A Short Introduction to Formal Specifications

Prof. Dr. Hans J. Schneider
Lehrstuhl für Programmiersprachen und Programmiermethodik
Friedrich-Alexander-Universität Erlangen-Nürnberg

30.11.2000

- Methods and Languages
- Algebraic Specification
- Model-Oriented, Operational Specification
- Temporal Logic
Formal Specification

● Categories of formalism:
  – **Informal** : These techniques do not have complete sets of rules to constrain the models that can be created.
  – **Semiformal** : These techniques have defined syntax. Typical instances are diagrammatic techniques.
  – **Formal** : These techniques have rigorously defined syntax and semantics. There is an underlying theoretical model against which a description can be verified.

● Methods and languages

● Model-oriented vs. algebraic methods
Formal Specification

- Categories of formalism
- Methods and languages:
  - **Formal specification methods** enable a software engineer to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation for describing system properties.
  - Using a **formal specification language**, a formal method provides a means for specifying a system so that consistency, completeness, and correctness can be assessed in a systematic fashion.
- Model-oriented vs. algebraic methods
Formal Specification

- Categories of formalism:

- Methods and languages:

- Model-oriented vs. algebraic methods:
  - Model oriented :
    A system is modeled using mathematical entities and describing their modification.
  - Algebraic :
    Object classes are specified in terms of the relationships between the operations defined on the classes.
Formal Specification Languages

- Aspects of a formal specification language:
  - a **syntax** that defines the specific notation with which the specification is represented,
  - a **semantics** that helps to define a “universe of objects” that will be used to describe the system, and
  - a set of **relations** that defines the rules that indicate which objects properly satisfy the specification.

- The **syntactic domain** of a formal specification language is often based on a syntax that is derived from standard set theory notation and predicate calculus.

- The **semantic domain** of a specification language indicates how the language represents system requirements.

- Some examples
Formal Specification Languages

• Aspects of a formal specification language

• Some examples:
  
  – **Larch** is an algebraic, sequential language. A mnemonic notation rather than specialized symbols is used.

  – **Z** is a model oriented, sequential language based on typed set theory. Z allows specifications to be integrated (reused) and has graphical highlighting.

  – **Temporal Logic** is a model-oriented, concurrent language that supports statements allowing to describe properties of sequences of states using temporal operators.
Advantages and Disadvantages of Formal Methods

- The desired properties of a formal specification – lack of ambiguity, consistency, and completeness – are the objectives of all specification methods. However, the use of formal methods results in a much higher likelihood of achieving these ideals.

- Advantages:
  - Enhanced insight into and understanding specifications by revealing contradictions, ambiguities, incompleteness,
  - Help in verification of the specification and their implementations,
  - Possible assistance in moving from requirements specification to their programming implementation.

- Disadvantages
Advantages and Disadvantages of Formal Methods

- Desired properties

- Advantages

- Disadvantages:
  - Application of formal methods needs tool support.
  - Writing (and reading) a formal specification requires some familiarity with mathematical notation.
  - The very formality which makes formal specifications desirable during the later phases makes them an inappropriate tool for communicating with the end user.
  - Formal specifications may not be an ideal tool for exploring and discovering the problem structure.
  - The payoff of the upfront investment is not immediate and difficult to quantify.
Formal Methods Are Recognized to be Necessary

- Electronic Mail of Oct. 6th, 2000:
  Intel Texas Development Center is offering positions in its Formal Verification Group.

- Wall Street Journal of Nov. 21st, 2000:
  Early shipments of Intel’s new Pentium 4 microprocessor contained an incorrect piece of software code, the chipmaker announced yesterday.

(http://www.msnbc.com/493028.asp)
A Short Introduction to Formal Specifications

Prof. Dr. Hans J. Schneider
Lehrstuhl für Programmiersprachen und Programmiermethodik
Friedrich-Alexander-Universität Erlangen-Nürnberg

30.11.2000

- Methods and Languages
- Algebraic Specification
- Model-Oriented, Operational Specification
- Temporal Logic
Algebraic Specification

• An algebraic specification consists of three parts:
  – a syntactic specification,
  – a semantic specification,
  – a restriction specification.

• The syntactic specification lists the names of the type, its operations and the types of the arguments and of the result of the operations.

• The semantic specification consists of a set of algebraic equations which describe the properties of the operations in a representation independent manner.

• The restriction specification states conditions which must be satisfied before the operations can be applied (or after they are completed).
Algebraic Specification of Queues

- The syntactic specification lists the **names** of the type, its operations and the **types** of the arguments and of the result of the operations.

