Development and Analysis of Real-Time and Hybrid Systems

Amir Pnueli

Thursdays, 5:00-6:50 PM

Copies of presentations and Lecture Notes will be available at
http://www.cs.nyu.edu/courses/fall04/G22.3033-005/index.htm

Textbook: Parallel Program Design: a Foundation
Course Outline

The course will focus on rigorous (formal) development of systems (programs) from specification to implementation.

According to this approach, we start with a high-level description of the system we wish to construct. Typically, such a description will be declarative and highly non-deterministic. Then, we apply a sequence of transformations, intended to make the implementation more efficient or fit particular architectural constraints. Each of these transformations should be formally verified or validated to guarantee preservation of correctness.

In the course we will consider two different starting points: According to the first approach, all description levels will be presented in the same programming language notation. In that approach we follow as text the book Parallel Program Design: a Foundation by K.M. Chandy and J. Misra.

In the other approaches to be considered in the course, we will consider as a starting point a high-level description presented in a variant of Temporal Logic. The variants we will consider are LTL, TLA, Statecharts, and LSC’s.

Course grades will be determined based on assignments and a term project.
Viewing Programming as a Mathematical Problem

The general mathematical paradigm considers a constraint $C(x)$, e.g.

$$2 < x \leq 10$$

and asks questions such as:

- Does $x = 5$ satisfy the constraint?
- Is the constraint satisfiable by some $x$?
- Find an $x$ which satisfies the constraint, i.e., find a solution.
- Find the best, say maximal, solution $x$ which satisfies $C$.

**Question:** If $x$ is the program, what is $C$?

**Answer:** $C$ is the specification which the program should satisfy.
The Formal Framework for Programming

The subject deals with relations of objects in two description languages on different levels:

- A **programming language** $\mathcal{P}$. Can be compiled and executed on conventional computing systems.

- A **specification language** $\mathcal{S}$. A higher level non-procedural language which offers a natural vehicle for humans to represent requirements and specification of computing tasks.
Questions which can be Asked

Given a verification framework, there are several questions one could ask about relationship between object in these two languages:

- **The Verification Problem**: Given a specification $S \in S$ and a program $P \in P$, check whether they are compatible, i.e. whether $P$ satisfies $S$.

- **The Synthesis Problem**: Given a specification $S \in S$, construct a program $P \in P$ which satisfies the specification.

- **The Analysis Problem**: Given a program $P \in P$, find its corresponding description $S \in S$ (reverse engineering).

- **The Debugging Problem**: Given a specification $S \in S$ and a program $P \in P$ known not to satisfy $S$, find a program $P' \in P$ “close” to $P$, i.e., transform $P$ into $P'$, such that $P'$ satisfies $S$.

- **The Optimization Problem**: Given a specification $S \in S$ and a program $P \in P$ satisfying $S$. Among all programs $P'$ “close” to $P$ and satisfying $S$, find the “best” program (i.e. maximizing some performance metric).
A central notion which appears in all of these questions is that of a program $P \in \mathcal{P}$ satisfying a specification $S \in \mathcal{S}$. For that reason, we should study the verification problem first.

In general, all of these problems are difficult, undecidable, and at best, intractable. However, if $S$ and $\mathcal{P}$ are close enough, they may admit algorithmic solutions. For example, compilation can be viewed as a special case of synthesis.
Synthesis vs. Verification

One of the themes central to these lectures is the strong connection between synthesis and verification.

While synthesis seems to obviate the need for verification, the theory and implementation of synthesis procedures are strongly linked and inspired by a corresponding theory of verification.

In some fortunate cases, we can obtain a synthesis procedure by inverting a deductive verification proof system. We will illustrate this on the case of Hoare logic for sequential programs.
A Simple Programming Language

We introduce a simple programming language to illustrate the idea of synthesis as inversion of verification.

Besides declaration of types and variables, our simple language allows the following statements:

- **Assignment** – $\vec{y} := \vec{E}(\vec{y})$, where $\vec{y}$ is a list of variables, and $\vec{E}$ is a list of type-compatible expressions over the program variables. The statement skip can be introduced as an abbreviation for the assignment $y := y$.

- **Concatenation** – If $S_1$ and $S_2$ are statements, then so is their concatenation $S_1; S_2$.

- **Conditional** – If $S_1$ and $S_2$ are statements and $c$ is a boolean expression, then if $c$ then $S_1$ else $S_2$ is a conditional statement. The one-branch conditional if $c$ then $S$ is an abbreviation for if $c$ then $S$ else skip.

