Program analysis parameterized by the semantics in Maude

A. Riesco (joint work with I. M. Asăvoae and M. Asăvoae)

Universidad Complutense de Madrid, Madrid, Spain

Workshop on Logic, Algebra and Category Theory, LAC 2018
Melbourne, Australia
February 16, 2018

Motivation

- Logical frameworks allow us to specify and analyze different logics and inference systems.
- In particular, we can represent the semantics of programming languages.
- We have a logical framework $\mathcal{L}$ where we specify the semantics $\mathcal{S}$ of programming languages.
- Then, we have programs p written for these programming languages.
- Is it possible to define generic analyses $\mathcal{A}$ so $\mathcal{A}(\mathcal{S})$ is a tool for programs p defined for $\mathcal{S}$?

Motivation

- It is possible but not straightforward.
- (Meta)Semantics of the framework vs. semantics defined for the programming language in the framework.
- We have a (declarative) debugger for logical framework that asks questions and locates bugs in the specification:

```
x = 0;
i = 1;
while i < 3 {
   x = x + 1; // It should be x = x + i
   i = i + 1;
}
```

- It locates an error in the semantics of loops.

Motivation

- However, it is useful (and usual) to use the tools for $\mathcal{L}$ to analyze programs p.
- For example, for looking for counterexamples with model checking.
- Finding a counterexample indicates that the program fails (assuming the semantics are correct).
- Interpreting the counterexample is difficult, because it does not refer to $\mathcal{S}$.

Motivation

- Particular syntax/semantics vs. another particular syntax/semantics.
- If we implement the debugger to locate problems in C-like functions (in a naïve way)...
- We would need to implement the same for Java-like functions.
- Both debuggers would be quite similar but the sorts would and particular pieces of syntax would different.

Motivation

- We choose *Rewriting Logic* as logical framework.
- We use *Maude* as implementation of rewriting logic.
- Our generic analysis will be *slicing*.
- For presentation purposes we will focus in a particular imperative language with conditional statements, loops, i/o buffer, and function calls.

Motivation

- Many researchers have specified the complete semantics of programming languages in Maude (and $\mathbb{K}$).
- Useful for theorem proving and model checking.
- Many specific tools for proving “particular” results on different languages.
- These particular results are applied to features such as classes/objects, threads, or message-passing.
- They also included refined theorem provers and modifications for improving model checking counterexamples.
- Generic analysis tools would be useful for all of the already defined languages.

Outline of the talk

1. Preliminaries
   - Rewriting Logic
   - Semantics in Rewriting Logic
   - Slicing
2. Parameterized analysis
   - Features synthesis
   - Slicing algorithm
3. Prototype
4. Conclusions and ongoing work

Membership equational logic

- Rewriting logic is parameterized by an underlying equational theory.
- In our case this logic is *membership equational logic* (MEL).
- We denote by $\Sigma \equiv (K, \Sigma_K, S)$ a *signature* where:
  - K is a set of *kinds*,
  - $\Sigma_K = \{\Sigma_{k_1 \ldots k_n, k}\}_{(k_1 \ldots k_n, k) \in K^* \times K}$ is a many-kinded signature, and
  - $S = \{S_k\}_{k \in K}$ is a pairwise disjoint K-kinded family of sets of *sorts*.
- Intuitively, terms with a kind but without a sort represent error elements.
- $T_{\Sigma,k}$ and $T_{\Sigma,k}(X)$ denote the set of ground Σ-terms and of Σ-terms over variables in X.

Membership equational logic

- The atomic formulas of MEL are:
  - *Equations* $t = t'$.
  - *Membership axioms* $t : s$.
- *Sentences* are Horn clauses of the form $(\forall X)\, A_0 \Leftarrow A_1 \wedge \cdots \wedge A_n$, with A_i either an equation or a membership axiom.
- A *specification* is a pair (Σ, E), where E is a set of sentences in MEL over the signature Σ.
- A MEL specification (Σ, E) has an initial model $\mathcal{T}_{\Sigma/E}$.
- Our MEL specifications are assumed to satisfy some executability requirements, so their equations can be oriented from left to right, $t \to t'$.

Membership equational logic

- Maude functional modules, with syntax `fmod ... endfm`, are executable MEL specifications.
- In a functional module we can declare:
  - `Sorts`.
  - `Subsort` relations between sorts.
  - Operators (`op`) for building values of these sorts, giving the arity and coarity, and which may have attributes such as `assoc` and `comm`.
  - Memberships (`mb`) asserting that a term has a sort.
  - Equations (`eq`) identifying terms.

