Specifications and formal program development
“in-the-large”

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.
- prototyping and testing
- theorem proving
**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 . . .

We will (pretend to) work in the standard algebraic framework

**BUT:** *everything carries over to more complex, and more realistic logical systems, capturing the semantics of more realistic programming paradigms.*

*more about this elsewhere: Institutions!*
Specification languages

Quite a few around... Choose one.

Of course, you should be choosing Casl :-)

Make even realistic large specification understandable!

Key idea: STRUCTURE

Use it to:

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

Given a requirements specification produce a module that correctly implements it

Given a requirements specification $SP$
build a program $P$ such that

$$SP \leadsto P$$

A formal definition of $SP \leadsto P$ is a given by the semantics
(of the specification formalism and of the programming language)
Recall the analogy:

module interface \sim\ signature
module \sim\ algebra
module specification \sim\ class of algebras

**Specification semantics**

Given a specification \( SP \):

- **signature of** \( SP \): \( \text{Sig}[SP] \)
- **models of** \( SP \): \( \text{Mod}[SP] \subseteq \text{Alg}(\text{Sig}[SP]) \)

**Specification equivalence:**

\( SP_1 \equiv SP_2 \) means \( \text{Sig}[SP_1] = \text{Sig}[SP_2] \) and \( \text{Mod}[SP_1] = \text{Mod}[SP_2] \)

**Specification consequences:**

\( SP \models \varphi \) means \( M \models \varphi \) for all \( M \in \text{Mod}[SP] \)
Basic specifications

\[ \langle \Sigma, \Phi \rangle \]

- \[ \text{Sig}[\langle \Sigma, \Phi \rangle] = \Sigma \]
- \[ \text{Mod}[\langle \Sigma, \Phi \rangle] = \text{Mod}(\Phi) \]

*Keep them small...*

Nothing to add, I think.
Structured specifications

Built by combining, extending and modifying simpler specifications

Specification-building operations

For instance:

**union:** to combine constraints imposed by various specifications

**translation:** to rename and introduce new components

**hiding:** to hide interpretation of auxiliary components

*Three typical, elementary, but quite flexible sbo’s*
Programmer’s task

Informally:

Given a requirements specification produce a module that correctly implements it

Semantically:

Given a requirements specification $SP$ build a model $M \in \text{Alg}(\text{Sig}[SP])$ such that

$M \in \text{Mod}[SP]$
Key idea

\[ SP \sim M \]

Never in a single jump!

Rather: proceed step by step, adding gradually more and more detail and incorporating more and more design and implementation decisions, until a specification is obtained that is easy to implement directly.
Development process:

$SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_n$

ensuring:

\[
\frac{SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_n \quad SP_n \rightsquigarrow M}{SP_0 \rightsquigarrow M}
\]
Simple implementations

\[ SP \mapsto SP' \]

Means:

\[ \text{Sig}[SP'] = \text{Sig}[SP] \text{ and } \text{Mod}[SP'] \subseteq \text{Mod}[SP] \]

So:

- preserve the static interface (by preserving the signature)
- incorporate further details (by narrowing the class of models)
Composability

\[
\begin{align*}
SP & \implies SP' & SP' & \implies SP'' \\
\hline
SP & \implies SP''
\end{align*}
\]

**Easy consequence:**

\[
\begin{align*}
SP_0 & \implies SP_1 \implies \cdots \implies SP_n & M & \in Mod[SP_n] \\
\hline
M & \in Mod[SP_0]
\end{align*}
\]
For instance

\[
\text{spec } \text{StringKey} = \text{String and Nat} \\
\text{then opn } \text{hash}: \text{String} \rightarrow \text{Nat} \\
\text{spec } \text{StringKey\_nil} = \text{String and Nat} \\
\text{then opn } \text{hash}: \text{String} \rightarrow \text{Nat} \\
\text{axioms } \text{hash}(\text{nil}) = 0 \\
\text{spec } \text{StringKey\_a\_z} = \text{String and Nat} \\
\text{then opn } \text{hash}: \text{String} \rightarrow \text{Nat} \\
\text{axioms } \text{hash}(\text{nil}) = 0 \\
\text{hash}(a) = 1 \ldots \text{hash}(z) = 26
\]