- **While** – If $S$ is a statement and $c$ is a boolean expression, then while $c$ do $S$ is a while statement.
Example: INT-MULTIPLY

Following is an example of a textual program INT-MULTIPLY which multiplies two natural numbers $x_1$ and $x_2$ and returns their result in $z$, using only multiplication and integer divisions by 2.

\[
(u_1, u_2, z) := (x_1, x_2, 0);
\]

while $u_1 \neq 0$ do
  if odd($u_1$) then
    \[
    (u_1, z) := (u_1 - 1, z + u_2);
    \]
    \[
    (u_1, u_2) := (u_1 \div 2, 2u_2)
    \]
Hoare Logic

Following the work of C.A.R. Hoare from 1969, we introduce a Hoare triplet \( \{p\} S \{q\} \) whose intended meaning is that the statement \( S \) is partially correct w.r.t. \( \langle p, q \rangle \). That is,

Every terminating computation of \( S \) which starts at a \( p \)-state ends at a \( q \)-state.

We refer to it as partial correctness because it does not imply termination of \( S \).
The Hoare Proof System

We introduce a list of inference rules as follows:

**Rule ASSGN**

\[
p \rightarrow q[\vec{e} / \vec{y}]
\]

\[
\{p\} \vec{y} := \vec{e} \{\vec{q}\}
\]

where \( q[\vec{e} / \vec{y}] \) is obtained from \( q \) by substituting the expression list \( \vec{e} \) for all occurrences of the variable list \( \vec{y} \), respectively.

For example, in order to prove \( \{x = 1 \land z = 2\} \ y := x + z \ {y = 3}\) \( q \), it is sufficient to prove

\[
x = 1 \land z = 2 \quad \rightarrow \quad x + z = 3
\]

Next, we consider a rule dealing with concatenation:

**Rule CONC**

\[
\{p\} S_1 \{q\}, \quad \{q\} S_2 \{r\}
\]

\[
\{p\} S_1; S_2 \{r\}
\]
A rule for **conditional** statements:

\[
\text{Rule COND} \quad \begin{array}{c}
\{p \land c\} S_1 \{q\}, \quad \{p \land \neg c\} S_2 \{q\} \\
\{p\} [\text{if } c \text{ then } S_1 \text{ else } S_2] \{q\}
\end{array}
\]

A rule for a **while** statement:

\[
\text{Rule WHILE} \quad \begin{array}{c}
\{p \land c\} S \{p\}, \quad p \land \neg c \to r \\
\{p\} [\text{while } c \text{ do } S] \{r\}
\end{array}
\]

Finally, the **consequence** rule:

\[
\text{Rule CONS} \quad \begin{array}{c}
p \to q, \quad r \to u, \quad \{q\} S \{r\} \\
\{p\} S \{u\}
\end{array}
\]


Example: Apply to INT-MULTIPLY

As illustration, we prove \{true\} INT-MULTIPLY \{z = x_1 \cdot x_2\} for the program:

\[\ell_0: (u_1, u_2, z) := (x_1, x_2, 0)\]

\[\ell_1: \text{while } u_1 \neq 0 \text{ do}\]

\[\begin{align*}
\ell_2: & \quad \text{if odd}(u_1) \text{ then } \\
\ell_3: & \quad (u_1, z) := (u_1 - 1, z + u_2) \\
\ell_4: & \quad (u_1, u_2) := (u_1 \div 2, 2u_2)
\end{align*}\]
Annotation

The proof is based on the annotated version of the program:

\[
\begin{align*}
\{\ell_0: \text{true}\} (u_1, u_2, z) & := (x_1, x_2, 0) \\
\{\ell_1: \varphi\} \text{ while } u_1 \neq 0 \text{ do} \\
& \quad \begin{align*}
\{\ell_2: \varphi \land u_1 \neq 0\} \text{ if } \text{odd}(u_1) \text{ then} \\
& \quad \begin{align*}
\{\ell_3: \varphi \land \text{odd}(u_1)\} (u_1, z) & := (u_1 - 1, z + u_2) \\
\{\ell_4: \varphi \land \text{even}(u_1)\} (u_1, u_2) & := (u_1 \div 2, 2u_2)
\end{align*}
\end{align*}
\{\ell_5: z = x_1 \cdot x_2\}
\end{align*}
\]

where

\(\varphi: x_1 \cdot x_2 = z + u_1 \cdot u_2\)
\[ \ell_0: \quad (u_1, u_2, z) := (x_1, x_2, 0) \]

\[ \ell_1: \quad \text{while } u_1 \neq 0 \text{ do } \begin{cases} \ell_2: & \text{if } \text{odd}(u_1) \text{ then } \ell_3: \ (u_1, z) := (u_1 - 1, z + u_2) \\ \ell_4: \quad (u_1, u_2) := (u_1 \div 2, 2u_2) \end{cases} \]