Membership equational logic

- Both membership axioms and equations can be conditional (`cmb` and `ceq`).
- Conditions can also be *matching equations* $t := t'$, which are solved by matching, thus instantiating new variables in t.
- Maude does automatic kind inference.
- Kinds correspond to the connected components of the subsort relation.

Membership equational logic

```
fmod OLIST is
 pr NAT .

 sorts List OList .
 subsort Nat < OList < List .

 op mt : -> OList [ctor] .
 op __ : List List -> List [ctor assoc id: mt] .

 vars N N' : Nat .
 var  L : List .

 cmb N N' L : OList
  if N <= N' /\
     N' L : OList .

 op |_| : List -> Nat .
 eq | mt | = 0 .
 eq | N L | = s(| L |) .
endfm
```

Rewriting Logic

- Rewriting logic extends equational logic by introducing the notion of *rewrites*.
- A rewriting logic specification, or *rewrite theory*, has the form $\mathcal{R} = (\Sigma, E, R)$, where:
  - (Σ, E) is an equational specification.
  - R is a set of *rules*.
- A rule q in R has the general conditional form

$$q : \ (\forall X)\ e \Rightarrow e' \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w'_k$$

Rewriting Logic

- Maude system modules, with syntax `mod ... endm`, are executable rewrite theories.
- They can contain rules (`rl`) and conditional rules (`crl`).
- Rewrite conditions instantiate variables in the righthand side.
- The strategy followed by Maude is to compute the normal form of a term w.r.t. the equations before applying a rule.
- This strategy works when the rules are *coherent* with respect to the equations.

Rewriting Logic

```
mod SHUFFLE is
 pr LIST .

 vars N N' : Nat .
 var  L : List .

 rl [shuffle] : N  L N'
 =>             N' L N .
endm
```

Rewriting Logic

- Rewriting logic is reflective.
- There is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that:
  - We can represent any finitely presented rewrite theory $\mathcal{R}$ and any terms t, t' in $\mathcal{R}$ as terms $\overline{\mathcal{R}}$ and $\overline{t}, \overline{t'}$ in $\mathcal{U}$.
  - Then we have the following equivalence

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

- Since $\mathcal{U}$ is representable in itself, we get a *reflective tower*.
- In Maude reflection is accesible via the predefined `META-LEVEL` Maude module.
- It allows specifiers to handle and reason about terms that represent specifications described in Maude itself.

Semantics in Maude

- Maude is a logical framework.
- Other languages are easily defined in Maude.
- Given a programming language defined by a grammar such as

$$
\begin{array}{lcl}
\mathit{Ins} & \rightarrow & \texttt{skip} \\
 & | & \mathit{Var} \ \texttt{:=} \ \mathit{Exp} \\
 & | & \texttt{while} \ \mathit{Cond} \ \texttt{do} \ \mathit{IL} \ \texttt{od} \\
 & & \ldots \\
\mathit{Cond} & \rightarrow & \texttt{true} \mid \texttt{false} \mid \texttt{not} \ \mathit{Cond} \mid \mathit{Cond} \ \texttt{and} \ \mathit{Cond} \mid \ldots \\
\mathit{IL} & \rightarrow & \varepsilon \\
 & | & \mathit{Ins} \ \texttt{;} \ \mathit{IL}
\end{array}
$$

Semantics in Maude

$$
\begin{array}{lcl}
\mathit{Ins} & \rightarrow & \texttt{skip} \\
 & \mid & \mathit{Var}\ \texttt{:=}\ \mathit{Exp} \\
 & \mid & \texttt{while}\ \mathit{Cond}\ \texttt{do}\ \mathit{IL}\ \texttt{od} \\
 & & \ldots \\
\mathit{Cond} & \rightarrow & \texttt{true} \mid \texttt{false} \mid \texttt{not}\ \mathit{Cond} \mid \mathit{Cond}\ \texttt{and}\ \mathit{Cond} \mid \ldots \\
\mathit{IL} & \rightarrow & \varepsilon \\
 & \mid & \mathit{Ins}\ \texttt{;}\ \mathit{IL}
\end{array}
$$

- Syntactic categories are defined as datatypes (sorts in Maude).

```
sorts Ins Cond IL Exp Var ... .
```

