n, n_1), \\
\qquad\qquad\qquad\quad merge(m_1, n_1, o). \\
\}
\end{array}
$$

Programs may be open independently from the framework, i.e., closed frameworks may contain open programs. For example, in the closed framework $\mathcal{NAT}$, we can declare:

$$
\begin{array}{l}
\text{Program } P_{prod} : S_{sum} \Rightarrow S_{prod} \\
\{ \\
prod(i, 0, 0) \quad\;\; \leftarrow \\
prod(i, s(j), h) \leftarrow prod(i, j, k), sum(k, i, h). \; \} \\
\{
\end{array}
$$

where:

$$
\begin{array}{l}
S_{sum} : sum(x, y, z) \leftrightarrow z = x + y; \\
S_{prod} : prod(x, y, z) \leftrightarrow z = x \cdot y.
\end{array}
$$

Now we can define program correctness. We will first explain the correctness of closed programs in closed frameworks, because it is simpler and more intuitive. Then we introduce correctness of open programs.

Correctness of Closed Programs. A closed program P has only defined predicates, and an interface specification of P is of the form $\Rightarrow S_\delta$. For simplicity, we will consider the case of interface specifications $\Rightarrow S_r$ with one defined predicate r (the extension to $\Rightarrow S_{r_1}, \ldots, S_{r_k}$, with $k > 1$, is immediate). We define program correctness in a closed framework as follows:

Definition 10 (Correctness of Closed Programs). *Let $\mathcal{F}$ be a closed framework with intended model* I*. Let S_r be a specification of a predicate r, P be a program that computes r, and* H *be the minimum Herbrand of P. P is* correct *with respect to the interface specification $\Rightarrow S_r$ iff the interpretation of r in* H *coincides with the interpretation of r in one of the S_r-expansions of* I.

For conciseness, we will say that $P : \Rightarrow S_r$ is correct, to indicate that P is correct with respect to $\Rightarrow S_r$.

For a strict specification S_r, there is only one S_r-expansion of I, that is, the new symbol r defined by S_r has a unique interpretation in I, and one in H. Correctness of $P : \Rightarrow S_r$ means that the two interpretations of r coincide, or, at least, are isomorphic. This is illustrated in Figure 2.

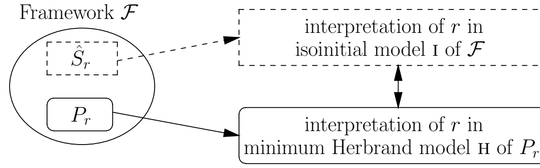

Fig. 2. Strict specifications.

If S_r is not strict, then r has many interpretations with respect to I. Correctness of $P : \Rightarrow S_r$ in this case means that the interpretation of r in H coincides with one of the interpretations of r with respect to I. This is illustrated in Figure 3.

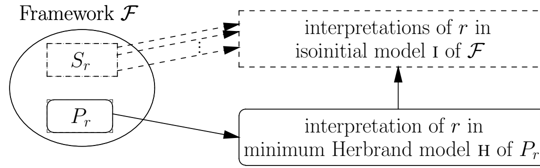

Fig. 3. Non-strict specifications.

Steadfastness: Correctness of Open Programs. Now we consider open programs in open frameworks, and we discuss the associated notion of correctness. An open program is correct if it behaves as expected in all the circumstances. We called this property *steadfastness* [34].

The correctness relation between an interface specification $S_\pi \Rightarrow S_\delta$ and an open program P cannot be defined as in Definition 10, because open frameworks may have many intended models and we cannot use minimum Herbrand models as the semantics of open programs, since in the minimum Herbrand models, open relations are assumed to be empty, and therefore cannot play the role of parameters. So in [34] we introduced *minimum* J*-models*, together with the notion of *steadfastness*, to serve as the basis for a model and proof theory of the correctness of open programs.

Here we first recall the definition of steadfastness informally, and then define correctness of open programs, and give its relevant properties. A *pre-signature* for an open program P is a signature Ω that contains the data signature and the open predicates of P, but *not* the defined predicates of P. A *pre-interpretation in* Ω is an Ω-interpretation. That is, symbols of Ω are considered to be open, and a pre-interpretation J interprets them arbitrarily. In contrast, the intended meaning of the defined predicates of P is stated by its clauses, in terms of J.

Let P be a program with defined predicates δ. To define the intended meaning of δ in a pre-interpretation J, we introduce J-models. A J-model of P is a model M of P, such that $\text{M} \mid \Omega = \text{J}$, i.e., M coincides with J over Ω. Since two distinct J-models M and N differ only for the interpretation of δ, we can compare them by looking at δ: we say that M is contained in N, written $\text{M} \subseteq_\delta \text{N}$, iff the interpretation of (the predicates of) δ in M is contained in that of δ in N. We can show that a minimum J-model (with respect to $\subseteq_\delta$) exists. The minimum J-model of P will be indicated by J^P, and it represents the interpretation of δ stated by the program P, in the pre-interpretation J.

Using minimum J-models, steadfastness in an interpretation can be defined as follows.

Definition 11 (Steadfastness). *Let P be an open program, Ω be a pre-signature for P, and r be a predicate defined by P. P is* steadfast *for r in a $(\Omega + r)$-interpretation I if and only if the interpretation of r in its minimum $\text{I} \mid \Omega$-model coincides with the interpretation of r in I.*

More intuitively, steadfastness in I for r means that the interpretation of r in I coincides with the interpretation of r stated by P, when the open symbols Ω are interpreted as in I (i.e., when the pre-interpretation is $\text{I} \mid \Omega$). Consider for example the open program P in the context of $\mathcal{NAT}$:

$$r(x) \leftarrow p(z, x)$$

where x and z are of sort Nat. $\Sigma_{Nat} \cup \{p : [Nat, Nat]\}$ is a pre-signature for this program. Consider the interpretation I_1 where Σ_{Nat} is interpreted as in $\mathcal{NAT}$, $r(x)$ means "*x is even*", and $p(z, x)$ means "$z + z = x$". If we interpret p as in I_1, we can easily see that the interpretation of $r(x)$ in the corresponding minimum

model of P coincides with the interpretation of $r(x)$ in I_1, i.e., P is steadfast in I_1. Similarly, if we consider I_2 that interprets $p(z,x)$ as "$z * z = x$", to get steadfastness I_2 has to interpret $r(x)$ as "*x is a perfect square*".

