Objective: implement two formal analysis tools for the K semantic framework

- **Semantics-based compiler** – to achieve much faster formal execution of programs, by statically summarizing the behavior of the program, using the semantics of the programming language
- **Bounded model checker** – to execute a program systematically, on all paths, up to a given depth/bound, in order to detect errors or “unexpected” behaviors

Extends previous project, which focused mainly on semantics-based execution in order to detect undefined behaviors – that technology is now being commercialized by RV Inc.
K Framework

Formal Language Definition (Syntax and Semantics) Java, JavaScript, C, C++ ...

Parser
Test-case generation
Deductive program verifier
Symbolic execution
Fast concrete interpreter

C, C++ semantics being developed at RV Inc.

Semantic debugger
Interpreter
Semantic compiler
Model checker
**K Framework + Strategy Language**

- Parser
- Interpreter
- Semantic debugger
- Semantic compiler
- Test-case generation
- Deductive program verifier
- Symbolic execution
- Model checker
- Formal Language Definition (Syntax and Semantics) Java, JavaScript, C, C++ ...
- Strategy Language
- Fast concrete interpreter

C, C++ semantics being developed at RV Inc.
Semantics-Based Compilation (SBC)

Problem: Given a program $P$ in language $L$, generate language $L'$ such that execution of program $P$ in languages $L$ and $L'$ is equivalent.

Goal: Language $L'$ should be “as simple as possible”, only capturing exactly the dynamics of $L$ necessary to execute program $P$. 
SBC Algorithm

One step of compilation:

- Check if current state has already been compiled or execution is finished (subsumed? or stuck?)
- Symbolically execute program until abstraction point is reached (step until cut-point?)
- Summarize previous statements into one rewrite rule (begin-rule ... end-rule)
- Abstract state to ensure completeness

```
if (subsumed? or stuck?)
then skip ;
else begin-rule ;
  step ;
  step until cut-point? ;
end-rule ;
abstract ;
```

Each line of code is executed once (often much faster than full execution of program).
Output of the algorithm is a semantics-preserving 𝕄 𝕈 definition (can be used in further analysis).
// start
int b, n, x;
b = 1; n = 1; x = 0;

// outer
while (b <= 27) {
    n = b;

    // inner
    while (2 <= n) {
        if (n <= ((n / 2) * 2)) {
            n = n / 2;
        } else {
            n = (3 * n) + 1;
        }
        x = x + 1;
    }
    b = b + 1;
}
// end
These numbers were gathered using concrete execution in the strategy harness
Normal $\mathbb{K}$ execution is uniformly $\sim 10x$ faster
Bounded Model Checking (BMC)

Problem: Given a program $P$, a language $L$, a property $\varphi$, and a bound $N$, output true if $\varphi$ is not violated within $N$ steps, and output false and the violating trace otherwise.
BMC Algorithm

Bounded Model Checking:

- record appends current state to the accumulated trace
- step takes one (user-defined) step of symbolic execution
- bmc-result checks whether the property $P$ holds at the current state, and outputs the current trace if it does not

This algorithm is useful for undefinedness checking, checking common programming errors (division by zero, off by one), etc.

```
record;
while N P (step ; record) ;
bmc-result P ;
```
```c
// div-good.imp
int x, y;
x = 3; y = 100;

while (0 < x) {
    y = y / x;
    x = x - 1;
}

// div-bad.imp
int x, y;
x = 3; y = 100;

while (0 <= x) {
    y = y / x;
    x = x - 1;
}
```

```bash
$> bmc 100 "not div-zero-error?" div-good.imp
#bmc-result #true in 32 steps ...

div-good.imp does not exhibit a division by zero error.

div-bad.imp does exhibit a division by zero error at 35 steps. When the bound is decreased to 34, the property is not violated but the program is about to divide by zero.

$> bmc 100 "not div-zero-error?" div-bad.imp
#bmc-result #false in 35 steps ...

$> bmc 34 "not div-zero-error?" div-bad.imp
<s> #bmc-result #true in 34 steps ... </s>
<k> ( 16 / 0 ) ... </k>
```
Accelerating BMC with SBC

Perform BMC on output of SBC instead of original program
- Avoid many steps of symbolic execution
- Roughly 10x speedup on Collatz program
- Higher density of straight-line code will yield more speedup

Since SBC produces a correct $\mathbb{K}$ definition, it can be used to speedup many other processes such as concrete execution and program verification.
Future Extensions

Run analysis tools using faster OCaml backend developed at RV Inc
- Ocaml backend about 10,000x faster than the current Java/Scala backend, but does not support symbolic execution yet

Develop more language-independent analysis tools using this strategy-based technique
- Runtime verification of programs, combined with SBC and BMC
- Static analysis of programs, using symbolic execution
- Full program verification using deductive reasoning and invariant inference, plus SBC and BMC

Demonstrate scalability on larger languages
- C, C++, Java, JavaScript: we have K semantics for them, but in v3.6; not yet updated to work with c4.0
- Add optimizations to BMC
- Transfer formal analysis technology in commercial-grade tools at RV Inc.