Constructive Specifications for Compositional Units

Kung-Kiu Lau[1], Alberto Momigliano[2], and Mario Ornaghi[2]

[1] School of Computer Science, The 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
{momiglia, ornaghi}@dsi.unimi.it

Abstract. In previous work, we have introduced a model-theoretic semantics for compositional units, i.e. reusable units that can be used for compositional program development. Such units contain open (logic) programs and our model-theoretic semantics characterizes their correctness and the correctness of their composition. However, for real-world software development, compositional units should be inter-operable, i.e. they should accept programs in different languages. To cater for this, our model-theoretic semantics needs to be used in conjunction with suitable semantics for behaviours and interfaces. In this paper we describe one possible approach based on constructive specifications.

1 Introduction

A reusable program unit, or a compositional unit, contains code that can be composed with other units, to yield a desired behavior. To be widely reusable, a unit should be open, i.e., not completely defined. The undefined parts become defined as a result of composition. Examples of open units are generic modules with import/export sections in modular languages, generic ADT's [7] and frameworks in OO design [6]. In the context of Logic Programming, open programs have been proposed as open units, due to their compositional properties [2]. Units should be only specified by an interface specification and context dependencies [17], and we need rules for composing and decomposing interface specifications and to compare them, that is, to state whether interface specification S_1 meets or entails interface specification S_2. These properties allow us to combine top-down and bottom-up development.

In all this, the problem context plays an underpinning role. Indeed, context dependencies and specifications have their proper meaning only in the problem context and the latter plays a fundamental aspect in comparing specifications and to prove program properties. We have considered the role of the problem context w.r.t. compositionality in our previous work [10]. We have introduced a model-theoretic semantics for the correctness of open programs and we have studied how to combine programs, contexts and specifications into compositional units of interface specifications and code.

Compositional units should be *inter-operable*, i.e. (pure) logic programs in such units should be able to inter-operate with programs in different languages. For this,

S. Etalle(Ed.): LOPSTR 2004, LNCS 3573, pp. 198–214, 2005.

a model-theoretic semantics does not suffice. Inter-operation involves properties that are not explicit in the model-theoretic specifications, such as, for example, input and output modes in a logic program. Properties of this kind should be *explicit* in the interface specification of a unit. In this paper we propose a notion of constructive interface specifications, which can be consistently used with the model-theoretic one – classical model theory still remaining the basic semantics. This allows us to make explicit modes and other computational behaviours that are relevant for the correct composition of programs, but are not apparent in a purely model-theoretic approach. The advantage of a formalisation of these aspects is that we get both a precise *semantics* and a *compositional calculus*. The latter allows us to reason about interface specifications and to derive *correct* unit compositions, possibly using automatic theorem provers.

2 Program Units and Their Composition

In this section we give an informal overview of our approach to program units and their composition. We define a *program unit* as a triple $U = \langle \Sigma, \mathcal{C}, Prog \rangle$, where Σ is a many-sorted signature, $\mathcal{C}$ is a class of Σ-models and *Prog* is a collection of programs. We call U a *program unit* (PU) to avoid confusion with compositional units (CU) introduced in our previous work [10]. The class $\mathcal{C}$ is called the *context* of U. It may be defined formally (as, e.g., in CU's) or informally (as we will do in the examples), with the purpose of *specifying the meaning of the procedures used in the unit, in terms of the problem domain*. We distinguish between *imported, exported and hidden* procedures. The imported procedures are assumed to be supplied by external units. For each exported procedure p, *Prog* contains a *program* implementing p, hidden in U, and a set of (public) *interfaces* $\mathtt{p} : \mathrm{I}_1$, ..., $\mathtt{p} : \mathrm{I}_k$, that refer to when and how p can be correctly composed with procedures imported from external units.

Example 1. The *context section* of the following program unit *LG* specifies the problem domain of labeled graphs. The ADT $List(V)$ (lists with elements from V) is included to define predicates on paths. For example, the definition of $x \xrightarrow{a^n} y$ (*x is linked to y by n consecutive arcs with label a*) uses $at : [V, Nat, List(V)]$, defined in $List(V)$ ($at(v,i,l)$ means that v occurs at position i in list l). The meaning of the imported and exported procedures is defined by the specifications $S_{\mathtt{arc}}$, $S_{\mathtt{conn}}$, $S_{\mathtt{path}}$, using the context signature and the concrete data types of the implementation language, such as `int` in $S_{\mathtt{conn}}$; $repr_{Nat}(i,n)$ means that the integer i represents the natural number n. We have used a many-sorted Prolog, to shrink the gap between the context and the implementation level for the sorts V, L, $List(V)$.

Program unit *LG*;
Context. SIGNATURE: V, L : sort; $\_ \xrightarrow{\_} \_ : [V, L, V]$; INCLUDES: $List(V)$;
INTERPRETATIONS: For each graph, $\mathcal{C}$ contains an interpretation where V and L are the sets of nodes and labels, and $x \xrightarrow{a} y$ holds iff there is an arc from x to y with label a.
$x \xrightarrow{a^n} y \leftrightarrow \exists p\,.\; at(x,0,p) \wedge at(y,n,p) \wedge \forall u,v,i.\; at(u,i,p) \wedge at(v,s(i),p) \rightarrow (u \xrightarrow{a} v)$
IMPORT: $S_{\mathtt{arc}:[V,L,V]} : \mathtt{arc}(x,a,x') \leftrightarrow (x \xrightarrow{a} x')$.
EXPORT: $S_{\mathtt{path}:[List(V)]} : \mathtt{path}(p) \leftrightarrow \forall x,x',i.\; at(x,i,p) \wedge at(x',s(i),p) \rightarrow \exists a\,.\; x \xrightarrow{a} x'$.
$S_{\mathtt{conn}:[V,L,V,\mathtt{int}]} : repr_{Nat}(i,n) \rightarrow (\mathtt{conn}(x,a,y,i) \leftrightarrow (x \xrightarrow{a^n} y))$

Programs.

$\texttt{path} : \text{P1}\ (\forall x,y.\ (\exists a.\ ?\texttt{arc}(x^+,a^-,y^+)) \vee T(\neg\exists a.\ \texttt{arc}(x,a,y)))$
$\qquad\qquad \rightarrow (\forall p.\ ?\texttt{path}(p^+) \vee T\neg\texttt{path}(p));$

$\texttt{path} : \text{P2}\ T(\exists n.\ \forall p.\ \texttt{path}(p) \rightarrow \mathit{length}(p) \leq n) \wedge\ (\exists s.\ \forall !x,a,y \in s.\ ?\texttt{arc}(x^-,a^-,y^-))$
$\qquad\qquad \rightarrow\ \exists s'.\ \forall !p \in s'.\ ?\texttt{path}(p^-);$

IMPL : $\texttt{path}([]).$
$\qquad \texttt{path}([\_]).$
$\qquad \texttt{path}([\texttt{X},\texttt{Y}|\texttt{R}]) : -\texttt{arc}(\texttt{X},\_,\texttt{Y}),\texttt{path}([\texttt{Y}|\texttt{R}]).$

$\texttt{conn} : \text{C1}\ (\forall x,a.\ \exists y.\ !\texttt{arc}(x^+,a^+,y^-))$
$\qquad\qquad \rightarrow (\forall x,a,i.\ T(i \geq 0) \rightarrow \exists y.\ !\texttt{conn}(x^+,a^+,y^-,i^+))$

IMPL : $\texttt{conn}(\texttt{X},\_,\texttt{X},0).$
$\qquad \texttt{conn}(\texttt{X},\texttt{A},\texttt{Y},\texttt{I}) : -\texttt{I} > 0,\texttt{J is I} - 1,\texttt{arc}(\texttt{X},\texttt{A},\texttt{Z}),\texttt{conn}(\texttt{Z},\texttt{A},\texttt{Y},\texttt{J}).$

We call $S_{\texttt{arc}}$, $S_{\texttt{conn}}$, $S_{\texttt{path}}$ *model-theoretic specifications*, to distinguish them from *constructive specifications* of the *interfaces* of the procedures implemented in the *program* section. By $\texttt{conn} : \text{C1}$ we mean that conn *realizes the interface* specified by C1. Similarly for $\texttt{path} : \text{P1}$ and $\texttt{path} : \text{P2}$.