1. \( x_1 \cdot x_2 = z + u_1 \cdot u_2 \land \text{odd}(u_1) \rightarrow x_1 \cdot x_2 = (z + u_2) + (u_1 - 1) \cdot u_2 \land \text{even}(u_1 - 1) \)

2. \( \{x_1 \cdot x_2 = z + u_1 \cdot u_2 \land \text{odd}(u_1)\} \ell_3 \{x_1 \cdot x_2 = z + u_1 \cdot u_2 \land \text{even}(u_1)\} \)

3. \( \varphi \land u_1 \neq 0 \land \text{odd}(u_1) \rightarrow \varphi \land \text{odd}(u_1) \)

4. \( \varphi \land u_1 \neq 0 \land \neg \text{odd}(u_1) \rightarrow \varphi \land \text{even}(u_1) \)

5. \( \{\varphi \land u_1 \neq 0\} \ell_2 \{\varphi \land \text{even}(u_1)\} \)

6. \( x_1 \cdot x_2 = z + u_1 \cdot u_2 \land \text{even}(u_1) \rightarrow x_1 \cdot x_2 = z + (u_1 \div 2) \cdot (2u_2) \)

7. \( \{\varphi \land \text{even}(u_1)\} \ell_4 \{\varphi\} \)

8. \( \{\varphi \land u_1 \neq 0\} \ell_2, \ell_4 \{\varphi\} \)

9. \( \varphi \land u_1 \neq 0 \rightarrow \varphi \)

10. \( x_1 \cdot x_2 = z + u_1 \cdot u_2 \land \neg (u_1 \neq 0) \rightarrow z = x_1 \cdot x_2 \)

11. \( \{\varphi\} \ell_1 \{z = x_1 \cdot x_2\} \)

12. \( \text{true} \rightarrow x_1 \cdot x_2 = 0 + x_1 \cdot x_2 \)

13. \( \{\text{true}\} \ell_0 \{\varphi\} \)

14. \( \{\text{true}\} \text{INT-MULTIPLY } \{z = x_1 \cdot x_2\} \)
Development by Inversion of the Rules

We can view the inference rules as heuristics for solving a constraint with an unknown $S$.

For example, rule \texttt{ASSGN} which is given by:

$$ (p \rightarrow q[\vec{e}/\vec{y}]) \rightarrow \{p\} \vec{y} := \vec{e} \{q\} $$

provides us with sufficient conditions under which we can solve the programming problem $\{p\} S \{q\}$ by taking $S = \vec{y} := \vec{e}$.

The associated development heuristic can be formulated as follows:

**Heuristic 1.** \texttt{ASSGN}

To solve the problem $\{p\} S \{q\}$, find a list of variables $\vec{y}$, and a list of expressions $\vec{e}$, such that $p \rightarrow q[\vec{e}/\vec{y}]$. Then, we can take the statement $\vec{y} := \vec{e}$ for $S$.

For example, in the following we wish to synthesize a program which computes in $y$ the maximum between $x_1$ and $x_2$. We define the assertion

$$ \text{max}(x_1, x_2; y) : x_1 \leq y \land x_2 \leq y \land (y = x_1 \lor y = x_2). $$

Using the \texttt{ASSGN} development heuristic, we can solve the problem $\{x_1 > x_2\} S \{\text{max}(x_1, x_2; y)\}$ by taking $S : y := x_1$. This is justified by

$$ x_1 > x_2 \rightarrow x_1 \leq x_1 \land x_2 \leq x_1 \land (x_1 = x_1 \lor x_1 = x_2). $$
Development Continued

In a similar way, we use the ASSGN heuristic to solve the problem \( \{x_1 \leq x_2\} S \{\max(x_1, x_2; y)\} \) by taking \( S : y := x_2 \). This is justified by

\[
x_1 \leq x_2 \rightarrow x_1 \leq x_2 \land x_2 \leq x_2 \land (x_2 = x_1 \lor x_2 = x_2).
\]

Next we consider a heuristic obtained from inference rule COND.