- Example:

  \[
  \begin{align*}
  \text{init\_qu} : & \quad \text{queue} \\
  \text{append} : & \quad \text{queue} \times \text{item} \rightarrow \text{queue} \\
  \text{front} : & \quad \text{queue} \rightarrow \text{item} \\
  \text{remove} : & \quad \text{queue} \rightarrow \text{queue} \\
  \text{is\_empty\_qu} : & \quad \text{queue} \rightarrow \text{bool}
  \end{align*}
  \]
Algebraic Specification of Queues

- The semantic specification consists of a set of algebraic equations which describe the properties of the operations in a representation independent manner.

- Example:

  \[
  \begin{align*}
  \text{is\_empty\_qu}(\text{init\_qu}) &= \text{true} \\
  \text{is\_empty\_qu}(\text{append}(q, i)) &= \text{false} \\
  \text{front}(\text{append}(\text{init\_qu}), i) &= i \\
  \text{front}(\text{append}(\text{append}(q, i), i')) &= \text{front}(\text{append}(q, i)) \\
  \text{remove}(\text{append}(\text{init\_qu}), i) &= \text{init\_qu} \\
  \text{remove}(\text{append}(\text{append}(q, i), i')) &= \text{append}(\text{remove}(\text{append}(q, i)), i')
  \end{align*}
  \]
Algebraic Specification of Queues

• The restriction specification states **conditions** which must be satisfied before the operations can be applied (or after they are completed).

• Example:

\[
\begin{align*}
\text{front}(\text{init}_qu) &= \text{error} \\
\text{remove}(\text{init}_qu) &= \text{error}
\end{align*}
\]
Algebraic Specification: Consistency And Completeness

• Semantics:

\[ \text{is\_empty\_qu(init\_qu)} = \text{true} \]
\[ \text{is\_empty\_qu(append(q, i))} = \text{false} \]
\[ \text{front(append(init\_qu), i)} = i \]
\[ \text{front(append(append(q, i), i'))} = \text{front(append(q, i))} \]
\[ \text{remove(append(init\_qu), i)} = \text{init\_qu} \]
\[ \text{remove(append(append(q, i), i'))} = \text{append(remove(append(q, i)), i')} \]

• Consistency:
If the specifier has understood the problem, (s)he does not write an inconsistent specification.

• Completeness:
The specification is sufficiently complete if each term that ends in a “primitive” sort can be reduced to a term that does not contain any of the new operations.
Parcel Distribution Machine

- Schema of a switch:
Parcel Distribution Machine

- Datatypes

\[
\begin{align*}
\text{dir} &= \{l, r\} \\
\text{req\_dir} &= \text{dir} \cup \{a\}
\end{align*}
\]

\[ l = \text{left}, \quad r = \text{right}, \quad a = \text{arbitrary} \]

- Operations on the switch:

\[
\begin{align*}
\text{init\_sw} &: \text{dir} \rightarrow \text{sw} \\
\text{shift} &: \text{sw} \times \text{req\_dir} \rightarrow \text{sw} \\
\text{entry} &: \text{sw} \rightarrow \text{sw} \\
\text{exit} &: \text{sw} \times \text{dir} \rightarrow \text{sw} \\
\text{is\_empty\_sw} &: \text{sw} \rightarrow \text{bool} \\
\text{act\_dir} &: \text{sw} \rightarrow \text{dir}
\end{align*}
\]
Algebraic Specification of the Switch

- $is\_empty\_sw$
  - returns $true$ if and only if each parcel that has entered the switch has also left it:
    \[
    is\_empty\_sw(init) = true
    \]
    \[
    is\_empty\_sw(entry(s)) = false
    \]
    \[
    is\_empty\_sw(exit(entry(s), d)) = is\_empty(s)
    \]
  - is not affected by the $shift$ operation:
    \[
    is\_empty\_sw(shift(s, d)) = is\_empty\_sw(s)
    \]

- Error indicating combinations:
  - Missing entry signal: $exit(init\_sw(d), d')$
  - Switch does not work: $exit(shift(s, d), d')$ with $d \neq d'$
Algebraic Specification of the Switch

- $act\_dir$
  - is set initially:
    $$act\_dir(init\_sw(d)) = d$$
  - is influenced by shifting the switch:
    $$act\_dir(shift(s,d)) = \begin{cases} act\_dir(s) & \text{if } d = a \\ act\_dir(s) & \text{if } \neg is\_empty\_sw(s) \\ d & \text{otherwise} \end{cases}$$
  - is checked by parcels passing an exit:
    $$act\_dir(exit(s,d)) = d$$
  - is not influenced by parcels entering the switch area:
    $$act\_dir(entry(s)) = act\_dir(s)$$
Problems in Specifying Concurrent Processes

- An operation may affect operations belonging to other objects.
  