We have used moded call statements, such as $?path(p^-)$, where '$-$' denotes the output mode and '?' the query mode. They will be explained later in the paragraph introducing behaviours. Interfaces are interpreted according to the *constructive semantics* that we are now going to informally introduce. To reuse *LG*, an external unit *Q* has to fix the interpretation of the open symbols *V*, *L* and $\_ \xrightarrow{\_} \_$, to represent the set of the nodes, labels of a specific graph and to implement the procedure arc. We want to guarantee that the program implementing arc correctly composes with the programs imported from *LG*. To this end, in [10], we have considered a notion of model theoretic interfaces and correctness. For example, $LG.\texttt{path} : S_{\texttt{arc}} \rightarrow S_{\texttt{path}}$ is a model theoretic interface for *LG*.path and its (model theoretic) correctness means that $S_{\texttt{path}}$ is true in the minimum Herbrand model (MHM) of *LG*.path, whenever *LG*.path is composed with a program *Q*.arc with a MHM satisfying $S_{\texttt{arc}}$. This entails that from $LG.\texttt{path} : S_{\texttt{arc}} \rightarrow S_{\texttt{path}}$ and $Q.\texttt{arc} : \mathit{true} \rightarrow S_{\texttt{arc}}$ we can infer $Q.\texttt{arc} \circ_{\texttt{arc}} LG.\texttt{path} : \mathit{true} \rightarrow S_{path}$. By $\circ_{\texttt{arc}}$ we denote the *import composition* with respect to arc, performed at the program level when the unit (*LG*) importing arc includes or is included in a unit (*Q*) exporting arc. For logic programs, $\circ_{\texttt{arc}}$ is the union, although renaming may be needed.

As the above discussion shows, model theoretic interfaces have a simple declarative semantics and simple composition rules. However, they do not take into account computational aspects that are relevant for correct program reuse. For example, the interface $LG.\texttt{path} : S_{\texttt{arc}} \rightarrow S_{\texttt{path}}$ cannot distinguish the following different uses of *P*.path:

a) If $Q.\texttt{arc}$ *decides* $\exists a.\ arc(x,a,y)$ for every pair x,y of nodes of a graph g, we can use $Q.\texttt{arc} \circ_{\texttt{arc}} LG.\texttt{path}$ to decide $\texttt{path}(p)$, for every ground sequence p of nodes.
b) If g has no infinite paths and via $Q_{\texttt{arc}}$ we can obtain a finite *enumeration* of all the arcs of g, then $Q.\texttt{arc} \circ_{\texttt{arc}} LG.\texttt{path}$ can be used to enumerate all the paths in g.

Interfaces based on *constructive specifications* make this kind of computational aspects explicit, while preserving the declarativeness of model-theoretic specifications. For example, interfaces P1and P2 correspond to uses a) and b). Here we give an informal account of constructive specifications and their role in procedure composition, by explaining the interfaces of the procedures of *LG*.

Interfaces and specifications. The interface P1 of $LG.\texttt{path}$ is an *implication* $I_{\mathrm{P1}} \rightarrow E_{\mathrm{P1}}$, where I_{P1} is a specification for the imported procedure arc and E_{P1} is a specification for the exported procedure path. The meaning is that $LG.\texttt{path}$ realizes the behaviour specified by E_{P1}, whenever LG is composed with a unit Q containing a program $Q.\texttt{arc}$ realizing I_{P1}. More precisely, after the composition, we get a richer program unit $LG+Q$, where $Q.\texttt{arc} \circ_{\texttt{arc}} LG.\texttt{path}$ realizes E_{P1}. Thus, *interfaces model unit composition at the program level*. I_{P1} is a *universal specification*. It is *realized* iff the main sub-formula $(\exists a.\ ?\texttt{arc}(x^+,a^-,y^+)) \vee T(\neg\exists a.\ \texttt{arc}(x,a,y))$ is realized for all the ground substitutions $x = t_1, y = t_2$ of the universally quantified variables. The main sub-formula is an *existential specification*. An existential specification E is built by means of $\wedge$, $\vee$, $\exists$, starting from *moded call statements* (such as $?\texttt{arc}(x^+,a^-,y^+)$) and *T-formulas* (such as $T(\neg\exists a.\ \texttt{arc}(x,a,y))$). A ground instance $E\sigma$ of E is *realized by a program unit* U iff it is realized by the *behaviour* of U, *observed* using the moded call statements of E.

Behaviours. To explain realizability, we have to consider moded call statements and behaviours. In the moded call statement $?\texttt{arc}(x^+,a^-,y^+)$ of P1, parameters x, y have *input mode* '+', parameter a has *output mode* '−'and arc has *query mode* '?'. Input and output modes have the usual meaning [4]: a call $\texttt{arc}(t_1,A,t_2)$ works properly if t_1,t_2 are ground and the (possible) variables of A will be instantiated by the answer substitution. In particular, the query $:-\texttt{arc}(t_1,A,t_2)$ is admitted, with A a variable. The query mode ?arc signifies that the set $A = a_1$, ..., $A = a_k$, ... of *all the answers* can be obtained by backtracking. We represent the corresponding *behaviour* $\mathcal{B}_{\texttt{arc}}$ by the *b-formulas* (behaviour formulas) $\texttt{arc}(t_1,a_1,t_2)$, ..., $\texttt{arc}(t_1,a_k,t_2)$, ...; if after n answers backtracking fails, we put the b-formula $T(\forall a.\ \neg a = a_1 \wedge \cdots \neg a = a_n \rightarrow \neg\texttt{arc}(t_1,a,t_2))$ in $\mathcal{B}_{\texttt{arc}}$ (if $n = 0$, $\mathcal{B}_{\texttt{arc}} = \{T(\forall a.\ \neg\texttt{arc}(t_1,a,t_2))\}$). The use of T for representing failure will be explained very soon.

Realizability of existential specifications. The formal definition of $\mathcal{B} \mid\!\models E$, i.e. when behaviour $\mathcal{B}$ realizes specification E is given in Def. 1. Note that realizability is *constructive*: a realization of $A \vee B$ allows us to decide which disjunct holds and a realization of $\exists x.\ A(x)$ to obtain an answer $x = t$. For example, $(\exists a.\ \texttt{arc}(t_1,a,t_2)) \vee T(\neg\exists a.\ \texttt{arc}(t_1,a,t_2))$ is realized by $\mathcal{B}_{\texttt{arc}}$, as follows. If $\texttt{arc}(t_1,a_1,t_2) \in \mathcal{B}_{\texttt{arc}}$, then the disjunct $\exists a.\ \texttt{arc}(t_1,a,t_2)$ is realized, with answer $a = a_1$. If no answers exist, then $T(\forall a.\ \neg\texttt{arc}(t_1,a,t_2)) \in \mathcal{B}_{\texttt{arc}}$. Therefore the disjunct $T(\neg\exists a.\ \texttt{arc}(t_1,a,t_2))$ is realized by the truth of $\neg\exists a.\ \texttt{arc}(t_1,a,t_2)$.

T-formulas. We have used T in a disjunction of the form $H \vee T(F)$, to deal with finite failure: if the attempt of realizing H fails, we conclude that F is true, but we do not require any further information. In this sense, T corresponds to the *classical truth operator* of [15]. T-formulas can be also used as pre-conditions, not requiring any computation. For example, $T(\exists n.\ \forall p.\ path(p) \rightarrow length(p) \leq n)$ in P2 requires that all the paths of the current graph have length limited by a fixed n, but it does not require to compute n. Another use of T is illustrated by the bounded quantification $\forall!x \in s.\ A(x)$

used in P2, which abbreviates $(\forall x.\, A(x) \vee T(\neg A(x))) \wedge (\forall x.\, T(member(x,s)) \leftrightarrow A(x))$ [1]. Let s be $[t_1,\ldots,t_n]$. Then a behaviour $\mathcal{B}$ realizes $\forall!x \in s.\, A(x)$ iff it realizes $A(t_1)$, …, $A(t_n)$ and $T(x \neq t_1 \wedge \cdots \wedge x \neq t_n \rightarrow \neg A(x))$. Similarly the notation $\exists!x.\, A(x)$ abbreviates $\exists x.\, A(x) \wedge T(\forall x,y.\, A(x) \wedge A(y) \rightarrow x = y)$.