Correctness is steadfastness in the expansions of the intended models stated by the interface specification:

Definition 12 (Correctness of Open Programs). *Let $\mathcal{F}$ be an open framework, and $P : S_\pi \Rightarrow S_\delta$ be an open program. $P : S_\pi \Rightarrow S_\delta$ is* correct *in $\mathcal{F}$ iff for every intended model I of $\mathcal{F}$ and every S_π-expansion I_π of I, there is a S_δ-expansion $\mathrm{I}_{\pi,\delta}$ of I_π, such that P is steadfast in $\mathrm{I}_{\pi,\delta}$ for the predicate symbols of δ.*

The intuitive meaning of the previous definition is the following: $\Sigma + \pi$ is a pre-signature for P, and I_π is a pre-interpretation that interprets the data signature according to $\mathcal{F}$ and the open symbols according to S_π, i.e., I_π represents a legal parameter passing. Steadfastness of P in $\mathrm{I}_{\pi,\delta}$ means that the interpretation of δ stated by P for the parameter passing I_π is correct with respect to S_δ.

The following important properties of correct reusability hold (see [34]):

Proposition 2 (Inheritance). *Let $\mathcal{F}'$ be an adequate extension of an adequate (open) framework $\mathcal{F}$. If $P : S_\pi \Rightarrow S_\delta$ is correct in $\mathcal{F}$, then it is correct in $\mathcal{F}'$.*

As we will show in Section 3, framework composition can be treated in terms of extension. Therefore inheritance yields a first level of correct reusability, namely reusability of correct programs through framework composition, extension and instantiation. This level of correct reusability would not be important, however, if we could not guarantee the correctness of the composition of the inherited open programs. This second, important level of correct reusability will be called *compositionality*. In compositionality, interface specifications play a central role, as shown by the following theorems:

Theorem 2 (Compositionality). *If $P : S_{\pi_1}, S_{\delta_2} \Rightarrow S_{\delta_1}$ and $Q : S_{\pi_2} \Rightarrow S_{\delta_2}$ are correct in a framework $\mathcal{F}$ and are not mutually recursive, then $P \cup Q : S_{\pi_1}, S_{\pi_2} \Rightarrow S_{\delta_1}, S_{\delta_2}$ is correct in $\mathcal{F}$.*

As we can see, interface specifications indicate how programs can be composed to correctly interact. Theorem 2 can be extended to mutually recursive programs, but in this case we have to check that open termination [34] is preserved. By inheritance and compositionality we get reusability, as shown by the following example.

Example 10. We can show that the open program $P : S_{split}, S_{merge} \Rightarrow S_{sort}$ in Example 9 is correct in the open framework $\mathcal{LIST}$. This means that, in every instance of $\mathcal{LIST}$, $P : S_{split}, S_{merge} \Rightarrow S_{sort}$ is always correct with respect to the specification $\Rightarrow S_{sort}$, provided that it is composed with closed correct programs $Q_{merge} : \Rightarrow S_{merge}$ and $Q_{split} : \Rightarrow S_{split}$.

This example shows that compositionality corresponds to *a priori* correctness of open programs in a framework. It thus corresponds to correctness of open modules in a library. It is to be contrasted with *a posteriori* correctness, i.e., correctness established by verification after program composition.

2.4 Related Work

Specification frameworks are similar to Abstract Data Types (ADT's). ADT's became popular in the 80's and have been widely studied [47]. In general, they are based on the initial algebra approach, that is, intended models are initial models. Parametric ADT's have also been studied. These are similar to our open frameworks, even though they are technically defined in a different way. A detailed treatment of algebraic ADT's, including the parametric case, can be found, for example, in [13].

The initial algebra approach is adequate for ADT specification, where the purpose is to give the minimal signature and axioms that are needed to characterise the desired data and operations. Initial models generalise the idea of minimum Herbrand models, and always exist for algebraic ADT's and consistent Horn theories [21]. The existence of an initial model allows us to axiomatise only positive knowledge and to use (consistently) negation as failure: a fact is false if we do not have evidence of its truth. This allows for very compact axiomatisations.

In contrast, our purpose is "knowledge representation", that is, we are looking for an expressive signature and a rich set of axioms, to obtain a framework that represents our overall knowledge of a problem domain and allows us to reason about it. Isoinitial semantics requires stronger axiomatisations, and better meets our "richness requirement", compared to initial semantics. It was introduced in [5], with the purpose of giving a model-theoretic characterisation of computable ADT's.

Finally, our approach is different from the algebraic approach in the three-level architecture of our compositional units, and in the role that frameworks play in it. In this regard, we are closer to the two-tiered specification style of Larch [20], where specifications have two components: the first one is written in the Larch Shared Language LSL, and the second one in a Larch Interface Language, which is oriented to the programming language and is used to specify the interfaces between program components, i.e., the way they communicate.

We consider non-recursive definitions, like explicit or conditional definitions, to be an important tool for both extending frameworks and specifying programs. In this regard, our work is similar to [36]. At the program and specification levels, our approach is in the tradition of logic program synthesis and correctness. Our notion of correctness for closed programs is similar to the one introduced in [22]. Correctness of open programs with respect to specifications similar to our interface specifications is considered in [10]. A conditional specification is like a pre-post-condition style of specification as in VDM [24], Z [45], and B [1], except that it is declarative. Declarative conditional specifications for logic programs were introduced in [8].