- Grammar rules are stated as operators:

```
op skip : -> Ins [ctor] .
op _:=_ : Var Exp -> Ins [ctor] .
op while_do_od : Cond IL -> Ins [ctor] .

subsort Ins < IL .
op mtIns : -> IL [ctor] .
op _;_ : IL IL -> IL [ctor assoc id: mtIns] .
```

Semantics in Maude

- Semantics is defined by means of inference rules:

$$\frac{\langle E, st, b, fs\rangle \Rightarrow v}{\langle X := E, st, b, fs\rangle \Rightarrow \langle \texttt{skip}, st[v/X], b, fs\rangle}\ \mathrm{Asg}$$

- Inference rules are defined by conditional rewrite rules:

```
crl [Asg] : < X := e, st, b, fs >
 =>         < skip, st[v / X], b, fs >
 if < e, st, b, fs > => v .
```

- Auxiliary functions are defined by means of equations.

Semantics in Maude

- In particular we need to specify abstract, implicit notions as the memory.

```
sorts Cell Memory .
subsort Cell < Memory .

op _|->_ : Var Value -> Cell [ctor] .
op noCell : -> Memory [ctor] .
op _;_ : Memory Memory -> Memory [ctor assoc comm id: noCell] .
```

- And functions for look-up and update:

```
op _[_]   : Memory Var -> Value .
op _[_/_] : Memory Value Var -> Memory .
...
```

Semantics in Maude

$$\frac{\langle C, st, b, fs\rangle \Rightarrow \texttt{true} \quad \langle IL\ ;\ \texttt{while}\ C\ \texttt{do}\ IL\ \texttt{od}, st, b, fs\rangle \Rightarrow \langle \texttt{skip}, st', b', fs\rangle}{\langle \texttt{while}\ C\ \texttt{do}\ IL\ \texttt{od}, st, b, fs\rangle \Rightarrow \langle \texttt{skip}, st', b', fs\rangle}\ \mathrm{W}_1$$

```
crl [W1] : < while C do IL od, st, b, fs >
  =>       < skip, st', b', fs >
 if < C, st, b, fs > => True /\
    < IL ; while C do IL od, st, b, fs > => < skip, st', b', fs > .
```

$$\frac{\langle I, st, b, fs\rangle \Rightarrow \langle \texttt{skip}, st', b', fs\rangle \quad \langle IL, st', b', fs\rangle \Rightarrow \langle \texttt{skip}, st'', b'', fs\rangle}{\langle I\ ;\ IL, st, b, fs\rangle \Rightarrow \langle \texttt{skip}, st'', b'', fs\rangle}\ \mathrm{Comp}$$

```
crl [Comp] : < I ; IL, st, b, fs >
 =>          < skip, st'', b'', fs >
 if < I, st, b, fs > => < skip, st', b', fs > /\
    < IL , st', b', fs > => < skip, st'', b'', fs > .
```

Semantics in Maude

- Maude is executable, so once the semantics are specified you can write programs in your syntax and execute them.
- We use the command `rew`:

```
Maude> (rew < x := 0 ;
              y := 1 ;
              while x < 3 do
                z := x + y ;
                x := x + 1
              od, noCell, noRWB, noFuns > .)
Result: < skip, x |-> 3 ; y |-> 1 ; z |-> 3, noRWB, noFuns >
```

Slicing

- In particular, we focus on *slicing*.
- Slicing is an analysis method that takes a program and a *slicing criterion* and produces a *program slice*.
- This slicing criterion is usually a program point *pc* and a set of program variables.
- The slice consists of those instructions potentially modifying the elements in the slicing criterion.

Slicing

- That is, the slicing criterion is updated by adding information information from those instructions producing side effects.
- For example, given `x = y;`, if `x` is in the slicing criterion we would add `y`.
- We also propagate information via parameter passing.
- If we have a call `f(a, b)`, with `a` in the slicing criterion, and the definition `f(x,y)`, then we add `x`.