Different kinds of interfaces. According to the informal explanations above, one can see that P1 corresponds to use a). Its import part I_{P1} and export part E_{P1} specify *decision behaviours*. P2 corresponds to use b). Its import and export parts specify *enumeration behaviours*. Import and export parts of C1 specify *selection behaviours*: if for every node x and label a, $Q_{\mathtt{arc}}$ selects a y such that $x \xrightarrow{a} y$, then for every node x, label a and integer $i \geq 0$, $LG.\mathtt{conn}$ selects an y such that $x \xrightarrow{a^i} y$. Here, $!\mathtt{arc}$ is the *selection mode*. It means that by means of a call to `arc` we get an answer (selected by the program), but backtracking is forbidden or it may not work properly. *Functional behaviours* are a special case of selection behaviours, where the selected output is also unique. Decision, enumeration, selection and functional behaviours are the most common. Of course, a complex specification may mix them.

Composition calculus. Finally, we have a *calculus* to compose procedures according to their interfaces, while preserving realizability. The composition calculus is explained in Sect. 5.1. We conclude our informal overview with an example, which shows the application of the calculus and introduces *bridge proofs*, namely proofs that allow us to link interfaces in case they do not match exactly. When we include a unit, the context is included with possible renaming and *closures*, i.e. instantiations of open symbols, as illustrated in the next example. Then procedures may be selectively included. We may *hide* them (when appropriate) via the operator #. We may also use local (hidden) bridge programs generated when building bridge proofs.

Example 2. The unit IT implements an iterator of a generic binary operation
Program Unit IT. **Context.** Contains Nat, a generic domain D and a binary operation $\cdot : [D,D] \rightarrow D$ with left unit $\mathbf{u} : D$. Defines $x^0 = \mathbf{u}$, $x^{(n+1)} = x^n \cdot x$.
IMPORT: $S_{\mathtt{unit}:[D]} : \mathtt{unit}(x) \leftrightarrow x = \mathbf{u}$; $S_{\mathtt{op}:[D,D,D]} : \mathtt{op}(x,a,y) \leftrightarrow y = x \cdot a$;
EXPORT: $S_{\mathtt{iter}:[D,\mathtt{int},D]} : repr_{Nat}(i,n) \rightarrow (\mathtt{iter}(x,i,z) \leftrightarrow z = x^n)$.
Programs.

```
iter : IT    (∃x. !unit(x⁻)) ∧ (∀x,a. ∃y. !op(x⁺,a⁺,y⁻))
                 → (∀x,i. T(i ≥ 0) → ∃!z. iter(x⁺,i⁺,z⁻));
       IMPL: iter(X,0,U) : −unit(U).
             iter(X,I,U) : −I > 0, H is I − 1, iter(X,H,V), op(V,X,U).
```

In the unit LG, $LG.\mathtt{conn}$ is tail recursive and $\mathtt{conn}(x,a,y,n)$ is defined as $x \xrightarrow{a^n} y$. If we include IT into LG and we take the sets V of nodes and L of labels as D, we may interpret each pair $(x, x \cdot a)$ as an arc $x \xrightarrow{a} x \cdot a$ and we can prove $x \xrightarrow{a^n} y \leftrightarrow y = x \cdot a^n$. Thus we can (re)use $LG.\mathtt{conn}$ to compute `iter` in a tail-recursive way, as follows.

Program Unit $LGIT$ EXTENDS LG, INCLUDES IT.
CLOSE: $V := D$; $L := D$; $x \xrightarrow{a} y \leftrightarrow y = x \cdot a$;

[1] With quantifiers such as $\forall!x,a,y \in s.$ in P2, s is a list of triples.

THM: $i \geq 0 \rightarrow (\texttt{iter}(x,i,z) \leftrightarrow \texttt{conn}(\mathbf{u},x,z,i))$; $\texttt{op}(x,a,y) \leftrightarrow \texttt{arc}(x,a,y)$.

Programs.

$\texttt{iter} : \#\text{IT}$ $\quad (\exists x.\ !\#\texttt{unit}(x^-)) \wedge (\forall x,a,i.\ T(i \geq 0) \rightarrow \exists y.\ !\#\texttt{conn}(x^+,a^+,y^-,i^+))$
$\rightarrow (\forall x,i.\ T(i \geq 0) \rightarrow \exists!z.\ !\texttt{iter}(x^+,i^+,z^-))$:

IMPL: $\texttt{iter}(\texttt{X},\texttt{I},\texttt{Y}) : -\#\texttt{unit}(\texttt{U}),\#\texttt{conn}(\texttt{U},\texttt{X},\texttt{Y},\texttt{I})$.

$\texttt{iter} : \text{IT}$ $\quad (\exists x.\ !\texttt{unit}(x^-)) \wedge (\forall x,a.\ \exists y.\ !\texttt{op}(x^+,a^+,y^-))$
$\rightarrow (\forall x,i.\ T(i \geq 0) \rightarrow \exists!z.\ !\texttt{iter}(x^+,i^+,z^-))$

IMPL: $(\texttt{rn}[\texttt{unit}/\#\texttt{unit}] \mid (\texttt{rn}[\texttt{op}/\#\texttt{arc}] \circ_{\#\texttt{arc}} LG.\#\texttt{conn})) \circ_{\#\texttt{conn},\#\texttt{unit}} (\texttt{iter} : \#\text{IT})$.

Program $\texttt{iter} : \#\text{IT}$ is a *local bridge program* (#IT is a local specification, hidden by #). The clause implementing $\texttt{iter} : \#\text{IT}$ has been derived from the following *bridge proof* $\texttt{pr}(\#lm1)$ and corresponds to lemma $bd1$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{I_{\#\text{IT}}}\,\texttt{rn},\wedge E_1}{\texttt{rn}[] : \exists x.\ !\#\texttt{unit}(x^-)} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{T(i \geq 0)} \quad \dfrac{}{I_{\#\text{IT}}}\,\texttt{rn},\wedge E_2,}{\forall E,\rightarrow E} \quad \dfrac{\dfrac{}{T(i \geq 0)} \quad \dfrac{}{!\#\texttt{unit}(u^-)} \quad !\#\texttt{conn}(u^+,x^+,y^-,i^+)}{(\texttt{iter} : \#\text{IT}) : !\texttt{iter}(x^+,i^+,y^-)}\,bd1}{\texttt{rn}[] : \exists y.\ !\#\texttt{conn}(u^+,x^+,y^-,i^+)}}{(\texttt{iter} : \#\text{IT}) : \exists!z.\ !\texttt{iter}(x^+,i^+,z^-)}\,\exists!I}{(\texttt{iter} : \#\text{IT}) : \exists!z.\ !\texttt{iter}(x^+,i^+,z^-)}\,\circ\exists E,\rightarrow I,\forall I,\rightarrow I}{(\texttt{iter} : \#\text{IT}) : I_{\#\text{IT}} \rightarrow E_{\text{IT}}}$$