THEN

\[\text{StringKey} \implies \text{StringKey\_nil} \implies \text{StringKey\_a\_z}\]
...and then, for instance

\[ \text{spec } \text{StringKeyCode} = \text{String } \& \text{ Nat} \]

\[ \text{then opns } \text{hash} : \text{String } \to \text{ Nat} \]

\[ \text{str2nat} : \text{String } \to \text{ Nat} \]

\[ \text{axioms } \text{str2nat}(\text{nil}) = 0 \]

\[ \text{str2nat}(a) = 1 \ldots \text{str2nat}(z) = 26 \]

\[ \text{str2nat}(\text{str}_1 \wedge \text{str}_2) = \text{str2nat}(\text{str}_1) + \text{str2nat}(\text{str}_2) \]

\[ \text{hash}(\text{str}) = \text{str2nat}(\text{str}) \mod 15485857 \]

\[ \text{hide } \text{str2nat} \]

**THEN**

\[
\text{StringKey} \rightsquigarrow \text{StringKey_nil} \rightsquigarrow \text{StringKey_a_z} \rightsquigarrow \text{StringKeyCode}
\]

...and the “code” in \text{StringKeyCode} defines a program/model for \text{StringKey}
Another example

\[
\text{spec } \text{Stack\_of\_String } = \text{String} \\
\text{then sort Stack} \\
\text{opns empty : Stack; } \\
\text{push : String } \times \text{ Stack } \rightarrow \text{ Stack;} \\
\text{top : Stack } \rightarrow \text{ String; } \\
\text{pop : Stack } \rightarrow \text{ Stack} \\
\text{axioms } \text{top(push(str, S))} = \text{str} \\
\text{pop(push(str, S))} = S
\]
spec \textbf{Array\_of\_String} = \textbf{String} and \textbf{Nat} \\
then sort \textit{Array} \\
\textbf{opns} newarr: Array; \\ 
\quad \textit{put}: Array \times Nat \times String \rightarrow Array; \\ 
\quad \textit{get}: Array \times Nat \rightarrow String \\
\textbf{axioms} \text{get(newarr, } i \text{)} = \text{nil} \\ 
\quad \text{get(put(a, } i, \text{str}, i \text{)}, i \text{)} = \text{str} \\ 
\quad i \neq j \implies \text{get(put(a, } j, \text{str}, i \text{)}, i \text{)} = \text{get(a, } i \text{)}
spec \texttt{Stack\_from\_Array} =
\{\texttt{Array\_of\_String} \texttt{then}
  \texttt{sort } \texttt{Stack} = \texttt{Array} \times \texttt{Nat}
  \texttt{opns empty: Stack;}
  \texttt{push: String} \times \texttt{Stack} \rightarrow \texttt{Stack};
  \texttt{top: Stack} \rightarrow \texttt{String};
  \texttt{pop: Stack} \rightarrow \texttt{Stack};
\texttt{axioms empty} = (\texttt{newarr}, 0)
  \texttt{push}((\texttt{str}, (a, i))) = (\texttt{put}(a, \texttt{str}, i + 1), i + 1)
  i > 0 \implies \texttt{top}((a, i)) = \texttt{get}(a, i)
  \texttt{top}((a, 0)) = \texttt{nil}
  i > 0 \implies \texttt{pop}((a, i)) = (\texttt{put}(a, i, \texttt{nil}), i - 1)
  \texttt{pop}((a, 0)) = (a, 0)
\}
\texttt{reveal STRING, Stack, empty, push, top, pop}

THEN

\texttt{Stack\_of\_String} \rightsquigarrow \texttt{Stack\_from\_Array
Extra twist

In practice, some parts will get fixed on the way:

