Specifications

- Basic specifications
- Specification engineering
- Structured specifications
- Proof systems for specifications

What are specifications for?

For the system user: specification captures the properties of the system the user can rely on.

For the system developer: specification captures all the requirements the system must fulfil.

Specification engineering

Specification development: establishing desirable system properties and then designing a specification to capture them.

Specification validation: checking if the specification does indeed capture the expected system properties.

Formal specifications

Model-oriented approach: give a specific model — a system is *correct* if it displays the same behaviour.

Property-oriented approach: give a list of the properties required — a system is *correct* if it satisfies all of them.

In either case, start by:

- determining the logical system to work with
- providing the signature of the system to be specified

Specification validation

- Theorem proving
- Prototyping and testing

Make specification understandable!

Key idea: STRUCTURE

Use it to:

- build, understand and prove properties of specifications
- (though not necessarily to implement them)

Specification languages

Quite a few around . . . Choose one.

Of course, you should be choosing
CASL

:-)

Semantics

Given a specification SP, determine:

- its *signature*: $Sig[SP] \in |\mathbf{Sign}|$
- its *models*: $Mod[SP] \subseteq |\mathbf{Mod}(Sig[SP])|$

Specification equivalence

$$\boxed{SP_1 \equiv SP_2}$$

means:

$$Sig[SP_1] = Sig[SP_2] \text{ and } Mod[SP_1] = Mod[SP_2]$$

Basic specifications

$$\boxed{\langle \Sigma, \Phi \rangle}$$

- $Sig[\langle \Sigma, \Phi \rangle] = \Sigma$
- $Mod[\langle \Sigma, \Phi \rangle] = Mod(\Phi)$

Keep them small . . .

Nothing to add, I think.

Structured specifications

Built by combining, extending and modifying simpler specifications

Specification-building operations

We will look at three elementary, but quite flexible **sbo**'s:

union: to combine constraints imposed by various specifications

translation: to rename and introduce new components, following a signature morphism

hiding: to hide interpretation of auxiliary components, receding to the source of a signature morphism

Union

Given specifications SP_1 and SP_2 with $Sig[SP_1] = Sig[SP_2]$,

$$SP_1 \cup SP_2$$

is a specification with the following semantics:

- $Sig[SP_1 \cup SP_2] = Sig[SP_1] (= Sig[SP_2])$
- $Mod[SP_1 \cup SP_2] = Mod[SP_1] \cap Mod[SP_2]$

CASL syntax: SP_1 **and** SP_2

Translation

Given a specification SP and a signature morphism $\sigma\colon Sig[SP] \to \Sigma'$,

$$\sigma(SP)$$

is a specification with the following semantics:

- $Sig[\sigma(SP)] = \Sigma'$
- $Mod[\sigma(SP)] = \{M' \in |\mathbf{Mod}(\Sigma')| \mid M'|_\sigma \in Mod[SP]\}$

CASL syntax: SP **with** $s \mapsto s', f \mapsto f', \ldots$

Actually, in CASL: SP_1 **and** $SP_2 \equiv \iota_1(SP_1) \cup \iota_2(SP_2)$
where $Sig[SP_1] \xrightarrow{\iota_1} Sig[SP_1] \cup Sig[SP_2] \xleftarrow{\iota_2} Sig[SP_2]$

Hiding

Given a specification SP' and a signature morphism $\sigma\colon \Sigma \to Sig[SP']$,

$$\sigma^{-1}(SP')$$

is a specification with the following semantics:

- $Sig[\sigma^{-1}(SP')] = \Sigma$
- $Mod[\sigma^{-1}(SP')] = \{M'|_\sigma \mid M' \in Mod[SP']\}$

CASL syntax (for signature inclusions only): SP' **reveal** $s, f, \ldots$
or SP' **hide** $s', f', \ldots$

Normal forms

Fact: *Any specification built out of basic specifications using union and translation only is equivalent to a basic specification.*

Fact: *If the category of signatures has pushouts and the institution admits amalgamation, then any specification SP built out of basic specifications using union, translation and hiding may be equivalently transformed to its normal form:*

$$\mathbf{nf}(SP) = \sigma_{res}^{-1}(\langle \Sigma_{all}, \Phi_{all} \rangle)$$