!iter's mode follows from !#conn and modes $+$, $-$ are treated as usual in logic programs. $I_{\#\text{IT}}$ and E_{IT} denote the import and export parts of the interface #IT of *LGIT* (the export part coincides with the one of IT). The unicity T-formula needed in $\exists!I$ is inherited from *LG*. The empty renaming $\texttt{rn}[]$ is generated by the rule rn. For homogeneous programs (as in our example), $\texttt{rn}[]$ works as the identity for import composition. For heterogeneous programs, it corresponds to a "communication channel". $\circ\exists E$ is a derived rule and gives rise to $\texttt{rn}[] \circ (\texttt{iter} : \#\text{IT})$, equal to $(\texttt{iter} : \#\text{IT})$ [2]. The composite program implementing *LGIT*.iter is obtained by the following *composition proof*.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{I_{\text{IT}}}\,\texttt{rn},\wedge E_1}{\texttt{rn}[\texttt{unit}/\#\texttt{unit}] : \exists x.\ !\#\texttt{unit}(x^-)} \quad \dfrac{\dfrac{\dfrac{}{I_{\text{IT}}}\,\texttt{rn},\wedge E_2}{\texttt{rn}[\texttt{op}/\#\texttt{arc}] : I_{\#\text{C1}}} \quad LG.\#\texttt{conn} : I_{\#\text{C1}} \rightarrow E_{\#\text{C1}}[\texttt{pr}(LG)]}{\texttt{rn}[\texttt{op}/\#\texttt{arc}] \circ_{\#\texttt{arc}} LG.\#\texttt{conn} : E_{\#\text{C1}}}\,\rightarrow E}{\texttt{rn}[\texttt{unit}/\#\texttt{unit}] \mid (\texttt{rn}[\texttt{op}/\#\texttt{arc}] \circ_{\#\texttt{arc}} LG.\#\texttt{conn}) : I_{\#\text{IT}}} \quad (\texttt{iter} : \#\text{IT}) : I_{\#\text{IT}} \rightarrow E_{\text{IT}}[\texttt{pr}(\#lm1)]}\,|_1,|_2,\wedge I}{(\texttt{rn}[\texttt{unit}/\#\texttt{unit}] \mid (\texttt{rn}[\texttt{op}/\#\texttt{arc}] \circ_{\#\texttt{arc}} LG.\#\texttt{conn})) \circ_{\#\texttt{conn},\#\texttt{unit}} (\texttt{iter} : \#\text{IT}) : I_{\text{IT}} \rightarrow E_{\text{IT}}}\,\rightarrow E,\rightarrow I$$

$I_{\#\text{C1}}$ and $E_{\#\text{C1}}$ denote the import and export parts of interface #C1 (included from *LG* and hidden by *LGIT*). The *hiding renaming* $\texttt{rn}[\texttt{unit}/\#\texttt{unit}]$ generates the clause $\#\texttt{unit}(\texttt{X}) : -\texttt{unit}(\texttt{X})$ (# preserves the model theoretic semantics). $\texttt{rn}[\texttt{op}/\#\texttt{arc}]$ is not a purely hiding renaming. It requires to prove $\texttt{op}(x,y,z) \leftrightarrow \texttt{arc}(x,y,z)$ in the context and generates $\#\texttt{arc}(X,Y,Z) : -\texttt{op}(X,Y,Z)$. The operation $\mid$ is parallel program composition. *LGIT extends IT* and *LGIT*.iter *correctly overrides IT*.iter (the exported iter programs implement the same interface).

[2] Importing #conn from an heterogeneous *LG*, the communication channel $(\texttt{rn}[] \circ_{\#\texttt{conn}} (\texttt{iter} : \#\text{IT}))$ would be generated.

3 Behaviours and Program Composition

In this section, we define *behaviours* and we discuss the rationale behind them. A *behaviour* represents the knowledge obtained from an *observation* of a program unit by calling some of its procedures. The observer may be a human or another procedure. A call is characterized by the observed *call statement* C and by the *activation substitution* α, indicating the values observed for the variables of C when the call is activated. For conciseness, we do not consider negation, i.e., C is an atom. We call $C\alpha$ an *activation*. The (call corresponding to an) activation $C\alpha$ starts a computation, which may succeed, fail, yield an error or loop. According to the case, we define the *answer of* $C\alpha$ as follows:

- If the computation started by $C\alpha$ yields an error or loops, we do not have any answers.
- If the computation started by $C\alpha$ succeeds, we get an *answer substitution* β, indicating the values returned for the variables of $C\alpha$, if any.
- If the computation started by $C\alpha$ fails, we get the *negative answer* NO.

To consider backtracking, we use an *observation index*. The same index is only used when a call is repeated (by backtracking) to obtain a new answer. Otherwise, different observations have different index. Thus we define observation sequences and their components as follows (components underline backtracking).

- An *observation sequence* S is a sequence of *observations* of the form $\langle c : C\alpha, \beta\rangle$, where $C\alpha$ is an activation with observation index c and β is its answer
- the *component* with index c of S is the sequence $S[c]$ obtained by deleting all the observations of S with index different from c.

Example 3. Let us consider the composite procedure $\texttt{sum} \circ_{\texttt{sum}} \texttt{prod}$:

```
sum :  sum(X,0,X).
       sum(X,s(Y),s(Z)) :- sum(X,Y,Z).
prod : prod(X,0,0).
       prod(X,s(Y),Z)   :- prod(X,Y,P),sum(P,X,Z).
```

If `prod` observes the call statement $\texttt{sum}(\texttt{P},\texttt{X},\texttt{Z})$ in the computation of $\texttt{prod}(\texttt{s}(0),\texttt{s}(\texttt{s}(0)),0)$, it obtains the following sequence S with components $S[1]$, $S[2]$:

$$\begin{aligned} S &= \langle 1 : sum(0,s(0),v_0), v_0 = s(0)\rangle, \langle 2 : sum(s(0),s(0),0), \text{NO}\rangle, \\ &\quad \langle 1 : sum(0,s(0),v_0), \text{NO}\rangle \\ S[1] &= \langle 1 : sum(0,s(0),v_0), v_0 = s(0)\rangle, \langle 1 : sum(0,s(0),v_0), \text{NO}\rangle \\ S[2] &= \langle 2 : sum(s(0),s(0),0), \text{NO}\rangle \end{aligned}$$

The following property is required (and assumed) in our approach.

Property 3.1. Let $S[c] = \langle c : C\alpha, \beta_1\rangle, \ldots, \langle c : C\alpha, \beta_n\rangle$ be a component of an observation sequence S. If β_n is NO, then $\langle c : C\alpha, \beta_n\rangle$ is the last observation with observation index c occurring in S.

Property 3.1 codifies the fact that backtracking halts when the answer NO is reached. We define the *behaviour* $\mathcal{B}_{S[c]}$ of a component $S[c]$ as the following set of b-formulae:

- An atom A belongs to $\mathcal{B}_{S[c]}$ iff $S[c]$ contains an observation $\langle c : C\alpha, \beta\rangle$ s.t. $A = C\alpha\beta$.
- $T(\forall y.\ y \neq t_1 \wedge \cdots \wedge y \neq t_h \rightarrow \neg C\alpha)$ belongs to $\mathcal{B}_{S[c]}$ iff the last answer of $S[c]$ is NO and $t_1, \ldots, t_h$ are all the ground terms associated with the open variables y of $C\alpha$ by the answer substitutions of $S[c]$.

We define the *behaviour* $\mathcal{B}_S$ associated with an observation sequence S as the union of the behaviours of its components.

