Principles for Value Annotation Languages

Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University

bjorn.lisper@mdh.se

July 8, 2014

WCET Workshop 2014
Introduction

WCET analysis tools require many kinds of *annotations*, like:

- Specifying the environment (hardware, entry points to code, . . .)
- Directing the analysis (context-sensitivity, abstract domain, what kinds of generated information, . . .)
- Specifying value ranges for inputs and program variables
- Specifying flow facts

The two last kinds are both *value annotations* (flow facts = value constraints on IPET execution counters)
State of Practice

Flow fact and value annotation languages are often defined in an ad-hoc manner:

- Tools have their own languages
- Often designed to fit the capabilities of the tool rather than being general
- Sometimes unclear semantics (usually no formal semantics)

Poor interoperability, harder than necessary to specify constraints
An Ecosystem of Embedded Systems Tools

Model

Code generator

User

Code–level Constraints

Code analysis tool

Code–level Constraints

Optimizing compiler

WCET analysis tool

Other tools for analysis & verification
Wish List for a Language for Code-level Value Constraints

Should work over a wide range of code-level tools

Should work over a wide range of “host” languages, on different levels

General yet simple, few but powerful constructs. Simple and clear semantics

Should have:

• Succinct, natural syntax for humans, as well as

• Easily machine-readable form (XML) for tools

Ability to express contexts where constraints are to hold
Contributions

A core assertion language for value constraints, derived from first principles (Floyd-Hoare logic)

Minimal assumptions on the “host” language

Can express flow facts as value constraints on IPET execution counters

Suggestions for user-friendly syntax

Simple, straightforward formal semantics

A theorem about compositionality of assertions
What Could a Core Language be Good For?

A basis for a standardised assertion language for value annotations and flow facts, shared by many tools (ambitious, not so realistic)

A means to specify and understand the semantics of existing annotation formats (more realistic)

- Reduce risk of misinterpretations
- Helpful when defining translators between different annotation languages

More thorough understanding how to design such languages (realistic)

- Will help get future annotation languages right
The Assertion Language of Floyd-Hoare Logic

A starting point: *Floyd-Hoare logic*

\[
\{P\} S \{Q\}
\]

Pre-condition – program – post-condition

\(P\) and \(Q\) express *constraints on states*

States \(\sigma\) map program variables to values (abstraction of memory)

Semantics: if \(P(\sigma)\) holds, and if \(S\) takes \(\sigma\) to \(\sigma'\), then \(Q(\sigma')\) must hold

Example:

\[
\{X = i\} X := X + 1 \{X = i + 1\}
\]
Pre- and post-conditions are expressed in a predicate language on states.

It has:

- program variables (like $X$), which depend on program state,
- *auxiliary* variables (like $i$), which are independent of state,

Auxiliary variables can be used to relate the values of program variables in pre- and post-conditions.
Taking it Further

We take Floyd-Hoare logic as a starting point

But the triples are not suitable. They presume structured (jump-free) code, no good for low-level code.

Solution: make the program point part of the state (add a "PC" variable)

Jumps are now modelled by state transitions that change the PC

Can constrain PC to certain program points in the constraints ("PC = Label")

Can be used to express pre- and post-conditions also on low-level code
A Proposed Syntax

A simple language of arithmetic constraints:

\[
\begin{align*}
a & ::= n \mid i \mid X \mid a_1 \ a_{\text{op}} \ a_2 \\
p & ::= \text{true} \mid \text{false} \mid p_1 \land p_2 \mid p_1 \lor p_2 \mid \neg p \mid a_1 \ a_{\text{op}} \ a_2 \mid \forall i.p \mid \exists i.p \mid PC = L \\
c & ::= p_1 \rightarrow p_2
\end{align*}
\]

\(a\) (arithmetic) expression, \(p\) predicate

\(p_1 \rightarrow p_2\) are assertions (like the triples in Floyd-Hoare logic). These are our value constraints!