  Example: A parcel arriving at the entry point of a distribution station affects
  
  - the state of the arriving queue,
  - the state of the internal queue,
  - the state of the switch.

- An operation can not be executed until a specific condition is satisfied.

- There are timing aspects.
A Short Introduction to Formal Specifications

Prof. Dr. Hans J. Schneider
Lehrstuhl für Programmiersprachen und Programmiermethodik
Friedrich-Alexander-Universität Erlangen-Nürnberg

30.11.2000

• Methods and Languages
• Algebraic Specification
▷ Model-Oriented, Operational Specification
• Temporal Logic
Z: A State-Based Technique

- A Z specification consists of a set of schemas, usually accompanied by corresponding and complementary informal description.

- A schema consists of
  - a name (which may be omitted in some cases),
  - a set of characteristic entities,
  - a (possibly empty) set of constraints on these entities.

- Schemas are used to describe
  - the admissible states of a system,
  - the effect operations of the system have on the states,
  - the relationship between the states of the system at different levels of abstraction.
Z: A State-Based Technique

- **Schema defining admissible states:**

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declarations of entities</td>
</tr>
<tr>
<td>Constraints</td>
</tr>
</tbody>
</table>

- **Schema defining the effect of an operation:**

<table>
<thead>
<tr>
<th>Name of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>States affected by operation</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>Relations between entities</td>
</tr>
</tbody>
</table>
Distribution Machine: Z-Definitions

• Directions:

  Directions
  
  \[ dir, req\_dir : Set \]
  
  \[ dir = \{l, r\} \]
  
  \[ req\_dir = dir \cup \{a\} \]

• Switches:

  Switch
  
  \[ act\_dir : dir \]
  
  \[ is\_empty\_sw : Bool \]
  
  \[ no\_prc : \mathbb{N} \]
  
  \[ is\_empty\_sw = (no\_prc = 0) \]
Distribution Machine: Z-Definitions

- Directions
- Switches
- Distribution stations:

\[
\text{DistStat} \quad \text{sw} : \text{Switch} \\
\text{arr}_\text{prc}, \text{int}_\text{prc} : \text{Queue} \\
\text{succ} : \text{dir} \rightarrow \text{Stat}
\]

- Stations:

\[
\text{Stat} = \text{DistStat} \cup \text{TgtStat}
\]

- \textit{no\_prc may be replaced by length(int\_prc)}. 
Effect of a Parcel Entering a Station

- Announce:

<table>
<thead>
<tr>
<th>Announce</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta DistStat$</td>
</tr>
<tr>
<td>$p?: Parcel$</td>
</tr>
<tr>
<td>$arr_{prc}' = append(arr_{prc}, p?)$</td>
</tr>
<tr>
<td>$int_{prc}' = int_{prc}$</td>
</tr>
<tr>
<td>$lg(int_{prc}) = 0 \Rightarrow sw' = sw.Shift(nxt_dir(p?))$</td>
</tr>
</tbody>
</table>

- Shift

- Entry
Effect of a Parcel Entering a Station

- Announce

- Shift:

<table>
<thead>
<tr>
<th>Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Delta Switch )</td>
</tr>
<tr>
<td>( d? : req_dir )</td>
</tr>
</tbody>
</table>

\[ d? \in dir \land \text{lg}(int\_prc) = 0 \Rightarrow act\_dir' = d? \]
\[ d? \notin dir \lor \text{lg}(int\_prc) \neq 0 \Rightarrow act\_dir' = act\_dir \]
\[ no\_prc' = no\_prc \]

- Entry
Effect of a Parcel Entering a Station

- Announce
- Shift
- Entry:

```
Entry

ΔDistStat
ΔSwitch

int_prce' = append(int_prce, front(arr_prce))
arr_prce' = remove(arr_prce)
no_prce' = no_prce + 1
act_dir' = act_dir
succ' = succ
```
Effect of a Parcel Leaving a Station

- Exit:

\[
\begin{array}{l}
\Delta DistStat \\
\Delta Switch \\
d? : dir \\
\end{array}
\]

- Exit:

\[
\begin{array}{l}
act_{\text{dir}}' = d? \\
no_{\text{prc}}' = no_{\text{prc}} - 1 \\
arr_{\text{prc}}' = arr_{\text{prc}} \\
int_{\text{prc}}' = \text{remove}(int_{\text{prc}}) \\
succ(d?).\text{Announce}(\text{first}(int_{\text{prc}})) \\
lg(arr_{\text{prc}}) \neq 0 \land lg(int_{\text{prc}}) = 0 \Rightarrow \\
sw' = sw.\text{Shift}(nxt_{\text{dir}}(\text{front}(arr_{\text{prc}})))
\end{array}
\]
Z: A Conclusion

- Schemas are used to describe
  - the admissible states of a system,
  - the effect operations of the system have on the states,
  - the relationship between the states of the system at different levels of abstraction.