3 Operations on Compositional Units

In this section we consider compositional units as building blocks for program development, that is, we focus on operations on compositional units that al-

low their correct reuse in the process of developing an application. This process starts from pre-existing compositional units, and iteratively extends them either directly, by inserting domain specific knowledge, or by reusing, i.e., incorporating, other compositional units. Reuse can, in turn, be factorised into *composition* and *extension*, which are the basic operations considered in this section. Such operations involve the framework, the specification and the program levels. We will consider the different levels separately.

3.1 Framework Reuse

To compose two units C_1 and C_2, first we compose and/or extend their frameworks $\mathcal{F}_1$ and $\mathcal{F}_2$ into a common extension $\mathcal{F}$, and then use the specifications and programs in this richer $\mathcal{F}$. Thus, in general, the framework level is the first level to be involved in operations on compositional units. Here we consider the basic operations needed at this level.

Framework Morphisms. Operations like renaming or identifying different symbols may be needed for framework reuse. This kind of operation is formalised by framework morphisms. Before introducing framework morphisms, we briefly recall signature and theory morphisms [19,13].

A *signature morphism* $\mu : \Sigma_1 \to \Sigma_2$ is a map from the symbols of Σ_1 to those of Σ_2 that preserves the declarations. Σ_2 extends Σ_1, in the following sense:

- Σ_2 contains the μ-image of Σ_1;
- every Σ_1-formula F translates into a Σ_2-formula $\mu(F)$;
- instead of Σ_1-reducts we have μ-reducts: the μ-reduct of a Σ_2-interpretation $\textsc{m}$ is the Σ_1-interpretation $\textsc{m}|\mu$ that interprets every symbol σ of Σ_1 as $\textsc{m}$ interprets the image $\mu(\sigma)$.

A *theory morphism* $\mu : \langle \Sigma_1, \mathbf{X}_1 \rangle \to \langle \Sigma_2, \mathbf{X}_2 \rangle$ is a signature morphism $\mu : \Sigma_1 \to \Sigma_2$ such that $\mu(\mathbf{X}_1)^* \subseteq \mathbf{X}_2^*$, where * denotes the proof-theoretic closure. μ works as a generalised extension, in the sense that Σ_2 contains (the μ-image of) Σ_1 and the theorems of $\mathbf{X}_2$ contain (the μ-translation of) those of $\mathbf{X}_1$.

Framework morphisms are defined as follows:

Definition 13 (Framework Morphism). *Let $\mathcal{F}(\Pi \Rightarrow \Delta) = \langle \Sigma, \mathbf{X} \rangle$ and $\mathcal{F}'(\Pi' \Rightarrow \Delta') = \langle \Sigma', \mathbf{X}' \rangle$ be two frameworks. A* framework morphism $\epsilon : \mathcal{F} \to \mathcal{F}'$ *is a theory morphism $\epsilon : \langle \Sigma, \mathbf{X} \rangle \to \langle \Sigma', \mathbf{X}' \rangle$ such that the kernel signature of $\mathcal{F}'$ contains the ϵ-image of the kernel signature of $\mathcal{F}$.*

$\mathcal{F}'$ can be considered as a generalised extension of $\mathcal{F}$. We say that it is the extension generated by the morphism $\epsilon : \mathcal{F} \to \mathcal{F}'$. Let $\mathcal{F}$ be adequate. We say that $\mathcal{F}'$ is an *adequate extension* of $\mathcal{F}$ if $\mathcal{F}'$ is adequate and, for every intended model $\textsc{i}'$ of $\mathcal{F}'$, the ϵ-reduct $\textsc{i}'|\epsilon$ is an intended model of $\mathcal{F}$.

Framework extensions considered in the previous section, that simply introduce new symbols and axioms, are a particular case. They correspond to *inclusion morphisms* ϵ that map each symbol σ into σ itself.

Example 11. A (generalised) adequate extension of $\mathcal{LIST}$ is:

Framework $\mathcal{LIST}_1(\lhd \;\Rightarrow ListNat, nil, ., @, nocc)$
EXTENDS $\mathcal{LIST}$;
CLOSE: X BY Nat;
RENAME: $ListX$ BY $ListNat$;

It is generated by the morphism ϵ defined by the clauses CLOSE and RENAME. ϵ maps the sort symbol X into Nat, the sort symbol $ListX$ into $ListNat$, and leaves the other symbols unchanged. Of course, arities and sorts in relation, function and constant declarations are translated by replacing X and $ListX$ by Nat and $ListNat$. For example, now we have $nil : [\,] \rightarrow ListNat$ and $. : [Nat, ListNat] \rightarrow ListNat$.

In $\mathcal{LIST}_1$, $ListNat$ is closed (its intended meaning is the set of finite lists of natural numbers) and the only open symbol is $\lhd : [Nat, Nat]$. A closed adequate extension can be obtained by closing $\lhd$ by

$$D_{\lhd} : \quad x \lhd y \leftrightarrow x \leq y$$

In this case, we have simply added the new axiom $D_{\lhd}$, that is, we have an inclusion morphism of $\mathcal{LIST}_1$ into a closed framework, that we will indicate by $\mathcal{LISTNAT}$.

The morphisms considered in this example are at the basis of the closure operations that we consider next.

Closure. A closure is an extension that closes the meaning of some symbols. Here, we consider closure by *internalisation*, as defined in [30]. As shown in [30], internalisation can be used to implement constrained parameter passing, as well as to introduce *objects* as the closures of suitable open frameworks that represent *classes*.