Slicing

- Assume we are interested on variable z.

```
function Main (){
 sum := 0;
 Local i;
 i := 1;
 while i < 11 do
  Call A (sum, i)
}
```

```
function A (x, y) {
 Call Add (x, y);
 Call Inc (y)
}

function Add (a, b) {
 a := a + b
}
```

```
function Inc (z) {
 Local i;
 i := 1;
 Call Add (z, i)
}
```

Slicing

- Assume we are interested on variable z.
- The sliced body of Add is included in the slice of Inc.
- It also indicates that i must be added to the slice.

```
function Main (){
 sum := 0;
 Local i;
 i := 1;
 while i < 11 do
  Call A (sum, i)
}
```

```
function A (x, y) {
 Call Add (x, y);
 Call Inc (y)
}

function Add (a, b) {
 a := a + b
}
```

```
function Inc (z) {
 Local i;
 i := 1;
 Call Add (z, i)
}
```

Slicing

- Assume we are interested on variable z.
- The return statement of procedure Inc is paired with the call to Inc, in the body of A so y becomes relevant for the slice.

```
function Main (){
 sum := 0;
 Local i;
 i := 1;
 while i < 11 do
  Call A (sum, i)
}
```

```
function A (x, y) {
 Call Add (x, y);
 Call Inc (y)
}

function Add (a, b) {
 a := a + b
}
```

```
function Inc (z) {
 Local i;
 i := 1;
 Call Add (z, i)
}
```

Slicing

- Assume we are interested on variable z.
- However, in this case we distinguish the call to Add from A.
- Since b does not depend on a, x is not added to the slice.

```
function Main (){
 sum := 0;
 Local i;
 i := 1;
 while i < 11 do
  Call A (sum, i)
}
```

```
function A (x, y) {
 Call Add (x, y);
 Call Inc (y)
}

function Add (a, b) {
 a := a + b
}
```

```
function Inc (z) {
 Local i;
 i := 1;
 Call Add (z, i)
}
```

Slicing

- Assume we are interested on variable z.
- Hence, in this case sum is not included in the slice.

```
function Main (){
 sum := 0;
 Local i;
 i := 1;
 while i < 11 do
  Call A (sum, i)
}
```

```
function A (x, y) {
 Call Add (x, y);
 Call Inc (y)
}

function Add (a, b) {
 a := a + b
}
```

```
function Inc (z) {
 Local i;
 i := 1;
 Call Add (z, i)
}
```

Slicing

- Assume we are interested on variable z.
- It would be the same for other languages like C or Python.

```
void Main (){
 sum = 0;
 int i;
 i = 1;
 while (i < 11) {
  A (sum, i);
 }
}
```

```
void A (int x, int y) {
  Add (x, y);
  Inc (y);
}

void Add (int a, int b) {
 a = a + b;
}
```

```
void Inc (int z) {
 int i;
 i = 1;
 Add (z, i);
}
```

Slicing

- Assume we are interested on variable z.
- It would be the same for other languages like C or Python.

```
def Main () :
 global sum
 sum = 0
 i = 1
 while (i < 11) :
  A (sum, i)
```

```
def A (x, y) :
  Add (x, y)
  Inc (y)

def Add (a, b) :
 a = a + b
```

```
def Inc (z) :
 i = 1
 Add (z, i)
```

Slicing in Maude

- There is a slicer available for Maude specifications.
- It slices Maude programs using Maude variables.
- Hence, it can debug the semantics.
- However, it cannot slice the programs written in the specified language, it is just a term!
- We show how to implement a slicer parameterized by the semantics defined in Maude.
- It computes the slices of programs whose semantics has been specified in Maude.
- We have implemented a tool called *Chisel* that implements the ideas presented here.

Parameterized analysis

- Given the standard slicing algorithm for imperative languages, it is “easy” to apply it to the programs written for the semantics given we know:
  - The sort for the elements in the slicing criterion

Parameterized analysis

- Given the standard slicing algorithm for imperative languages, it is “easy” to apply it to the programs written for the semantics given we know:
  - The sorts for the elements in the slicing criterion, the instructions

Parameterized analysis

- Given the standard slicing algorithm for imperative languages, it is “easy” to apply it to the programs written for the semantics given we know:
  - The sorts for the elements in the slicing criterion, the instructions, and the memory.

Parameterized analysis

- Given the standard slicing algorithm for imperative languages, it is “easy” to apply it to the programs written for the semantics given we know:
  - The sorts for the elements in the slicing criterion, the instructions, and the memory.
  - The instructions generating side-effects (e.g. assignments).

Parameterized analysis

- Given the standard slicing algorithm for imperative languages, it is "easy" to apply it to the programs written for the semantics given we know:
  - The sorts for the elements in the slicing criterion, the instructions, and the memory.
  - The instructions generating side-effects (e.g. assignments).
  - The data-flow information (e.g. `a := b` generates $\overset{\curvearrowleft}{\texttt{a b}}$).