Know about them — never use them

Semantic consequence

$$\boxed{SP \models_\Sigma \varphi}$$

Σ-sentence φ is a semantic consequence of a Σ-specification SP

if φ holds in every model of SP, i.e.,

$$\varphi \in Th(Mod[SP])$$

One way to prove this

$$\frac{\mathbf{nf}(SP) = \sigma_{res}^{-1}(\langle \Sigma_{all}, \Phi_{all} \rangle) \qquad \Phi_{all} \vdash_{\Sigma_{all}} \sigma_{res}(\varphi)}{SP \vdash \varphi}$$

This is sound and complete when the category of signatures has pushouts, the institution admits amalgamation, and $\vdash_{\Sigma_{all}}$ is sound and complete, but:

This is a bad way!

- lack of compositionality
- no use of the specification structure
- typically, Φ_{all} is HUGE
- no help in proof search

Compositional proof system

$$\frac{\varphi \in \Phi}{\langle \Sigma, \Phi \rangle \vdash \varphi} \qquad \frac{SP_1 \vdash \varphi}{SP_1 \cup SP_2 \vdash \varphi} \qquad \frac{SP_2 \vdash \varphi}{SP_1 \cup SP_2 \vdash \varphi}$$

$$\frac{SP \vdash \varphi}{\sigma(SP) \vdash \sigma(\varphi)} \qquad \frac{SP' \vdash \sigma(\varphi)}{\sigma^{-1}(SP') \vdash \varphi}$$

Plus a *structural rule* (relying on a proof system for the underlying institution to derive $\Phi \vdash \varphi$):

$$\frac{SP \vdash \varphi_1 \quad \cdots \quad SP \vdash \varphi_n \qquad \{\varphi_1, \ldots, \varphi_n\} \vdash \varphi}{SP \vdash \varphi}$$

Soundness

$$\boxed{SP \vdash \varphi \implies SP \models \varphi}$$

Completeness

Fact: *If the category of signatures has pushouts, the institution admits amalgamation and interpolation, and has a sound and complete proof system to derive $\Phi \vdash \varphi$, then*

$$\boxed{SP \vdash \varphi \iff SP \models \varphi}$$

In general: there is *no* sound and complete *compositional* proof system for semantic consequence for structured specifications.

Example: (double/linear) hash table

spec $\textsc{Nat} = \ldots$

spec $\textsc{String} = \ldots$

spec $\textsc{Elem} =$ **sort** Elem

spec $\textsc{StringKey} = \textsc{String}$ **and** $\textsc{Nat}$
  then op $\mathit{hash} : \mathit{String} \to \mathit{Nat}$

spec $\textsc{StringKey0} = \textsc{StringKey}$ **with** $\mathit{hash} \mapsto \mathit{hash0}$

spec $\textsc{Array\_of\_Elem} = \textsc{Elem}$ **and** $\textsc{Nat}$
  then sort $\mathit{Array}[\mathit{Elem}]$
    ops $\mathit{empty} : \mathit{Array}[\mathit{Elem}]$;
      $\mathit{put} : \mathit{Nat} \times \mathit{Elem} \times \mathit{Array}[\mathit{Elem}] \to \mathit{Array}[\mathit{Elem}]$;
      $\mathit{get} : \mathit{Nat} \times \mathit{Array}[\mathit{Elem}] \to \mathit{Elem}$
    pred $\mathit{used} : \mathit{Nat} \times \mathit{Array}[\mathit{Elem}]$
    $\forall\, i, j : \mathit{Nat}; e : \mathit{Elem}; a : \mathit{Array}[\mathit{Elem}]$

- $\neg \mathit{used}(i, \mathit{empty})$
- $\mathit{used}(i, \mathit{put}(i, e, a))$
- $i \neq j \implies (\ \mathit{used}(i, \mathit{put}(j, e, a)) \iff \mathit{used}(i, a)\ )$
- $\mathit{get}(i, \mathit{put}(i, e, a)) = e$
- $i \neq j \implies \mathit{get}(i, \mathit{put}(j, e, a)) = \mathit{get}(i, a)$