Let $\mathcal{F}(\Pi \Rightarrow \Delta) = \langle \Sigma, \mathbf{X} \rangle$ be an open framework. An *internalisation* of an open symbol is one of the following operations:

- *Sort closure*. The closure: CLOSE S BY s
  renames the open *sort* S by a sort s of the signature Σ_K of the closed kernel. No axioms are added.
- *Relation closure*. The operation: CLOSE r BY $\forall \underline{x} \bullet r(\underline{x}) \leftrightarrow R(\underline{x})$
  closes r by the new *closure axiom* $\forall x \bullet r(\underline{x}) \leftrightarrow R(\underline{x})$. The declaration of r may contain only sorts of Σ_K, and the defining formula $R(\underline{x})$ is a Σ_K-formula.
- *Function closure*. The operation: CLOSE f BY $\forall \underline{x} \bullet F(\underline{x}, f(\underline{x}))$
  closes f by the new *closure axiom* $\forall \underline{x} \bullet F(\underline{x}, f(\underline{x}))$. The declaration of f may contain only sorts of Σ_K, and the defining formula $F(\underline{x}, y)$ is a Σ_K-formula such that $\mathbf{X}_K \vdash \forall \underline{x} \bullet \exists! y \bullet F(\underline{x}, y)$.

Let $\mathcal{F}(\Pi \Rightarrow \Delta) = \langle \Sigma, \mathbf{X} \rangle$ be an open framework. A *closure by internalisation* is an internalisation that closes all the open sorts by closed sorts and all the open

relation and function symbols by a set D_Π of closure axioms, and satisfies the following *constraint satisfaction* condition:

$$\mathbf{X}_K \cup D_\Pi \vdash \mathbf{X}_C \tag{2}$$

It produces the framework $\mathcal{F}' = \langle \Sigma', (\mathbf{X} \backslash \mathbf{X}_C) \cup D_\Pi, \mathbf{T} \cup \mathbf{X}_C \rangle$ where Σ' is obtained by replacing in Σ each open sort s with the sort s' closing s, D_Π are the closure axioms of the open functions and relations, and, by (2), the constraints $\mathbf{X}_C$ have been deleted from the axioms and added to the theorems.

Example 12. $\mathcal{LISTNAT}$ of Example 11 is a closure of $\mathcal{LIST}$. It has been obtained by a sort closure and a relation closure, by the definition $D_\lhd$. Constraints are satisfied because

$$\mathbf{X}_{Nat} \cup D_\lhd \vdash \mathit{TotalOrdering}(\lhd)$$

Now the total ordering axioms $\mathit{TotalOrdering}(\lhd)$ are no longer constraints, but theorems.

A different closure could be obtained, e.g., by closing $\lhd$ by the reverse ordering $x \lhd y \leftrightarrow y \leq x$.

A closure of a framework $\mathcal{F}$ should be an adequate closed extension of $\mathcal{F}$. We can prove:

Theorem 3 (Closure). *Let* $\mathcal{F}(\Pi \Rightarrow \Delta) = \langle \Sigma, \mathbf{X} \rangle$ *be an adequate framework, and* $\mathcal{F}' = \langle \Sigma', (\mathbf{X} \setminus \mathbf{X}_C) \cup D_\Pi, \mathbf{T} \cup \mathbf{X}_C \rangle$ *be the result of a closure. Then* $\mathcal{F}'$ *is an adequate closed extension of* $\mathcal{F}$ *iff* $\mathcal{F}'$ *is* consistent *and* atomically complete.

The relation and function closures preserve consistency because D_Π are explicit definitions in the kernel and, by the constraint satisfaction condition, $\mathbf{X}_C$ become theorems. Thus *consistency* is preserved if sort closures preserve the consistency of $\mathbf{X} \setminus \mathbf{X}_C$. A sufficient condition to preserve consistency is that no cardinality restrictions are imposed on the open sorts, as is commonly the case (like, e.g., the open sort X in generic lists).

Concerning *atomic completeness*, let $\mathcal{K}$ be the extension of the closed kernel of $\mathcal{F}$ by the closure axioms D_Π. Atomic completeness may be not guaranteed for two reasons: (a) $\mathcal{K}$ is not atomically complete because D_Π are not adequate explicit definitions in the kernel, or (b) the atomic completeness of $\mathcal{K}$ is not sufficient to obtain the atomic completeness for the defined symbols, because stronger properties are required by the definition axioms. To avoid (a), D_Π must be adequate explicit definitions in the kernel. With quantifier free defining formulas, adequacy is guaranteed by Proposition 1. An example of (b) is the definition axiom $r(x) \leftrightarrow \exists y \bullet p(x, y)$, where p is a parameter; in this case, $\mathcal{K}$ should prove $\exists y \bullet p(\mathbf{x}, y)$ or $\neg\exists y \bullet p(\mathbf{x}, y)$ for every ground $\mathbf{x}$, i.e., atomic completeness of $\mathcal{K}$ does not suffice. However, in general it is reasonable to look for definition axioms that close the defined symbols whenever the open ones become closed, i.e., case (b) should be the exception. Thus, if we do not use quantified defining formulas, closures by internalisation are, in general, adequate.

Closure may also be performed incrementally, step by step. A partial closure is called a *specialisation*, because it does not close all the open symbols. Besides partial closure, we may have other kinds of specialisation. For example: adding constraints, using open symbols in the defining formulas, mapping open sorts into non-closed sorts, and so on. All these operations can be formalised by extension morphisms, but we will omit relevant details.

Example 13. The framework $\mathcal{LIST}_1$ of Example 11 is a specialisation of $\mathcal{LIST}$, obtained by the partial closure of X.