Keep them apart from whatever is really left for implementation:

\[ SP'_0 \rightsquigarrow SP'_1 \rightsquigarrow SP'_2 \rightsquigarrow \cdots \rightsquigarrow \cdots = SP'_n = EMPTY \]
Constructor implementations

\[ \kappa: \text{Alg}(\text{Sig}[SP']) \rightarrow \text{Alg}(\text{Sig}[SP]) \]

Means:

\[ \kappa(\text{Mod}[SP']) \subseteq \text{Mod}[SP] \]

where

\[ \kappa: \text{Alg}(\text{Sig}[SP']) \rightarrow \text{Alg}(\text{Sig}[SP]) \]

is a constructor:

Intuitively: parameterised program (generic module, SML functor)

Semantically: function between model classes

Proof obligation linked with such implementations

putting aside: partiality, persistency…
For instance

\[
\text{constructor } K(A:\text{Sig[Array_of_String]}):\text{Sig[Stack_of_String]}
\]
\[
\text{open } A
\]
\[
\text{type } Stack = \text{Array} \times \text{Nat}
\]
\[
\text{val empty} = (\text{newarr}, 0)
\]
\[
\text{fun push}(\text{str}, (a, i)) = (\text{put}(a, \text{str}, i + 1), i + 1)
\]
\[
\text{fun top}((a, i)) = \text{if } i > 0 \text{ then } \text{get}(a, i) \text{ else } \text{nil}
\]
\[
\text{fun pop}((a, i)) = \text{if } i > 0 \text{ then } (\text{put}(a, i, \text{nil}), i - 1) \text{ else } (a, i)
\]
\[
\text{end}
\]

THEN

\[
\text{Stack_of_String} \overset{K}{\longrightarrow} \text{Array_of_String}
\]

sufficiently true :)

Andrzej Tarlecki: Semantics & Verification - 281 -
Composability revisited

\[
\begin{align*}
SP \xrightarrow{\kappa} SP' & \quad \quad \quad SP' \xrightarrow{\kappa'} SP'' \\
\hline \\
SP \xrightarrow{\kappa';\kappa} SP''
\end{align*}
\]

Easy consequence:

\[
\begin{align*}
SP_0 \xrightarrow{\kappa_1} SP_1 \xrightarrow{\kappa_2} \cdots \xrightarrow{\kappa_n} SP_n = EMPTY \\
\kappa_1(\kappa_2(\cdots \kappa_n(\text{empty}) \cdots)) \in \text{Mod}[SP_0]
\end{align*}
\]

Methodological issues:

- *top-down* vs. *bottom-up* vs. *middle-out* development?
- *modular decomposition* (designing modular structure)
WARNING

Specification structure may change during the development process!

Separate means are necessary
to design the final modular structure
of the program under development
Branching implementation steps

This involves a "linking procedure" (n-argument constructor, parameterised program)

\[ \kappa : \text{Alg}(\text{Sig}[SP_1]) \times \cdots \times \text{Alg}(\text{Sig}[SP_n]) \to \text{Alg}(\text{Sig}[SP]) \]

We require:

\[
\begin{align*}
M_1 & \in \text{Mod}[SP_1] & & \cdots & & M_n & \in \text{Mod}[SP_n] \\
\kappa(M_1, \ldots, M_n) & \in \text{Mod}[SP]
\end{align*}
\]

Proof obligation linked with such design steps
Casl provides an explicit way to write down the design specification branching amounts to:

\[
\text{arch spec } ASP = \begin{array}{l}
\text{units } U_1 : SP_1 \\
\ldots \\
U_n : SP_n \\
\text{result } \kappa(U_1, \ldots, U_n)
\end{array}
\]

Moreover:

- units may be generic (parameterised programs, SML functors), but always are declared with their specifications

- Casl provides a rich collection of combinators to define \(\kappa\) and various additional ways to define units
Instead of conclusions

• Quite a lot of good theory around this;
• Even more bad practice . . .

Ever evading overall goal

Practical methods
for software specification and development
with solid foundations