Parametrized specification

spec $\textsc{Array}[\textsc{Elem}] = \textsc{Array\_of\_Elem}$

$\textsc{Array}[\mathit{SP}$ **fit** $\mathit{Elem} \mapsto \mathit{Asort}]$
stands for
$\{\textsc{Array}[\textsc{Elem}]$ **with** $\mathit{Elem} \mapsto \mathit{Asort}\}$ **and** SP

spec $\textsc{Bucket} = \textsc{Array}[\textsc{String}$ **fit** $\mathit{Elem} \mapsto \mathit{String}]$
  with $\mathit{Array}[\mathit{String}] \mapsto \mathit{Bucket}$

spec $\textsc{Table} = \textsc{Array}[\textsc{Bucket}$ **fit** $\mathit{Elem} \mapsto \mathit{Bucket}]$
  with $\mathit{Array}[\mathit{Bucket}] \mapsto \mathit{Table}$

$\textsc{StringHashTable0} = \textsc{StringKey0}$ **and** $\textsc{Bucket}$
  then ops $\mathit{add} : \mathit{String} \times \mathit{Bucket} \to \mathit{Bucket}$;
    $\mathit{putnear} : \mathit{Nat} \times \mathit{String} \times \mathit{Bucket} \to \mathit{Bucket}$
  preds $\mathit{present} : \mathit{String} \times \mathit{Bucket}$
    $\mathit{isnear} : \mathit{Nat} \times \mathit{String} \times \mathit{Bucket}$
  $\forall\, i : \mathit{Nat}; s : \mathit{String}; b : \mathit{Bucket}$

- $\mathit{add}(s, b) = \mathit{putnear}(\mathit{hash0}(s), s, b)$
- $\neg \mathit{used}(i, b) \implies \mathit{putnear}(i, s, b) = \mathit{put}(i, s, b)$
- $\mathit{used}(i, b) \wedge \mathit{get}(i, b) = s \implies \mathit{putnear}(i, s, b) = b$
- $\mathit{used}(i, b) \wedge \mathit{get}(i, b) \neq s \implies$
  $\mathit{putnear}(i, s, b) = \mathit{putnear}(\mathit{succ}(i), s, b)$
- $\mathit{present}(s, b) \iff \mathit{isnear}(\mathit{hash0}(s), s, b)$
- $\neg \mathit{used}(i, b) \implies \neg \mathit{isnear}(i, s, b)$
- $\mathit{used}(i, b) \wedge \mathit{get}(i, b) = s \implies \mathit{isnear}(i, s, b)$
- $\mathit{used}(i, b) \wedge \mathit{get}(i, b) \neq s \implies (\cdots)$

$\textsc{StringHashTable} =$

$\quad \textsc{StringHashTable0}$ **and** $\textsc{StringKey}$ **and** $\textsc{Table}$

$\qquad$ **then op** $add : String \times Table \to Table$

$\qquad\quad$ **pred** $present : String \times Table$

$\qquad\quad \forall\, i : Nat; s : String; t : Table$

- $hash(s) = i \wedge used(i,t) \implies$
  $add(s,t) = put(i, add(s, get(i,t)), t)$
- $hash(s) = i \wedge \neg used(i,t) \implies$
  $add(s,t) = put(i, add(s, empty), t)$
- $hash(s) = i \wedge used(i,t) \implies$
  $(present(s,t) \iff present(s, get(i,t)))$
- $hash(s) = i \wedge \neg used(i,t) \implies \neg present(s,t)$

spec $\textsc{UserStringHashTable} =$

$\quad \textsc{StringHashTable}$

$\qquad$ **reveal** $String, nil, a, \ldots, z, \_\,^\frown\,\_, Table, empty : Table,$

$\qquad\qquad add : String \times Table \to Table, present : String \times Table$

Specification structure

- This is a (nicely) structured specification
- The specification structure can guide, for instance, proof search
- The specification structure does not prescribe the structure of programs that implement the specification

spec $\textsc{SimpleUserStringHashTable} = \textsc{String}$

$\quad$ **then sort** $Table$

$\qquad$ **ops** $empty : Table;$

$\qquad\quad add : String \times Table \to Table$

$\qquad$ **pred** $present : String \times Table$

$\qquad \forall\, s, s' : String, t : Table$

- $\neg present(s, empty)$
- $present(s, add(s,t))$
- $s \neq s' \implies (present(s, put(s',t)) \iff present(s,t))$

Program development

- Simple refinements
- Constructor refinements
- Local constructions in global refinement steps
- Architectural specifications

Working in an arbitrary institution