Framework Composition. Framework composition essentially coincides with framework union. The simplest case is disjoint union. However, it may happen that we want to preserve a common part, for example natural numbers. Here we consider the composition of two frameworks $\mathcal{F}_1$ and $\mathcal{F}_2$ that have a common subframework $\mathcal{G}$ containing their closed kernel, and have disjoint signatures for the symbols not in $\mathcal{G}$. In this case, composition preserving $\mathcal{G}$ can be defined as the operation $+_{\mathcal{G}}$ that builds the composite $\mathcal{F}_1 +_{\mathcal{G}} \mathcal{F}_2$ simply by making the union of signatures, open and closed symbols, and axioms. If $\mathcal{F}_1$ and $\mathcal{F}_2$ share symbols not in $\mathcal{G}$, then we rename such symbols, to make them different before performing the union.

$+_{\mathcal{G}}$ is syntactic composition. Its semantic counterpart is amalgamation. Two intended models I_1 of $\mathcal{F}_1$ and I_2 of $\mathcal{F}_2$ are amalgamable if they coincide over the common signature. Their amalgamation is the interpretation $\mathrm{I}_1 + \mathrm{I}_2$ that coincides with I_1 over the signature of $\mathcal{F}_1$ and with I_2 over the signature of $\mathcal{F}_2$ (the definition is consistent, because I_1 and I_2 coincide over the common signature). The intended models of $\mathcal{F}_1 +_{\mathcal{G}} \mathcal{F}_2$ are the amalgamations of the pairs of amalgamable intended models of $\mathcal{F}_1$ and $\mathcal{F}_2$.

This kind of composition has been formalised in ADT's using *pushouts* (see e.g., [13]), and the pushout approach also works for frameworks, and it allows us to generalise the operation $+_{\mathcal{G}}$. We do not consider the general case here for conciseness.

Example 14. Let $\mathcal{BOOL}$ be a framework defining booleans in the usual way. Lists of booleans with open ordering $\lhd : [Bool, Bool]$ can be defined starting from the disjoint union $\mathcal{LIST} + \mathcal{BOOL}$, as follows:

Framework $\mathcal{LIST}_2(\lhd \;\Rightarrow ListBool, nil, ., @, nocc)$
EXTENDS $\mathcal{LIST} + \mathcal{BOOL}$;
CLOSE: X BY $Bool$;
RENAME: $ListX$ BY $ListBool$;

We can compose lists of booleans $\mathcal{LIST}_2$ and lists of natural numbers $\mathcal{LIST}_1$ (see Example 11). To avoid duplicating the kernel of natural numbers, we perform the composition with common subframework $\mathcal{NAT}$:

$$\mathcal{LIST}_1 \;\; +_{\mathcal{NAT}} \;\; \mathcal{LIST}_2$$

To distinguish the non-common symbols, the composition renames them. Since we allow overloading, only sort and constant renaming may be needed. For example, we have $nil_1 : [\,] \rightarrow ListNat$,] $nil_2 : [\,] \rightarrow ListBool$, overloaded $. : [Nat, ListNat] \rightarrow ListNat$ and $. : [Nat, ListBool] \rightarrow ListBool$, and so on.

3.2 Specification Reuse

Specifications are used in two ways. Before composition or extension, they are used as a guide to search for possible compositional units that specify a desired context and set of operations. For example, if we need list sorting, we look for compositional units that contain the framework for lists with totally ordered elements, a specification S_{sort} of a *sort* operation, and a program $P : \ldots \Rightarrow S_{sort}, \ldots$. After composition or extension, the compositional units guide program composition, according to Theorem 2.

Reusability after composition or extension is enhanced by *specification reduction*, as considered in [15]. Indeed, after extension or composition, we have a richer framework, where new properties have been added. It may happen that a specification can be reduced to a new specification, that is, in the new context the new specification can replace the old one.

Informally, an “old” specification S reduces to a “new” specification S' if correctness with respect to the new S' entails correctness with respect to the old S. Formally, we give the following definition:

Definition 14 (Specification Reduction). *Let $\mathcal{F}$ be a framework, and S_ω, $S'_{\omega'}$ be two sets of specifications in $\mathcal{F}$. We say that S_ω* reduces to *$S'_{\omega'}$ iff $\omega \subseteq \omega'$ and $\mathcal{F} \vdash S'_{\omega'} \rightarrow S_\omega$.*

For two interface specifications $S_{\pi_1} \Rightarrow S_{\delta_1}$, $S_{\pi_2} \Rightarrow S_{\delta_2}$, we say that $S_{\pi_1} \Rightarrow S_{\delta_1}$ reduces *$S_{\pi_2} \Rightarrow S_{\delta_2}$ iff S_{π_2} reduces to S_{π_1} and S_{δ_1} reduces to S_{δ_2}.*

Reduction is transitive and reflexive. Its meaning is made clear by Theorem 4:

Theorem 4. *Let $\mathcal{F}$ be a framework, and $S_{\pi_1} \Rightarrow S_{\delta_1}$ and $S_{\pi_2} \Rightarrow S_{\delta_2}$ be two interface specifications. If $S_{\pi_1} \Rightarrow S_{\delta_1}$ reduces to $S_{\pi_2} \Rightarrow S_{\delta_2}$ in $\mathcal{F}$, then every program P that is correct with respect to $S_{\pi_2} \Rightarrow S_{\delta_2}$ is also correct with respect to $S_{\pi_1} \Rightarrow S_{\delta_1}$ (in $\mathcal{F}$).*

Example 15. Let **K** be a compositional unit with open framework $\mathcal{LIST}$, and let S_{lhd} be the strict specification $S_{lhd} : lhd(x, y) \leftrightarrow x \lhd y$. In the extension $\mathcal{LISTNAT}$ of $\mathcal{LIST}$, $S'_{lhd} : lhd(x, y) \leftrightarrow x \leq y$ reduces to S_{lhd} (we prove $S_{lhd} \rightarrow S'_{lhd}$ by the closure axiom $x \lhd y \leftrightarrow x \leq y$). Thus $S_{lhd} \Rightarrow S_{merge}$ (where S_{merge} is defined in Example 8) reduces to $S'_{lhd} \Rightarrow S_{merge}$, and we can use S'_{lhd} when deriving correct *merge* programs. For example, we could write a correct program $P_{merge'} : S'_{lhd} \Rightarrow S_{merge}$ which avoids comparisons with 0, since 0 is the minimum natural number; $P_{merge'}$ would correctly override a (possibly) inherited $P_{merge} : S_{lhd} \Rightarrow S_{merge}$.