Parameterized analysis

- Given the standard slicing algorithm for imperative languages, it is "easy" to apply it to the programs written for the semantics given we know:
  - The sorts for the elements in the slicing criterion, the instructions, and the memory.
  - The instructions generating side-effects (e.g. assignments).
  - The data-flow information (e.g. `a := b` generates $\overset{\curvearrowleft}{a\ b}$).
  - The instructions in charge of context-updates (e.g. function calls).

Parameterized analysis

- Given the standard slicing algorithm for imperative languages, it is “easy” to apply it to the programs written for the semantics given we know:
  - The sorts for the elements in the slicing criterion, the instructions, and the memory.
  - The instructions generating side-effects (e.g. assignments).
  - The data-flow information (e.g. `a := b` generates $\overset{\curvearrowleft}{\texttt{a b}}$).
  - The instructions in charge of context-updates (e.g. function calls).
  - The parameter-passing policy.

Parameterized analysis

- Given the standard slicing algorithm for imperative languages, it is “easy” to apply it to the programs written for the semantics given we know:
  - The sorts for the elements in the slicing criterion, the instructions, and the memory.
  - The instructions generating side-effects (e.g. assignments).
  - The data-flow information (e.g. `a := b` generates $\overset{\curvearrowleft}{\texttt{a}\,\texttt{b}}$).
  - The instructions in charge of context-updates (e.g. function calls).
  - The parameter-passing policy.
  - The memory policies about references (e.g. pointers).

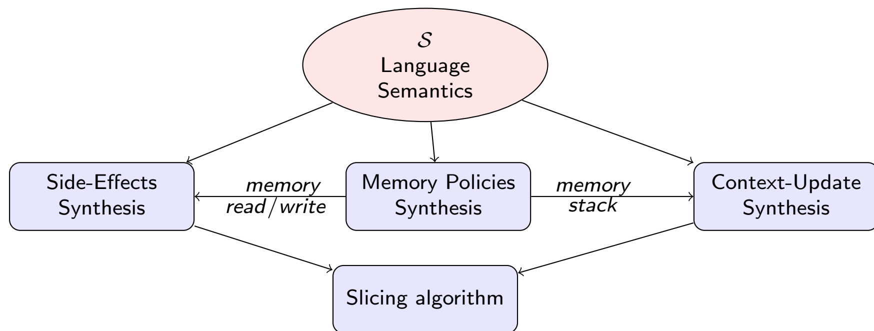
Parameterized analysis

- Given the standard slicing algorithm for imperative languages, it is “easy” to apply it to the programs written for the semantics given we know:
  - The for the elements in the slicing criterion
  - The instructions generating side-effects (e.g. assignments).
  - The data-flow information (e.g. `a := b` generates $\overset{\curvearrowleft}{\texttt{a b}}$).
  - The instructions in charge of context-updates (e.g. function calls).
  - The parameter-passing policy.
  - The memory policies about references (e.g. pointers).
  - How to deal with parallelism.

Parameterized analysis

- We should not force the user to provide all this information.
- We only support *call-by-value* parameter passing.
- We *do not* support pointers.
- We *do not* parallelism.
- We infer the sort for the elements in the slicing criterion from the initial command.
- We infer the rest of information (over-approximations).

Chisel



Chisel: Memory policies synthesis

- We start the analysis by classifying the sorts.
- We distinguish between them depending on how terms of these sorts are used in rules:
  - Terms being read and manipulated. Values obtained from these terms are used to access and manipulate terms of other sorts: *instructions*.
  - Terms used to store and access information: *memory, i/o buffer, registers, etc.*
  - Terms containing instructions and only read: *function definitions*.
  - Auxiliary sorts, which cannot be classified in any of the categories above: *program counter*.

Chisel: Memory policies synthesis

- Given the sorts for the memory, we can:
  - Find the functions accessing and modifying it.
  - Find its constructors.
- Functions accessing and modifying the memory show the data-flow information.
- The constructors allow us to detect *stack structures* used when changing context.

Chisel: Memory policies synthesis

```
op _[_]   : Memory Var -> Value .
op _[_/_] : Memory Value Var -> Memory .
```

- We know from the first analysis that `Memory` is the sort for the memory.
- Operators with `Memory` as coarity update it.
- Operators with it in the arity (and not in the coarity) access it.