Example 4. The behaviours corresponding to the observation sequences of Ex. 3 are:
$\mathcal{B}_{S[1]} = \{sum(0, s(0), s(0)), T(\forall x.\ x \neq s(0) \rightarrow \neg sum(0, s(0), x))\}$
$\mathcal{B}_{S[2]} = \{T(\neg sum(s(0), s(0), 0))\}$
$\mathcal{B}_S \ = \mathcal{B}_{S[1]} \cup \mathcal{B}_{S[2]}$

Now we link behaviours and unit composition at the program level. A procedure p exported by P *depends* on a set π of import procedures of P iff every procedure called by the program implementing p belongs to π. By $P : \pi \Rightarrow \delta$ we mean that δ is the set of the export procedures, π is a set of import procedures and the procedures of δ depend on π. If $P : \emptyset \Rightarrow \delta$, P is a *closed program unit*, i.e., every and each program does not need other imported procedures to run. Program composition occurs when the contexts of two units have been merged. We can model this stage by two $\mathcal{C}$-units $P = \langle \Sigma, \mathcal{C}, Prog_1\rangle$ and $Q = \langle \Sigma, \mathcal{C}, Prog_2\rangle$ with a common context $\mathcal{C}$, which specifies *the import, export and hidden procedures of both* $Prog_1$ *and* $Prog_2$. That is, a set of $\mathcal{C}$-units is a set of separate program units with a common context $\mathcal{C}$. $\mathcal{C}$-units can be composed by *restriction*, *parallel* and *import composition*, yielding new $\mathcal{C}$-units, as follows:

Restriction. The *restriction* $R.\delta$ of R to a subset δ of the exported procedures is the $\mathcal{C}$-unit where those procedures and related interfaces not in δ have been deleted.
Parallel composition. The parallel composition $R \mid S$ is defined only if $R : \pi_1 \Rightarrow \delta_1$, $S : \pi_2 \Rightarrow \delta_2$, and $(\pi_1 \cup \pi_2) \cap (\delta_1 \cup \delta_2) = \emptyset$. The resulting $\mathcal{C}$-unit contains and exports $R.\delta_1$ and $S.\delta_2$, i.e.: $R \mid S : \pi_1 \cup \pi_2 \Rightarrow \delta_1 \cup \delta_2$. The (possible) shared procedures must have the same interfaces and implementations. We have: $R \mid S = S \mid R$.
Import composition. The import composition $R \circ_\gamma S$ is defined only if $R : \pi_1 \Rightarrow \delta_1 \cup \gamma$, $S : \pi_2 \cup \gamma \Rightarrow \delta_2$ and $(\pi_1 \cup \pi_2) \cap (\delta_1 \cup \delta_2) = \emptyset$. The resulting $\mathcal{C}$-unit exports $R.\delta_1$ and the procedures $R.\gamma$ are locally composed with $S.\delta_2$, so that they no longer depend on γ, i.e.: $R \circ_\gamma S : \pi_1 \cup \pi_2 \Rightarrow \delta_1 \cup \delta_2$. We have: $R \circ_{\gamma_1} (S \circ_{\gamma_2} T) = (R \circ_{\gamma_1} S) \circ_{\gamma_2} T$ and, if $R \mid S$ is defined, $R \circ_{\gamma_1} (S \circ_{\gamma_2} T) = (R \mid S) \circ_{\gamma_1 \cup \gamma_2} T$.

Example 5. We consider $\mathcal{C}$-units $P = \langle \Sigma, \mathcal{C}, Prog\rangle$ containing logic programs. Within P, we associate the axioms $Ax(\mathrm{p}) = Ocomp(\mathrm{p}) \cup CET(\mathrm{p})$ with the program-clauses implementing a procedure $P.\mathrm{p}$, where $Ocomp(\mathrm{p})$ is the open completion of (the clauses implementing) p [11] and $CET(\mathrm{p})$ is Clark's Equality Theory for p [12]. The operation $P \mid Q$ is defined if the general conditions introduced above for composition are satisfied and $Ax(\delta_1 \cup \delta_2) = Ax(\delta_1) \cup Ax(\delta_2)$. The resulting $\mathcal{C}$-unit contains and exports the programs for $\delta_1 \cup \delta_2$. Operation $P \circ_\gamma Q$ is defined if the general conditions introduced above for composition are satisfied and $Ax(\delta_1 \cup \gamma \cup \delta_2) = Ax(\delta_1 \cup \gamma) \cup Ax(\delta_2)$. The resulting $\mathcal{C}$-unit contains the programs for $\delta_1 \cup \gamma \cup \delta_2$, hides γ and exports $\delta_1 \cup \delta_2$. The hidden procedures γ are used locally by δ_2. Going back to Ex. 3, if *SUM* is the closed unit exporting `sum` and *PROD* the one importing the latter and exporting `prod`, the import

composition $SUM.\mathtt{sum} \circ_{\mathtt{sum}} PROD.\mathtt{prod} : \emptyset \Rightarrow \mathtt{prod}$ is defined and satisfies the above requirements. As we have seen, if we are only using Prolog programs, it suffices to put the various programs together, in the same name space. Hiding may be performed, e.g., by renaming. If we are using heterogeneous programs, we need also an environment supporting the communication among them. This is necessary for import composition.

Let $P \circ_\gamma Q : \emptyset \Rightarrow \delta$ be a $\mathcal{C}$-unit. We can build an observation sequence S_E by means of calls to δ. We call S_E an *experiment for* $P \circ_\gamma Q$. While doing the experiment S_E, we observe the calls performed by Q on γ. We obtain an observation sequence S_I, with a behaviour $\mathcal{B}_{S_I}$. We say that S_I is an *interface observation sequence* and $\mathcal{B}_{S_I}$ is an *interface behaviour* for $P \circ_\gamma Q$. We can abstract from the "server" unit P by introducing generic behaviours and "oracles".

A *generic b-formula* is a ground procedure-atom (i.e., built by a procedure symbol) or a T-formula of the form $T(\forall x.\ \neq t_1 \wedge \cdots \wedge x \neq t_n \rightarrow \neg A)$, where A is a procedure-atom with variables x. A *generic behaviour* is a consistent set $\mathcal{B}$ of generic b-formulae. $\mathcal{B}$ is an *import behaviour for* Q if the atoms of $\mathcal{B}$ contain only import procedures of Q. We now introduce $\mathcal{B}$-oracles ($\mathcal{B}[o]$): when an activation $C\alpha$ is performed with observation index c, a $\mathcal{B}$-oracle $\mathcal{B}[o]$ yields one of the following answers:

- $\langle c : C\alpha, \beta \rangle$, if there is an atom $A \in \mathcal{B}$ such that $A = C\alpha\beta$, or
- $\langle c : C\alpha, \mathrm{NO} \rangle$, if $\mathcal{B}$ contains $T(\forall y.\ y \neq t_1 \wedge \cdots \wedge y \neq t_n \rightarrow \neg C\alpha)$ and all the possible answers to $C\alpha$ have been given in previous steps, or
- the computation aborts, if none of the previous cases holds.

If there are different possible answers, the oracle makes a choice. It is only obliged to be *fair* on backtracking, that is, if an answer of an activation $C\alpha$ can be chosen, it will be. An oracle $\mathcal{B}[o]$ for a finite behaviour $\mathcal{B}$ can (in principle) be implemented and composed with other programs. To simulate moded procedures, an oracle may be moded. A *moded oracle* $\mathcal{B}[o_\mu]$ has a set μ of moded call statements and aborts if a call is activated that is not allowed by μ.

Example 6. Let P be a $\mathcal{C}$-unit containing logic programs and $\mathcal{B}$ be an import behaviour for P. A $\mathcal{B}$-oracle is a logic program that contains every atom $A \in \mathcal{B}$ as a fact and suitable clauses that abort the program when required. For example, the following program is an oracle for the behaviour $\mathcal{B}$ of Ex. 3.

```
sum(0,s(0),s(0)).
sum(s(0),s(0),s(s(0))).
sum(X,Y,Z) :- not((X == 0,Y == s(0))),
              not((X == s(0),Y == s(0),Z == s(s(0)))), abort.
```

Different oracles are obtained by changing the order of the facts and allowing possible repetitions. As an example of another kind of programming language style, we can consider program units containing imperative style procedures. If we only allow input-output procedures, the modes will always be of the kind $!\mathtt{p}(\mathtt{x}^+ ; \mathtt{y}^\downarrow)$. In the context, a model-theoretic specification $S_\mathtt{p}$ specifies p as a predicate. By the input mode $\mathtt{x}^+$, we declare that x are value parameters. By $\mathtt{y}^\downarrow$ we denote the *reference mode* (var-parameters in Pascal). In an activation, reference parameters can be only replaced by

variables. Oracles can be defined as follows: when a computation performs a call to an imported procedure q, we provide the result by a table-look-up mechanism, where the table contains a finite part of $\mathcal{B}$. By the selection mode !, only the first answer is considered; the computation aborts if no answer is found or the activation is not allowed by $!\mathtt{p}(\mathtt{x}^+ ; \mathtt{y}^\downarrow)$. That is, the oracle is moded.