In the reduction of conditional specifications [15], we can take into account the *call context*. This is shown in the following example.

Example 16. In the open framework $\mathcal{LIST}$ we can give the following specifications:

$$S'_{merge} : \ l = x.nil \wedge ord(m) \rightarrow (merge(l, m, o) \leftrightarrow ord(o) \wedge perm(x.m, o));$$
$$S'_{split} : \ split(x.l, m, n) \leftrightarrow m = x.nil \wedge n = l.$$

S_{split} of Example 8 reduces to S'_{split} ($S'_{split} \rightarrow S_{split}$ can be proved in $\mathcal{LIST}$). S_{merge} of Example 8 reduces to S'_{merge} in a *call context* where the input condition $l = x.nil \wedge ord(m)$ of S'_{merge} holds for the lists l and m to be merged. Indeed, in such a context, S'_{merge} corresponds to $merge(x.nil, m, o) \leftrightarrow ord(o) \wedge perm(x.m, o)$, S_{merge} to $merge(x.nil, m, o) \leftrightarrow ord(o) \wedge perm(x.nil|m, o)$, and they are equivalent. We will say that S_{merge} *contextually reduces* to S'_{merge}.

Contextual reduction implies *contextual reuse*, that is, S_{merge} correctly reduces to S'_{merge} only when the input condition of S'_{merge} is true. As a consequence, we cannot replace S_{merge} by S'_{merge} in isolation, but we have to consider the call context. In contrast, we can replace S_{split} by S'_{split} in isolation, because the corresponding reduction is not contextual.

As we will see in Example 17, S'_{merge} and S'_{split} are tailored to the insertion sort algorithm. In a similar way, we can specialise S_{merge} and S_{split} to obtain specifications tailored to different sorting algorithms, like merge sort, quick sort, and so on.

In general, it is useful to list proven reduction theorems in the specification section of a compositional unit. Such a list would allow us to automatically search for families of program compositions, giving rise to families of implementations. It is for this reason that we have put $RD_1, ...RD_k$ in Fig. 1.

3.3 Program Reuse

Like specifications, programs in compositional units can be used before and after unit composition.

We use programs before composition when we look for existing compositional units containing *specific algorithms*. Otherwise, reuse is after unit composition, when we use the inherited programs to solve the problem in question. The operation that allows us to reuse the inherited programs is program composition. It is strongly guided by specifications. Specification reduction is important for program reusability, since it allows us to use the richer knowledge obtained after framework composition and extension to solve the puzzle of composing the inherited open programs into a correct solution of the problem at hand.

Example 17. Let **K** be a compositional unit with framework $\mathcal{LIST}$, and let us assume that it already contains the correct program $P_{insert} : S_{lhd} \Rightarrow S_{insert}$, where P_{insert} implements the usual algorithm for inserting an element into its

correct position in an ordered list, S_{lhd} is the specification shown in Example 15, and S_{insert} is:

$$S_{insert}: \quad ord(l) \rightarrow (insert(x,l,m) \leftrightarrow (ord(m) \wedge perm(x.l,m))).$$

We show how reductions of Example 16 can be used to solve the puzzle of obtaining a correct sorting program $Q_{sort} : S_{lhd} \Rightarrow S_{sort}$. If we compose P_{insert} with the correct one-clause programs

$$P_{split} : \Rightarrow S'_{split} \qquad split(x.l, x.nil, l) \leftarrow$$
$$P_{link} : S_{insert} \Rightarrow S'_{merge} \qquad merge(x.nil, l, o) \leftarrow insert(x, l, o)$$

we get a correct program $Q_{aux} : S_{lhd} \Rightarrow S'_{split}, S'_{merge}$. By the specification reductions of Example 16, the interface specification $S'_{split}, S'_{merge} \Rightarrow S_{sort}$ *contextually* reduces to $S_{split}, S_{merge} \Rightarrow S_{sort}$. Thus, the program $P_{sort} : S_{split}, S_{merge} \Rightarrow S_{sort}$ of Example 9 is also correct with respect to $S'_{split}, S'_{merge} \Rightarrow S_{sort}$, because the input condition of S'_{merge} is satisfied in the call context of $merge$, as required. By composing P_{sort} and Q_{aux}, we get a correct $Q_{sort} : S_{lhd} \Rightarrow S_{sort}$.

Q_{sort} can be closed in the instances that close lhd. For example, S_{lhd} reduces to S'_{lhd} in a compositional unit with framework $\mathcal{LISTNAT}$, as shown in Example 15. Suppose that our compositional unit already contains a correct program $P_{leq} : \Rightarrow S_{leq}$. If we compose it with the correct one-clause program

$$P_{lhd} : S_{leq} \Rightarrow S'_{lhd} \qquad lhd(x,y) \leftarrow leq(x,y)$$

we get a closed correct program $Q_{lhd} :\Rightarrow S_{lhd}$. By specification reduction we get that $Q_{lhd} : \Rightarrow S'_{lhd}$ is also correct. Then the closed program $Q_{lhd} \cup Q_{sort} : \Rightarrow S_{sort}$ is correct in $\mathcal{LISTNAT}$.

3.4 Related Work

At the framework level, our approach to modularity and reuse is in the tradition of algebraic ADT's [2,13,47]. We can apply the techniques developed there, based on theory morphisms. Our *specification frameworks* should not be confused with the *specification frames* introduced in [25]. The latter, like institutions [19], are general frames for the composition and reuse of formal theories. With respect to modularity and compositionality, our frameworks with open symbols and defined symbols are similar, for example, to modules with import and export interfaces, as introduced in [14].