Semantics: if \(p_1(\sigma)\) holds, and \(\sigma \rightarrow^* \sigma'\), then \(p_2(\sigma')\) must hold
Relation to the Host Language

Minimal assumptions on the host language:

• Its programs have states $\sigma$, and state transitions $\sigma \rightarrow \sigma'$
• It has program variables $X$. States $\sigma$ map program variables $X$ to (numerical) values $\sigma(X)$
• It has a dedicated program variable $PC$ that holds the current position in the code (a label). Labels can be basically anything that identifies a program point

Examples:

• C: program variables are C variables, labels are C labels or (line number, column number) pairs
• Linked binaries: program variables and labels are addresses
(Assume labels “entry”, “exit” for the entry and exit point of the host program)

• \((PC = entry) \rightarrow (PC = L \implies X < 17)\): for all states reachable from the start of the program, if at label \(L\) then \(X < 17\);

• \((PC = entry \land 1 \leq X \leq 10) \rightarrow (PC = exit \implies Y \leq 100)\): if the program is started with \(1 \leq X \leq 10\) then, at exit, \(Y \leq 100\);

• \((PC = entry) \rightarrow X < 32768\): a global invariant, in all reachable states holds that \(X < 32768\).
Some Syntactic Sugar

Let $@L$ stand for $PC = L$ (common to constrain to a certain label)

Let $p$ stand for $@entry \rightarrow p$ (common to consider all states reachable from the entry point)

Some examples revisited:

- $(@entry \land 1 \leq X \leq 10) \rightarrow (@exit \implies Y \leq 100)$
- $X < 32768$ (understood, for all states reachable from the entry point)
- $@L \implies X < 17$ (ditto)
For any label $L$, a global IPET execution counter $\# L$

Can be used in constraints expressing flow facts:

- $\@\text{exit} \implies \# L < 100$: a simple capacity constraint;
- $\@\text{exit} \implies \# L = 99$: an exact capacity constraint;
- $\@\text{exit} \implies \# L_1 + \# L_2 \leq 1$: a mutual exclusivity constraint;
- $(\@\text{entry} \land 1 \leq X \leq 10) \implies (\@\text{exit} \implies \# L \leq 100)$: a capacity constraint under the condition that the value of $X$ lies in the range $[1 \ldots 10]$ at entry;
- $(\@\text{entry} \land X = n) \implies (\@\text{exit} \implies \# L \leq 2 \cdot n + 1)$: a parametric constraint relating the number of executions of $L$ to the value of $X$ at entry;
The state could contain time (represented, say, by program variable $T$)

An example of a real-time constraint. Assume that $L, L'$ are labels in a loop with loop counter $I$. Then

$$(@L \land T = t \land I = i) \rightarrow (@L' \land I = i \implies T - t \leq 7)$$

expresses that for each iteration, $L'$ is reached at most 7 time units after $L$

Uses auxiliary variables $i, t$ for “old” values of $I, T$ (in pre-condition). Could use “$X.old$” to refer to value of $X$ in pre-condition. Example becomes

$$@L \rightarrow (@L' \land I = I.old \implies T - T.old \leq 7)$$
Semantics

The language can be given a formal semantics

Completely standard, I will not bring it up here

Important to have to make the notation well-defined

Also makes it possible to prove certain laws

**Theorem 1** (compositionality of assertions):

\[ p_1 \rightarrow p_2 \land p_2 \rightarrow p_3 \implies p_1 \rightarrow p_3. \]
We can define call-strings as sequences of labels that are call sites for functions.

Let $S$ be a call-string. $p \rightarrow p'$ through $S$ means that if $p(\sigma)$ holds, and $\sigma'$ can be reached from $\sigma$ through a sequence of transitions visiting the labels in $S$, then $p'(\sigma')$ must hold.

Can be used to “qualify” assertions to hold only for certain contexts.

Theorem 1 can be extended to context-sensitive assertions:

**Theorem 2:**

$$p_1 \rightarrow p_2 \text{ through } S \land p_2 \rightarrow p_3 \text{ through } S' \implies p_1 \rightarrow p_3 \text{ through } S \cdot S'.$$
Conclusions

A simple core language for value constraints
Like Floyd-Hoare logic, but not restricted to structured (jump-free) languages
Minimal assumptions on the host language
Can express very general value constraints, including general flow facts
Formal semantics, exact meaning, no room for misinterpretations
Straightforward to extend to context-sensitive constraints
Not restricted per se to WCET analysis tools, any code level tool could potentially use it