If $\mathcal{J}$ is an interface behaviour of $P \circ_\gamma Q$ observed by an experiment S, then there is an oracle o_P such that $\mathcal{J}[o_P] \circ_\gamma Q$ replicates exactly the computation of $P \circ_\gamma Q$. However, a different oracle o' may yield a different computation of $\mathcal{J}[o'] \circ_\gamma Q$. We assume that Q is behaviourally stable (b-stable), i.e., we assume that for every o' there is an experiment S' for $\mathcal{J}[o'] \circ_\gamma Q$ with the same observed behaviour ($\mathcal{B}_{S'} \supseteq \mathcal{B}_S$ suffices). B-stability of Q means that it is able to cooperate with external units (simulated by the oracles) independently from the backtracking details, such as order or repetitions of the answers. We believe b-stability should be a property of high-level program units, because this enhances their re-usability. In the next section we will consider b-stability with respect to the moded import behaviours that realize their specification.

4 Behaviour Specifications and Their Realization

A *constructive specification* is a Σ-formula given by the following syntax, where BF stands for b-formulas (atoms built by a procedure symbol) and TF for T-formulas (of the form $T(F)$, where F is any Σ-formula):

$$\begin{aligned} \textit{Basic specifications } & BS ::= BF \mid TF. \\ \textit{Existential specifications } & ES ::= BS \mid ES \wedge ES \mid ES \vee ES \mid \exists x.\, ES. \\ \textit{Universal specifications } & US ::= ES \mid US \wedge US \mid \forall x.\, US. \\ \textit{Interfaces } & IC ::= US \mid US \to IC \mid IC \wedge IC \mid \forall x.\, IC. \end{aligned}$$

Universal specifications are a description of behaviours that are realized (exported) or used (imported) by a program unit. Interfaces relate imported and exported behaviours and model correct unit composition. They will be explained in the next section. Here, we consider universal specifications and their realization by closed $\mathcal{C}$-units within a signature Σ. For behaviours of closed programs we assume that data are *reachable* (i.e., representable by ground Σ-terms). The partial model theoretic correctness of programs is assumed, to ensure the truth (in $\mathcal{C}$) of the observed b-formulas. Thus a *behaviour correct in $\mathcal{C}$ is a set $\mathcal{B}$ of b-formulas of the signature Σ, such that $\mathcal{C} \models \mathcal{B}$.*

The *behaviour semantics* of universal specifications is given by the *realization* relation $|\!\models$. We write $P \,|\!\models H$ to denote that a $\mathcal{C}$-unit P realizes a specification H. This means that H is realized by the observable behaviour of P, in the way informally explained in Sect. 2. We firstly define realizability by correct behaviours, considered as sets of b-formulas true in $\mathcal{C}$. Then we define it by program units.

Definition 1 (Behaviour Realizability). *Let $\mathcal{B}$ be a behaviour correct in $\mathcal{C}$ and E a ground instance of an ES. Then $\mathcal{B} \,|\!\models E$ iff one of the following clauses applies:*
Basis. *For a T-formula $T(F)$, $\mathcal{B} \,|\!\models T(F)$ iff $\mathcal{C} \models F$.*
For a b-formula B, $\mathcal{B} \,|\!\models B$ iff $B \in \mathcal{B}$.

Step. *According to the cases, we have:*

- $\mathcal{B} \models F \wedge G$ *iff* $\mathcal{B} \models F$ *and* $\mathcal{B} \models G$.
- $\mathcal{B} \models F \vee G$ *iff* $\mathcal{B} \models F$ *or* $\mathcal{B} \models G$.
- $\mathcal{B} \models \exists x.\, F(x)$ *iff there is a ground term* t *such that* $\mathcal{B} \models F(t)$.
- $\mathcal{B} \models \forall x.\, F(x)$ *iff* $\mathcal{B} \models F(t)$, *for every ground term* t.

Theorem 1. *Let* $\mathcal{B}$ *be a correct behaviour and* U *an US.* $\mathcal{B} \models U$ *entails* $\mathcal{C} \models U$.

That is, the realized formulas are true in the context (assuming reachability). In the previous definition, behaviours may be infinite. Behaviors of observation sequences are finite. The following theorem can be easily proved.

Theorem 2. *Let* E *be an ES and* $\mathcal{B} \models E$. *Then there is a finite* $\mathcal{B}' \subseteq \mathcal{B}$ *s.t.* $\mathcal{B}' \models E$.

A realization of a universal specification U is, in general, infinite. It can be seen as the unions of the finite realizations of the existential instances of U.

Definition 2 (Inst(U)). *The set* $Inst(U)$ *of the instances of a ground universal specification* U *is the smallest subset of formulas satisfying the following clauses:*

- *if* U *is an ES, then* $U \in Inst(U)$;
- *if* t *is a ground term and* $F \in Inst(H(t))$, *then* $F \in Inst(\forall x.\, H(x))$;
- *if* $F \in Inst(H)$ *and* $G \in Inst(K)$, *then* $F \wedge G \in Inst(H \wedge K)$.
- *If* U *is open,* $Inst(U)$ *is the union of the* $Inst(U\sigma)$, *where* σ *is a grounding substitution.*

Theorem 3. *Let* U *be a ground universal specification and* $\mathcal{B}$ *be a behaviour.* $\mathcal{B} \models U$ *iff for every instance* $E \in Inst(U)$ *there is a finite* $\mathcal{B}' \subseteq \mathcal{B}$ *s.t.* $\mathcal{B}' \models E$.

Now we can consider realizability by program units. Since T-formulas do not require any realization, the (possible) procedure symbols occurring in them do not require computations. We say that they are *hidden* by T, or *inactive*; non-hidden occurrences are called *active*. A specification S is an *export specification* for a closed program unit $P : \emptyset \Rightarrow \delta$ iff the active call statements of S have procedure symbols from δ. Program realizability is defined starting from finite experiments. Let E be a ground export existential specification for P. We say that P *realizes* E *with a (finite) experiment* S, written $P \models_S E$, iff S is a finite experiment for P such that $\mathcal{B}_S \models E$. For a universal specification U, we need to validate the various instances $I \in Inst(U)$ by U*-moded experiments*, namely experiments that apply moded call statements from U.

Definition 3 (Program Realizability of US**).** *Let* U *be an export universal specification for a closed* $\mathcal{C}$*-unit* P. $P \models U$ *iff for every instance* $I \in Inst(U)$ *there is an* U*-moded experiment* S *s.t.* $P \models_S I$.

Export US are specifications for closed program units. In particular, decision, enumeration, selection and functional behaviours can be specified as explained in Sect. 2. We can use a universal specification U also as an assumption on the expected import behaviuor of an open unit. In this case, U states the following *collaboration agreement*:

a client unit Q is assumed to apply only moded call statements of U, while a server unit P is assumed to answer without loop or abort errors, when it is called according to the modes declared in U. Concerning the server side (the client side will be considered in the next section), we require correct moding, as the past history is needed to properly treat the call index i.

Definition 4 (Correct Moding). *Let U be an export universal specification for a closed $\mathcal{C}$-unit P. U is* correctly moded *with respect to P iff every U-moded experiment S for P can be continued into an experiment $S, \langle i : C\alpha, ans\rangle$, whenever the activation $i : C\alpha$ is legal with respect to the modes of U and the past history S.*

5 Interfaces

Interfaces allow us to specify open program units and their clients relations. To properly abstract from moded server units, we introduce *U-moded $\mathcal{J}$-oracles* $\mathcal{J}[o_U]$, where $\mathcal{J}$ is a possibly infinite [3] behaviour, U is a universal specification and o_U is a moded oracle with moded call statements from U. An experiment S for $\mathcal{J}[o_U]$ is obtained through the answers chosen by o_U and $\mathcal{J}[o_U] \mid\mid\models U$ is defined as in Def. 3. We assume *fairness* (if $\mathcal{J}$ contains an answer for a call statement $i : ?C\alpha$, this answer will be chosen by o_U) and *correct moding* (U is correctly moded with respect to o_U, i.e., each legal continuation of an experiment is answered by o_U).