In [25], a distinction between parameterised specifications and parameterised data types is introduced, following [42]. In [42], programs and specifications are considered as different entities, involved in different phases and different methodological aspects of program development, and a distinction between parameterised specifications and specifications of parameterised programs is introduced. In this, [42] is very close to our general view, but our approach is different. Our three-level architecture of compositional units is closer to Larch [20]. Like Larch, our specifications state precisely how open programs interact,

and allow us to compose them correctly. However, unlike Larch, we have a further specification level, which is intermediate between the framework level and the interface specification level. This yields a further level of correct reuse, through the specification reduction theorems.

With regard to modularity in logic programming, there are approaches based on ideas similar to our J-models (see [7]), while the approach proposed in [39] relates to specification frames [25]. However, all these approaches do not distinguish between specifications and programs. A distinction between programs and specifications is made in [40], where modular Prolog programs (as proposed in [44]) are derived from first-order specifications (based on Extended ML [43]). However, in [40], the role of specifications is different from ours, and there is no counterpart of specification frameworks.

Finally, in the area of object-oriented analysis and design, component-based development methods [12,3] have emerged, where components and reuse are two of the main aspects of the software development process. In this area, a software component is a unit of composition with contractually specified interfaces and context dependencies only [46]. Our compositional units broadly fit this characterisation, considering interface specifications $S_\pi \Rightarrow S_\delta$ as interfaces, and specifications and their reducibility relation in the context of the framework as context dependencies.

4 Conclusion

In this paper we have essentially collected our previous work on program specification and synthesis, and we have organised it by introducing compositional units, which are a more complete and refined version of correct schemas [16]. Then we have illustrated the basic operations for extending and correctly reusing (composing) compositional units.

A compositional unit is a unit of reuse that contains both a formalisation of the problem domain, at the framework level, and a collection of open programs, correct with respect to their specifications, at the specification and program levels. The framework level specifies, by the constraint axioms, when and how a compositional unit can be correctly reused. The specification and program levels support program reuse and development. The examples of Section 3 have been mainly devoted to illustrating the role of specifications in the correct reuse of compositional units for program development. In particular, specifications are a guide for program composition, and specification reduction allows us to deal with the problem of adapting the inherited open programs to the specific context of reuse.

In this paper, we have not considered program synthesis, because we concentrated on specifications and their role in the reuse of compositional units and correct open programs. However, there is a strong relationship to logic program synthesis [11], and indeed our research started in this area. An interesting fact is the possibility of using logic program synthesis as a way for expanding frameworks in an adequate way [27].

The distinct levels for specifications and programs distinguish our approach. At these levels, we have integrated our research on steadfast open programs and specifications. As we have shown, specifications and steadfast programs yield a further level of reuse, through specification reduction and program composition. We believe that this is an important feature of our approach, especially in the context of so-called software components [46]. Our future work will be devoted to the study of the applicability of our approach to the development of correct component-based software.

On the one hand, we want to develop the approach further, based on logic programs, along the following two lines: (a) We will extend our approach to other kinds of logic programs. For constraint logic programs and those normal programs that have one intended model, the extension of our results is almost immediate. (b) We will study methods for deriving steadfast programs from their interface specifications, based on our compositional units and on the results of [34] and the ideas exposed in [16]. To this end, tools would be necessary for developing an interactive environment where we can define and compose specification frameworks, specifications and programs, and use a proof assistant for developing the necessary proofs. We are looking at logical frameworks like Isabelle [23] as possible candidates.

On the other hand, we want to consider the extension of our approach to different programming paradigms. This can be done in two ways. The first choice is to define, on top of specification frameworks, different specification formalisms, oriented to different program languages. Such formalisms would provide different interface specification languages, in a way similar to Larch [20]. The second choice is to use our compositional units as meta-level declarative specifications of systems implemented in possibly imperative programming languages.

So far, we have considered only the second choice. We began a study of object-oriented systems, with the aim of testing the versatility of our model and, hopefully, of obtaining a formalisation of object-oriented compositional units that could be used as software components. In [32], we introduced a static model of object-oriented systems, suitable for formalising states and queries. Our static approach shares similarities with [6,36], and allows us to formalise UML class and object diagrams [41], queries and OCL constraints [9]. The introduction of time in our object-oriented systems is work in progress.

Our final goal is to obtain a methodology for the specification and the development of correct component-based software, where programs are developed together with the formal proof of their correctness. This methodology should allow the development of correct compositional units to be used as software components, that is, units of composition that can be deployed independently and are subject to composition by third parties [46].

Acknowledgements

We are very grateful to the referees for their valuable suggestions and comments. This paper has been radically improved as a result of their efforts.