Chisel: Data-flow synthesis

- This stage receives the set of rules modifying the memory, as well as the sets of functions modifying and accessing the memory, respectively.
- When facing a rule that modifies the memory, we take the instruction and slice the rule w.r.t. the variables for each element to see whether it accesses or modifies the memory.
- For example, in `X := e` we find that `e` accesses the memory.
- The value thus obtained is used to update the memory for the position `X`.
- This stage returns a set of elements of the form $\mathtt{x} \curvearrowleft \mathtt{s}$, indicating that the variable `x` is modified by the variables in `s`.

Chisel: Data-flow synthesis

```
crl [Asg] : < X := e, st, b, fs > => < skip, st[v / X], b, fs >
 if < e, st, b, fs > => v .
```

- From the previous analysis we know that the X is the value being modified and v is the new value.
- From Maude slicing, we see that v depends on e and st.
- From the first analysis we know that X and e are part of the instruction.
- So e is modifying X.

Chisel: Context-update synthesis

- This stage receives the stack constructors and identifies the rules for context-updates.
- However, some languages (e.g. assembly) do not rely on building a stack of environments when a function is called.
- In this case we use a test-case generator to analyze how the execution sequence varies and identify those rules that are in charge of these variations.
- This stage is under development, our random test-case generator does not provide good results so we need to rely in an external tool.

Program slicing as term slicing: algorithm

- The algorithm has two phases:
  - The initialization of the data structures.
  - The loop implementing the fixpoint.
- The initialization part computes a hyper-tree following the structure of the program.
- The fixpoint loop discovers the call graph in an on-demand fashion using the context-update set, which directs the fixpoint iteration.

Program slicing as term slicing: algorithm

- This fixpoint iteration is composed of three parts:
  - Updating the slice by using the called functions.
  - Updating the slice by using the calling functions.
  - Updating the slice by applying intraprocedural with the new variables.
- The algorithm terminates because there exists a finite set of function skeleton subterms, a finite set of data flow graphs, a finite set of edges in the call graph for each function, and any loop in the call graph is solved based on the data flow graph ordering.
- It produces a valid slice because it exhaustively saturates the slicing criterion.
- It might not be minimal, although it produces better results than many standard techniques.

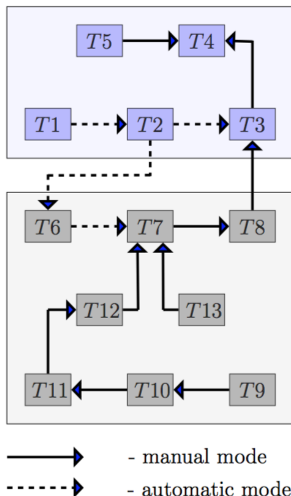
Prototype

- We test our proposed method for slicing on a set of benchmarks addressing embedded and real-time applications.
- We have compiled them into C and MIPS (assembly).
- The C code has been tested with different implementations.
- Disclaimer: we have the semantics for a small subset of C, not for the whole language.

https://github.com/ariesco/chisel

Prototype

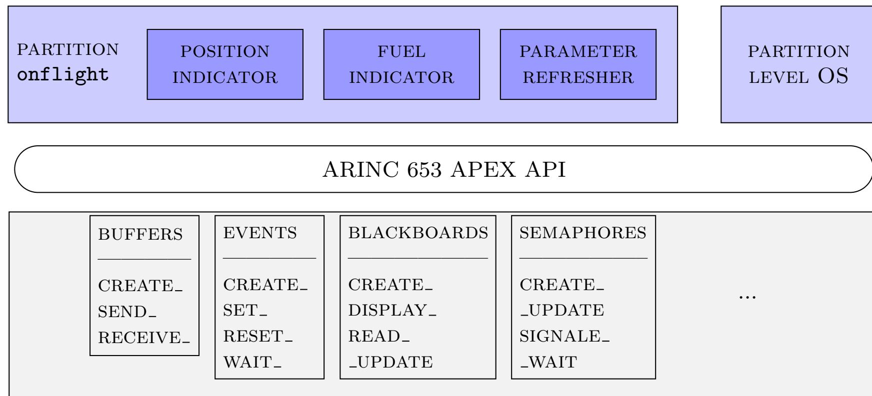
Tasks : fbw and autopilot		
T1	-	receive_radio_commands
T2	-	send_data_to_autopilot
T3	-	receive_data_from_autopilot
T4	-	transmit_servos
T5	-	check_failsafe
T6	-	manage_radio_commands
T7	-	control_stabilization
T8	-	send_data_to_fbw
T9	-	receive_gps_data
T10	-	control_navigation
T11	-	control_altitude
T12	-	control_climb
T13	-	manage_reporting