**Heuristic 2. COND**

To solve the problem \( \{p\} S \{q\} \), find a condition \( c \) and two statements \( S_1, S_2 \), satisfying \( \{p \land c\} S_1 \{q\} \) and \( \{p \land \neg c\} S_2 \{q\} \). Then we can take \( S \) to be if \( c \) then \( S_1 \) else \( S_2 \).

Let us try to solve the programming problem \( \{1\} S \{\max(x_1, x_2; y)\} \) using heuristic COND. As condition \( c \) we choose \( x_1 > x_2 \). It now remains to find statements \( S_1 \) and \( S_2 \), such that \( \{x_1 > x_2\} S_1 \{\max(x_1, x_2; y)\} \) and \( \{x_1 \leq x_2\} S_2 \{\max(x_1, x_2; y)\} \). We have already solved these two sub-problems using heuristic ASSGN. It follows that a possible solution to the programming problem \( \{1\} S \{\max(x_1, x_2; y)\} \) is given by

\[
S : \quad \text{if } x_1 > x_2 \text{ then } y := x_1 \text{ else } y := x_2
\]
A CONC Heuristic

Heuristic 3. CONC

To solve the problem \( \{ p \} S \{ r \} \), find an assertion \( q \) and two statements \( S_1, S_2 \), satisfying \( \{ p \} S_1 \{ q \} \) and \( \{ q \} S_2 \{ r \} \). Then we can take \( S \) to be \( S_1; S_2 \).

We will use this heuristic to solve the problem \( \{ 1 \} S \{ \max(x_1, x_2, x_3; y) \} \). As the intermediate assertion we choose \( q : \max(x_1, x_2; u) \), where \( u \) is a new temporary variable. It only remains to solve the two programming problems:

1. \( \{ 1 \} S_1 \{ \max(x_1, x_2; u) \} \)

2. \( \{ \max(x_1, x_2; u) \} S_2 \{ \max(x_1, x_2, x_3; u) \} \)

The programming problem for \( S_1 \) has been solved before, yielding (with variable renaming) \( S_1 : \text{if } x_1 > x_2 \text{ then } u := x_1 \text{ else } u := x_2 \). The problem for \( S_2 \) can be solved by a similar combination of a single application of COND with \( c : u > x_3 \), and two applications of ASSGN. This would yield a statement \( S_2 \) given by \( S_2 : \text{if } u > x_3 \text{ then } y := u \text{ else } y := x_3 \). It follows that the final solution for \( \{ 1 \} S \{ \max(x_1, x_2, x_3; y) \} \) is given by

\[
S = \begin{cases} 
\text{if } x_1 > x_2 \text{ then } u := x_1 \text{ else } u := x_2 & ; \\
\text{if } u > x_3 \text{ then } y := u \text{ else } y := x_3 
\end{cases}
\]
A WHILE Heuristic

**Heuristic 4.** WHILE

To solve the problem \( \{p\} S \{r\} \), find a condition \( c \), an assertion \( q \) and a statements \( S \), satisfying \( p \rightarrow q, q \land \neg c \rightarrow r \), and \( \{q \land c\} S \{q\} \). Then we can take \( S \) to be \( \text{while } c \text{ do } S \).

We will use this heuristic to solve the problem \( \{1\} S \{y = \sum_{j=1}^{n} a[j]\} \), where \( n \) is a natural variable, and \( a[1..n] \) is an array of integers.

We apply first heuristic ASSGN to derive a statement \( S_1 : (i, y) := (0, 0) \), satisfying \( \{1\} S_1 \{i = 0 \land y = 0\} \). Next, we will use heuristic WHILE to derive a statement \( S_2 \) satisfying \( \{i = 0 \land y = 0\} S_2 \{y = \sum_{j=1}^{n} a[j]\} \). The final solution will be given by \( S_1; S_2 \).

For condition \( c \) we choose \( i < n \). For assertion \( q \), we take \( i \leq n \land y = \sum_{j=1}^{i} a[j] \). It is easy to show that \( q \land \neg c \rightarrow y = \sum_{j=1}^{n} a[j] \). It only remains to find a statement \( S_3 \) satisfying \( \{i < n \land y = \sum_{j=1}^{i} a[j]\} S_3 \{i \leq n \land y = \sum_{j=1}^{i} a[j]\} \). Applying heuristics ASSGN, we can derive \( S_3 : (i, y) := (i + 1, y + a[i + 1]) \). It follows that the final solution is given by

\[
(i, y) := (0, 0); \\
\text{while } i < n \text{ do } (i, y) := (i + 1, y + a[i + 1])
\]