For every server program unit P realizing U there are a behaviour $\mathcal{J}$ and an U-moded oracle o_U, such that the experiments of P coincide with those of $\mathcal{J}[o_U]$. We use $\mathcal{J}[o_U]$ as abstractions of server units and we introduce the semantics of interfaces. We proceed gradually. We say that $U \rightarrow V$ is a *simple interface* if U, V are universal specifications. U must be an *import specification*, that is a specification with only import-active call statements. It is an assumption on the possible import behaviours. V must be an *export specification*, that is a specification with only export-active call statements. It states the expected export behaviour.

Definition 5 (Realizability for Simple Interfaces). *Let Q be a $\mathcal{C}$-unit and $U \rightarrow V$ a simple interface. $Q \mid\mid\models U \rightarrow V$ iff U is an import specification, V an export specification and for every import oracle $\mathcal{J}[o_U]$ such that $\mathcal{J}[o_U] \mid\mid\models U$, it holds $\mathcal{J}[o_U] \circ_U Q \mid\mid\models V$.*

This definition of realisability requires b-stability (see Sect. 3) with respect to the behaviours realizing U (we may choose, for a behaviour $\mathcal{J}$, any oracle o_U). The compositional meaning of simple interfaces is given by the following theorem.

Theorem 4. *Let P be a $\mathcal{C}$-unit such that $P \mid\mid\models U$ and U correctly moded with respect to P and let Q be a $\mathcal{C}$ unit such that $Q \mid\mid\models U \rightarrow V$, for a simple interface $U \rightarrow V$. Then $P \circ_U Q \mid\mid\models H$.*

Simple interfaces represent simple composition rules, where the import behaviour U has to be realized by the server unit as a whole. It may be useful to choose server units

[3] We could consider only finite behaviours, but this would complicate our treatment.

incrementally, by building intermediate open units. To this aim, we allow conjunctions and right-nested implications. For example, the interfaces $U \to (V \to H)$ allows us to choose a server unit for U and delay the choice of the unit for V.

Definition 6 (Realizability for Interfaces). *Let Q be a $\mathcal{C}$-unit and I be an IC. We inductively define* realizability*, as follows:*
B*asis. I is a universal specification. Realizability is defined as in the previous section.*
S*tep. According to the cases:*

- *I is $U \to H$. We proceed by a secondary induction on H. The* base case *coincides with simple interfaces. The* step case *is as follows:*
  - *H is $V \to K$. $Q \mid\models U \to (V \to K)$ iff U, V are universal import specifications for Q and $Q \mid\models U \wedge V \to K$.*
  - *H is $H_1 \wedge H_2$. $Q \mid\models U \to (H_1 \wedge H_2)$ iff U is a universal import specification for Q, $Q \mid\models U \to H_1$ and $Q \mid\models U \to H_2$.*
  - *H is $U \to \forall x.\, H(x)$. $Q \mid\models U \to \forall x.\, H(x)$ iff U is a universal import specification for Q and $Q \mid\models U \to H(t)$, for every ground term t.*
- *I is $H \wedge K$. $Q \mid\models H \wedge K$ iff $Q \mid\models H$ and $Q \mid\models K$.*
- *I is $\forall x.\, H(x)$. $Q \mid\models \forall x.\, H(x)$ iff $Q \mid\models H(t)$ for every ground t.*

Composition proofs, bridge proofs and correct modes. The compositional meaning of interfaces is given by the rules of the compositional calculus shown in Sect. 5.1. By these rules, we can correctly compose $\mathcal{C}$-units at the program level. Our syntax distinguishes universal specifications and interfaces. Correspondingly, we have two kinds of proofs. A proof that applies only the renaming rule `rn` and the rules for $\to$, $\wedge$, $\forall$ to interfaces is called a *composition proof*. It allows us to compose units whose interfaces match via (possible) renaming. If this level of "exact" matching fails, then we de-structure the universal specifications used in the interfaces and we try to re-structure them in a different form, by means of the other rules and possible "bridge lemmas", such as *bdl*1 in Ex. 2. Proofs of this kind are called *bridge proofs*. Correct modes are required to prove the validity theorem (Thm. 5). They are preserved by composition proofs, but might be destroyed by the bridge proofs. For example, a (trivial) unit containing the procedure $\mathtt{p(a)}$ realizes $\exists x.\, !p(x^-)$. If we apply the $\vee I_2$-rule without restrictions, we prove that it realizes $\exists x.\, !p(x^-) \vee \exists x.\, !q(x^-)$. But $:-\mathtt{q(X)}$ fails, while it should succeed by mode !. Thus correct moding should be always checked in bridge proofs. Fortunately, there are universal specifications that are correctly moded independently from the implementation details. Specifications of this kind are called *correctly moded* (with respect to any unit P). They have the following property: for every instance $E \in Inst(U)$, every finite behaviour $\mathcal{B}$ such that $\mathcal{B} \mid\models E$, and every moded oracle o_U (applying the modes of U), there is an observation sequence S for $\mathcal{B}[o_U]$ such that $\mathcal{B}[o] \mid\models_S E$. There are *syntactical sufficient conditions* for correct moding (omitted here). For example, $A \vee T(F)$ is correctly moded if so is A and it has the query mode (modes of complex formulas are determined starting from those in call statements).

Bidirectional interfaces. In our syntax, interfaces model *unidirectional composition*. If we enlarge the syntax by allowing interfaces of the form $(H \to K) \to R$, we can model *bidirectional composition*. We have not yet completed the analysis of this case,

so we only briefly comment on it, by considering the simpler case $(U \to V) \to W$, with U, V, W universal specifications. The active procedures that occur positively (namely in U and W) must be export procedures, while the ones occurring negatively (namely in V) must be import procedures. Moreover, the export procedures of U must not depend on the import procedures of V. In this case, the interface correctly specifies a bidirectional collaboration, as follows. We say that $Q \mid\models (U \to V) \to W$ iff $Q \mid\models U$ and $Q \mid\models V \to W$. A server unit P has to realize $U \to V$. We want to get a bidirectional composition $\circ_{U \to V}$ such that $P \circ_{U \to V} Q \mid\models W$. Under our hypotheses, we can say: $P \circ_{U \to V} Q = (Q \circ_U P) \circ_V Q$. That is, P uses the behaviour for U exported by Q to realize the behaviour for V needed by Q. Q uses this behaviour to realize W.

5.1 The Compositional Calculus

In this section we present a possible compositional calculus, where proof-trees are in the style of natural deduction. The root of a proof-tree is of the form $P : A$, where P is a logic program and A is a constructive specification. $P : A$ is the *consequence* of the proof tree. Assumptions are specifications. The rules are shown next.

$$\frac{H_1, \ldots, H_n}{P : T(F)}\,\mathtt{tt}(L) \qquad \frac{H_1, \ldots, H_n}{P : A}\,\mathtt{pr}(L) \qquad \frac{\mathit{false}}{P : A}\,\mathtt{ff} \qquad \frac{H_1, \ldots, H_n}{\mathtt{rn}[\rho] : \rho\, H_i}\,\mathtt{rn} \qquad \frac{P_i : A}{P_1 \mid P_2 : A}\,|_i$$

$$\frac{P : A_1 \qquad P : A_2}{P : A_1 \wedge A_2}\,\wedge I \qquad \frac{P : A_1 \wedge A_2}{P : A_i}\,\wedge E_i \qquad \frac{P : A_i}{P : A_1 \vee A_2}\,\vee I_i \qquad \frac{A_1 \vee A_2 \qquad \begin{array}{cc}[A_1] & [A_2] \\ \vdots & \vdots \\ P : C & P : C\end{array}}{P : C}\,\vee E$$

$$\frac{P : A(t)}{P : \exists x.\, A(x)}\,\exists I \qquad \frac{\exists x.\, A(x) \qquad \begin{array}{c}[A(a)] \\ \vdots \\ P : C\end{array}}{P : C}\,\exists E \qquad \frac{\begin{array}{c}[A] \\ \vdots \\ P : B\end{array}}{P : A \to B} \to I \qquad \frac{P : A \qquad Q : A \to B}{P \circ_A Q : B} \to E$$

$$\frac{P : A(a)}{P : \forall x.\, A(x)}\,\forall I \qquad \frac{P : \forall x.\, A(x)}{P : A(t)}\,\forall E$$