References

1. J.R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors. *Algebraic Foundations of Systems Specifications.* Springer, 1999.
3. C. Atkinson *et al. Component-based Product Line Engineering with UML.* Addison-Wesley, 2001.
4. J. Barwise, editor. *Handbook of Mathematical Logic.* North Holland, 1977.
5. A. Bertoni, G. Mauri, and P. Miglioli. On the power of model theory in specifying abstract data types and in capturing their recursiveness. *Fundamenta Informaticae*, VI(2):127–170, 1983.
6. R.H. Bourdeau and B. H.C. Cheng. A formal semantics for object model diagrams. *IEEE Trans. Soft. Eng.*, 21(10):799–821, 1995.
7. M. Bugliesi, E. Lamma, and P. Mello. Modularity in logic programming. *J. Logic Programming*, 19,20:443–502, 1994. Special issue: Ten years of logic programming.
8. K.L. Clark. *Predicate Logic as a Computational Formalism.* Report 79/59, Imperial College of Science and Technology, University of London, 1979.
9. S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The Amsterdam manifesto on OCL. In T. Clark and J. Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language, LNCS 2263*, pages 115–149. Springer, 2002.
10. Y. Deville. *Logic Programming. Systematic Program Development.* Addison-Wesley, 1990.
11. Y. Deville and K.-K. Lau. Logic program synthesis. *J. Logic Programming*, 19,20:321–350, 1994. Special Issue: Ten Years of Logic Programming.
12. D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, 1999.
13. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, 1987.
14. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2.* Springer-Verlag, 1989.
15. P. Flener, K.-K. Lau, and M. Ornaghi. On correct program schemas. In N. Fuchs, editor, *Proc. LOPSTR 97, Lecture Notes in Computer Science 1463*, pages 124–143. Springer-Verlag, 1998.
16. P. Flener, K.-K. Lau, M. Ornaghi, and J. Richardson. An abstract formalisation of correct schemas for program synthesis. *Journal of Symbolic Computation*, 30(1):93–127, July 2000.
17. J.H. Gallier. *Logic for Computer Science: Foundations for Automatic Theorem Proving.* Harper and Row, 1986.
18. C. Ghezzi, M. Jazayeri, and D.Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, second edition, 2003.
19. J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.
20. J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.
21. W. Hodges. Logical Features of Horn Clauses. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 1*:449–503, Oxford University Press, 1993.
22. C.J. Hogger. *Derivation of Logic Programs. J. ACM*, 28(2):372–392, 1981.

23. Isabelle: www.cl.cam.ac.uk/Research/HVG/Isabelle
24. C.B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, second edition, 1990.
25. R.M. Jimenez, F. Orejas, and H. Ehrig. Compositionality and Compatibility of Parametrization and Parameter Passing in Specification Languages. *Math. Struct. in Computer Science*, 5:283–314, 1995.
26. C. Kreitz, K.-K. Lau, and M. Ornaghi. Formal reasoning about modules, reuse and their correctness. In D.M. Gabbay and H.J. Ohlbach, editors, *Proc. Int. Conf. on Formal and Applied Practical Reasoning, Lecture Notes in Artificial Intelligence 1085*, pages 384–399. Springer-Verlag, 1996.
27. K.-K. Lau and M. Ornaghi. On specification frameworks and deductive synthesis of logic programs. In L. Fribourg and F. Turini, editors, *Proc. LOPSTR 94 and META 94, Lecture Notes in Computer Science 883*, pages 104–121. Springer-Verlag, 1994.
28. K.-K. Lau and M. Ornaghi. Forms of logic specifications: A preliminary study. In J. Gallagher, editor, *Proc. LOPSTR 96, Lecture Notes in Computer Science 1207*, pages 295–312. Springer-Verlag, 1997.
29. K.-K. Lau and M. Ornaghi. The relationship between logic programs and specifications — the subset example revisited. *J. Logic Programming*, 30(3):239–257, March 1997.
30. K.-K. Lau and M. Ornaghi. OOD frameworks in component-based software development in computational logic. In P. Flener, editor, *Proc. LOPSTR 98, Lecture Notes in Computer Science 1559*, pages 101–123. Springer-Verlag, 1999.
31. K.-K. Lau and M. Ornaghi. Isoinitial semantics for logic programs. In J.W. Lloyd *et al*, editor, *Proc. 1st Int. Conf. on Computational Logic, Lecture Notes in Artificial Intelligence 1861*, pages 223–238. Springer-Verlag, 2000.
32. K.-K. Lau and M. Ornaghi. Correct object-oriented systems in computational logic. In A. Pettorossi, editor, *Proc. LOPSTR 01, Lecture Notes in Computer Science 2372*, pages 168–190. Springer-Verlag, 2002.
33. K.-K. Lau and M. Ornaghi. Specifying object-oriented systems in computational logic. In M. Bruynooghe, editor, *Pre-Proceedings of LOPSTR 03*, pages 49–64, 2003. Report CW 365, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium.
34. K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *J. Logic Programming*, 38(3):259–294, March 1999.
35. J.W. Lloyd. *Foundations of Logic Programming.* 2nd edn., Springer-Verlag, 1987.
36. T. Maibaum. Conservative extensions, interpretations between theories and all that. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT '97: Theory and Practice of Software Developement*, pages 40–67. Springer-Verlag, 1997. LNCS 1214.
37. Yu.V. Matijacevic. Recursively enumerable sets are Diophantine. *Dokl. Akad. Nauk SSSR*, 191:279–282, 1970.
38. P. Miglioli, U.Moscato, and M. Ornaghi. Abstract parametric classes and abstract data types defined by classical and constructive logical methods. *J. Symbolic Computation*, 18:41–81, 1994.
39. P. Miglioli, U.Moscato, and M. Ornaghi. An algebraic framework for the definition of compositional semantics of normal logic programs. *The Journal of Logic Programming*, 40:89–123, 1999.
40. M.G. Read and E.A. Kazmierczak. Formal program development in modular Prolog: A case study. In T.P. Clement and K.-K. Lau, editors, *Proc. LOPSTR 91*, pages 69–93. Springer-Verlag, 1992.

41. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
42. D. Sannella, S. Sokolowski, and A. Tarlecki. Toward formal development of programs from algebraic specifications: parametrisation revisited. *Acta Informatica*, 29(8):689–736, 1992.
43. D. Sannella and A. Tarlecki. Extended ML: past, present and future. In *Proc. 7th workshop on specification of abstract data types*, LNCS 534, pages 297–322. Springer-Verlag, 1991.
44. D. Sannella and L.A. Wallen. A calculus for the construction of mdular prolog programs. In *IEEE 4th Symposium on Logic Programming*, IEEE, 1987.
45. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
46. C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.
47. M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier, 1990.