Prototype

Name	# Funs	# Calls	LOC (WhileFun)	red (%) (WhileFun)	LOC (MIPS)	red (%) (MIPS)
scheduler_fbw	14	18	103	72.8 %	396	44.4 %
periodic_auto	21	80	225	73.3 %	779	36.3 %
fly_by_wire	**41**	**110**	**638**	**91.1 %**	**1913**	**41 %**
T1	10	26	119	76.5 %	534	36.2 %
T2	9	9	59	69.5 %	329	44.4 %
T3	9	24	82	76.5 %	501	43.6 %
T4	9	14	50	61.5 %	235	34.5 %
T5	7	22	66	67 %	453	51 %
autopilot	**95**	**214**	**1384**	**92 %**	**5639**	**41.5 %**
T6	36	71	306	77.2 %	1329	54 %
T7	9	13	57	70 %	426	42 %
T8	7	15	54	69.2 %	219	38 %
T9	15	30	87	75 %	617	36.5 %
T10	18	27	102	71.1 %	1002	42.2 %
T11	3	2	15	63.4 %	90	70.6 %
T12	4	3	49	66.2 %	363	50 %
T13	37	93	240	79.7 %	1535	42 %

Prototype



Prototype

Functionality	Vars	# LOC (WhileFun)	# LOC (MIPS)	red (%) (WhileFun)	red (%) (MIPS)
GLOBAL_PROCESS_MANAGER	28	377	7020	90.7 %	33.6 %
GLOBAL_SEMAPHORE_MANAGER	6	114	6949	97.2 %	34.2 %
GLOBAL_EVENT_MANAGER	6	95	6944	97.6 %	34.3 %
GLOBAL_BLACKBOARD_MANAGER	6	90	6942	97.7 %	34.3 %
GLOBAL_BUFFER_MANAGER	6	147	6945	96.4 %	34.3 %
GLOBAL_SAMPLINGPORT_MANAGER	7	87	6971	97.8 %	34 %
CREATE_COMPONENTS	17	366	6945	91 %	34.3 %
PRESENT_SIGNALS	35	372	7054	90.9 %	33.2 %
INITIALIZATION_SIGNALS	2	78	6937	98 %	34.3 %
NORMAL_SIGNALS	1	83	6933	97.9 %	34.4 %
POSITION_INDICATOR	10	276	6931	93.2 %	34.4 %
FUEL_INDICATOR	12	221	6949	94.5 %	34.2 %
PARAMETER_REFRESHER	12	233	6949	94.3 %	34.2 %

Prototype

- What is happening?

Prototype

- What is happening?
- In WhileF we have `x = 0;`
- `x` is a variable and `0` a value.

Prototype

- What is happening?
- In WhileF we have `x = 0;`
- `x` is a variable and `0` a value.
- In MIPS we have `mv Ri RZ`.
- `Ri` is a register and `RZ` a register as well.
- `RZ`, which behaves as a constant, is added to the slice.

Conclusions

- We have presented a generic algorithm for interprocedural slicing based on results of meta-level analysis of the language semantics.
- In summary, the slicing prerequisites are:
  - Context-update rules and side-effect sorts.
  - Call-by-value parameter passing.
  - No parallelism.
- The actual program slicing computation is done through term slicing and is meant to set the aforementioned set of prerequisites.
- Specific problems for some languages worsen the performance.

Conclusions

- We have worked with many different languages and different Maude specifications for them.
- We have also used Maude for other parameterized analyses.
- We have a generic declarative debugger for big-step and small-step semantics
- We also transform the counterexample obtained from the model checker to correspond to the semantics of the particular programming language.

Ongoing work

- We are interested on studying how the technique works for languages with pointers and parallelism.
- We are also interested on call-by-name parameter passing.
- From the prototype point of view, we also plan to investigate how to infer the parameter passing pattern.
- We intend to improve our test-case generator to detect context-updates.
- We want to slice the counterexample obtained from the model checker.
- Finally, we aim to develop the method for language semantics defined in Maude but also in $\mathbb{K}$.