The rules apply to $\mathcal{C}$-units at the program level. The program rule $\mathtt{pr}(L_j)$ allows us to use an already proven `pr`-*"lemma"* L_j about a unit P. It is also possible to introduce knowledge from the problem context by the rule $\mathtt{tt}(L_k)$, indicating a `tt`-*"lemma"* L_k, proving $\mathcal{C} \models H_1 \wedge \cdots \wedge H_n \to F$. Lemmas (in particular `tt`-lemmas) are not necessarily developed in the compositional calculus, but may use any system consistent with classical logic, or they may be informal. Rule `rn` encodes renaming, which may be needed for hiding or bridging purposes (see Ex. 2). The application of `rn` is systematically enforced by our calculus (programs do not occur in the assumptions, but are needed in the consequences). The idea is that renaming correspond to "communication channels". The way of forcing renaming is preliminary and needs to be further experimented. The other rules are similar to the ones of intuitionistic predicate logic. The

usual provisos apply. We require also that the involved formulas are constructive specifications in the syntax given in Sect. 4. This entails, in particular, that the assumptions may only be universal specifications and that the rules for $\vee$, $\exists$ can be only applied to existential specifications. To deal with assumptions, we enlarge our definition of interface and allow interface sequents $\Gamma \Rightarrow K$, where Γ is a set of interfaces that are universal specifications:

Definition 7 (Realizability for Interface Sequents). *Let $\Gamma \Rightarrow K$ be an interface sequent for a $\mathcal{C}$-unit P. We say that P realizes $\Gamma \Rightarrow K$, written $P \mid\!\models \Gamma \Rightarrow K$, iff for every grounding Σ-substitution σ, $P \mid\!\models (\wedge(\Gamma) \rightarrow K)\sigma$.*

The following theorem states the validity of the calculus. The proof considers separately the bridge proofs, which structure and de-structure universal specifications and the composition proofs, which structure and de-structure the interfaces. For the bridge proofs, the preservation of correct moding ought to be checked.

Theorem 5 (Validity). *Consider a proof-tree with assumptions Γ and consequence $P : H$. If the (possibly informal)* `tt`*-lemmas and* `pr`*-lemmas are correct, then $P \mid\!\models \Gamma \Rightarrow H$.*

6 Discussion

Constructive specifications make explicit aspects that are important for correct composition. They are based on correct modes and on a constructive semantics. This semantics is not alternative, but complementary to our previous model-theoretic approach [10]: the reference semantics still remains classical model theory and constructive logic is used to obtain more expressive interface specifications. The idea of using constructive logic to specify interfaces has been influenced by [16], even if we use a different semantics. The semantics and the calculus explained here are a first step and have been inspired by the *collection semantics* introduced in [13, 14] to prove constructivity results for intermediate first order systems. The T-operator for formulas that are not constructively evaluated comes from [15]. The use of distinguished levels for the problem context and for programs and their interfaces, as well as the possibility of using different programming languages is similar to the Larch specification language [8], although in the latter constructive logic is not used. We next outline some improvements to our work.

Constructive proofs bridging not exactly coinciding specifications are, in general, simpler than the proofs needed to derive programs from scratch. A study of proof strategies oriented to module composition and reuse would be interesting.

Behaviours with closed formulae allow us to capture input-output modes with ground results and the query mode. A possible improvement, as far as logic programs are concerned, is to consider open b-formulas, as e.g. in [1], to capture more general kinds of modes and compositions, including open answers.

The restriction to correctly moded universal specifications guarantees that a constructive evaluation of the export specifications of a server program unit P can be obtained by a client unit Q, by querying P. This explains also our restricted syntax. For example, we do not allow a disjunction such as $(\forall x.\, A(x)) \vee (\forall x.\, B(x))$. Indeed, to state

that ($A(t)$ holds for all the ground terms t), or ($B(t)$ does), a (possibly) infinite computation is needed. On the other hand, if we know that, say, $\forall x.\, A(x)$ holds, we can use this information by inserting it in the behaviour. In this way, is possible to use any first order formula as a constructive specification. The price to pay is that we have to introduce in behaviours the knowledge needed to trace constructive evaluations, along the lines of [13]. The advantage of a restricted syntax is that it adapts to the operational semantics of any kind of programs, with the only requirement that the model theoretic meaning of the procedures and functions is to be specified in the context.

Nevertheless, it is useful to investigate possible extensions, related to other kinds of module operations. In particular, it would be useful to extend $\circ$ to bidirectional collaboration of program units. This corresponds to the use of implications of the form $(A \to B) \to C$. From a first partial analysis of this case (see the discussion in Sect. 5), it seems that it can be treated without altering the general lines of our approach. In this way we get a notion of interface that has similarities to the one discussed in [3].

The requirement of b-stability is suited to relatively complete programs, while it may not work for small pieces of code. In this case a lower level of abstraction is required. It can be introduced by a different definition of the behaviour associated to observation sequences and experiments. We believe that an analysis of different levels of abstraction is interesting and potentially fruitful. In particular, it would be interesting to consider behaviours as multisets to exploit the use of linear logic programming techniques [9] to express properties of program units that consume resources and to consider their geometry of collaboration.

In this paper we have concentrated on the definition of behaviour semantics and we have devised a first composition calculus. We have a validity result. It is difficult to even define completeness, due to the presence of modes and the fact that we want to deal with heterogeneous systems. The next step is to experimentally check our ideas and our calculus in real examples, e.g. addressing case studies of formal specification.

References

1. A. Bossi, M. Gabbrielli, G. Levi, and M.C. Meo. A compositional semantics for logic programs. *Theoretical Computer Science*, 122:3–47, 1994.
2. 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.
3. L. de Alfaro and T. Henzinger. Interface Theories for Component-based Design Proc. of EMSOFT 2001, LNCS 2211, pp. 148–165, Springer Verlag, 2001.
4. S.K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs, ACM Transactions on Programming Languages and Systems, 11(3):418–450, 1989.
5. Y. Deville. *Logic Programming. Systematic Program Development*. Addison-Wesley, 1990.
6. D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.
7. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2*. Springer-Verlag, 1989.
8. J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
9. J. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation*, 110(2):327–365, 1994.

10. K.-K. Lau and M. Ornaghi. Specifying Compositional Units for Correct Program Development in Computational Logic. *Program Development in Computational Logic: A Decade of Research Advances in Logic-Based Program Development*, LNCS, vol 3049, pp. 1–29, 2004
11. K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *J. Logic Programming*, 38(3):259–294, March 1999.
12. J.W. Lloyd. *Foundations of Logic Programming*. 2nd ed., Springer-Verlag, 1987.
13. P. Miglioli, M. Ornaghi. A logically justified model of computation I , II *Fundamenta Informaticae*, 4(1): 151-172, 4(2): 277-342 , 1981.
14. P. Miglioli, U. Moscato, M. Ornaghi. Constructive theories with abstract data types for program synthesis In D.G. Skordev, editor, *Mathematical Logic and its Applications*, pages 293–302. Plenum Press, 1987.
15. P. Miglioli, U. Moscato, M. Ornaghi and G. Usberti. A Constructivism based on classical truth *Notre Dame Journal of Formal Logic*, 30(1):67–90, 1989.
16. D. Miller. A logical analysis of modules in logic programming. *JLP*, 6(1-2):79–108, 1989.
17. C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.