- **Advanced notations** allow to simplify specifications by modularization:
  - Schema inclusion,
  - Schema disjunction and schema conjunction,
  - Restricting the constraint part to variables that change their value.
A Short Introduction to Formal Specifications

Prof. Dr. Hans J. Schneider
Lehrstuhl für Programmiersprachen und Programmiermethodik
Friedrich-Alexander-Universität Erlangen-Nürnberg

30.11.2000

- Methods and Languages
- Algebraic Specification
- Model-Oriented, Operational Specification

▷ Temporal Logic
Temporal Logic

- We assume a linearly ordered, infinite sequence of states 
  \[ x = (s_0, s_1, s_2, \ldots) \]
  - Each state is associated with a point of time.
  - There is an initial state.
- Notations:
  - \( x^i = (s_i, s_{i+1}, s_{i+2}, \ldots) \), especially: \( x^0 = x \).
  - \( x \models p \) means that \( p \) is true in the first state of sequence \( x \).
- Standard operations:
  \[
  x \models p \land q \iff x \models p \land x \models q
  \]
  \[
  x \models \neg p \iff \neg (x \models p)
  \]
Temporal Logic

- **Basic temporal operations:**
  - **Next time:** $Xp$
    
    **Definition:** $x \models Xp \iff x^1 \models p$
  - **Until:** $p \mathcal{U} q$
    
    **Definition:** $x \models p \mathcal{U} q \iff (\exists j)(x^j \models q \land (\forall k < j)(x^k \models p))$

    **Comment:** There is a future state satisfying $q$, and $p$ is valid in the meantime. (May be that $p$ also holds later on.)
  - Please note that these definitions are recursive: $p$ and $q$ may be composed of temporal operations.

- **Example:**

  \[
  \neg \text{is\_empty\_qu(arr\_prc(s))} \land \text{is\_empty\_qu(int\_prc(s))} \Rightarrow \\
  X \text{act\_dir}(s) = \text{nxt\_dir(front(arr\_prc(s)))}
  \]
Derived Temporal Operations

- **Basic temporal operations:**
  - Next time: $Xp$
  - Until: $p \cup q$

- **Sometimes:**
  \[
  x \models \lozenge p \iff (\exists j)(x^j \models p)
  \]
  \[
  \lozenge p \iff true \cup p
  \]

- **Always:**
  \[
  x \models \square p \iff (\forall j)(x^j \models p)
  \]
  \[
  \square p \iff \neg \lozenge \neg p
  \]
Derived Temporal Operations

- Basic temporal operations:
  - Next time: $Xp$
  - Until: $p \cup q$

- Unless:
  $$x \models p W q \iff (x \models p \cup q \lor x \models □p)$$

- Temporal consequence:
  If $p$ holds sometimes, then at a later point of time, $q$ will hold:
  $$x \models □(p \Rightarrow ◊q)$$

- Infinitely often:
  $$x \models □◊q \iff (∀j)(∃k > j)(q \text{ holds in } s_k)$$
Conclusion

- Importance of mathematical techniques
  - In fields of engineering such as mechanical engineering and electrical engineering, mathematical techniques based on geometry, calculus and complex function theory have been developed and put to invaluable industrial use for more than two hundred years.
  - Mathematical techniques for specification, development and verification of software systems, often termed formal methods, are now coming into use for the construction of real systems, ... in particular for the development of systems whose function or malfunction may seriously affect lives, property and society.

- Testing vs. correctness proof

- Effects of formal methods
Conclusion

- Importance of mathematical techniques

- It is important to distinguish between testing and correctness proofs.
  - Testing is performed by executing the product in a known environment using selected test data.
  - A correctness proof is a formal mathematical verification that the product is correct with respect to a given specification.

- Effects of formal methods
Conclusion

- Importance of mathematical techniques
- Testing vs. correctness proof
- Effects of formal methods:
  - The need to write precisely concerning what is meant by the operations required by a user can cause many points of clarification to arise. The fewer of these left outstanding, the less the scope for error in subsequent stages.
  - It would seem feasible to use formal specifications to direct the testing process into corners of the operational envelope of the system where errors are most likely to lurk.
  - An inevitable consequence of the application of rigorous mathematical analysis to the specification phase of a project is the calling into question of many of the assumptions made thus.