Formal Refinement Verification in Metropolis

Douglas Densmore
University of California, Berkeley
densmore@eecs.berkeley.edu

***Abstract*—When building system level models of computer systems, often it is advantageous to begin with an abstract model for the purposes of simulation or initial behavior verification. However, once this abstract model has served its purpose, a more detailed, less abstract model should replace it to give not only a more "realistic" performance estimation but also bring the model closer to a possible physical implementation for reasons such as synthesis. However, in order to make this refinement, one must be assured that the refined models conform to the behavior of their abstract counterparts. This paper introduces the beginning of a framework for the Metropolis Design Environment which begins to verify this refinement.**

I. INTRODUCTION

As computer systems become more and more complex, abstraction is used to simplify many portions of the design into selected behaviors needed at that particular point in a design. These behaviors reflect the necessary information to proceed with the design while not hindering the designer with tedious details or overly complex interactions. As the design progresses however, those details are added back into the design as the level of abstraction decreases. The abstract components of a design must be replaced with their more detailed counterparts. Key in this transformation is that the replacing components do not introduce behaviors that were not present in the abstract system. It is the process of ensuring that the set of possible refined behaviors are a subset of the abstract behaviors that is refinement verification.

The Metropolis Design environment [5] is particularly in need of such a verification procedure. Metropolis is a framework based around a *meta-model* enforcing a semantics involving the separation of communication, computation, and coordination activities. From this meta-model there are paths to simulation, analysis, and synthesis tools. Metropolis has as part of its syntax and semantics the notion of successive refinement. Figure 1 shows briefly how refinement is used currently.

```
//In Metropolis Netlist

/*introduce to the netlist(this),
and object for refinement(ref_obj)*/
refine(ref_obj,this);

/*redefine the connects
so that the refinement input
and outputs map to the abstracts ports*/
refineconnect(this,src_connect(ref_obj,out),
     port(ref_obj,out),abs_out);
refineconnect(this,src_connect(ref_obj,in),
     port(ref_obj,in),abs_in);
```

Fig. 1. Current Metro Refinement

Previous work in the area of model checking these types of systems has included [12] which examines how to represent process network structures, [4] and [9] which examine how to use Co-Design Finite State Machines for Functional Equivalence. [11] creates a way to look at analysis of C programs for source-to-source transformations and was useful in thinking how to represent the structure of programming languages. Each of these sources provides some insight and background into how to think about verifying structures in programs. These helped identify that the key to this project was to have a methodology in place with a compact data structure which was flexible, allowed for abstraction, and which can be evaluated efficiently.

A. YAPI and TTL Models

One clear motivating example for the need for refinement in Metropolis involves the YAPI (Y-Chart API) [6] and the TTL (Task Transaction Level) Libraries. These are both process network based FIFO libraries. In Metropolis, the YAPI library has unbounded FIFO-like elements while the TTL library attempts to be a refined version with *boundedfifo*, *yapi2TTL*, *TTL2yapi*, and *rdwrthreshold* elements. The boundedfifo simply is the storage mechanism now with a fixed size. The rdwrthreshold element acts as the coordination for access to this element. Finally, yapi2TTL and TTL2yapi are used for the refinement interface in the refined netlist similar to figure 1.

During the use of these elements in a multi-media application exercise in Metropolis, several bugs were discovered. This drew attention to the fact that refinement checking is a crucial element as the design process becomes more complex and specifications are adhered to in an *ad hoc* manner.

This paper goes on to demonstrate a particular methodology on much smaller models then either the TTL or YAPI library elements. The flow is not currently mature enough for a model as complex as TTL or YAPI but the overall flow and goals will not change when moved to this issue. Ultimately an example file provided with the Metropolis distribution and a test file were used as will be shown in section VI. However, it is this background which gave rise to this project.

B. Purpose of paper

The purpose of this paper is to detail the initial tool chain and methodology for verifying that one Metropolis Model is a refinement of the other. Section II formally introduces what refinement verification is. Section II-A states the problem in terms of the Metropolis Meta-Model. Section III describes the process of writing a Metropolis backend to create the structure on which refinement verification would be based. Sections IV and V discuss two important tools in this investigation. Finally sections VI, VII, and VIII detail the results, conclusions, and future work respectively.

II. Refinement

The notion of refinement verification for this paper stems from that of Hierarchical Verification in [2]. This project begins with Hierarchical Verification as the foundation and modifies it slightly for this project. [2] uses the term *implementation* whereas we will use the term *refinement* to mean the same thing. We will define the problem as follows:

A model is generically defined as an object which can generate a set of finite sequences of behaviors, *B*. One of these possible finite sequences, *B*, is considered a trace, $\overline{a}$. Given a model *X* and a model *Y*, *X* refines the model *Y*, denoted $X \preceq^{Ref} Y$ if given a trace $\overline{a}$ of *X* then the projection $\overline{a}$[ObsY] is a trace of *Y*. A trace, $\overline{a}$ is considered a sequenced set of observable values for a finite execution of the module. A projection of a trace, $\overline{a}$[ObsY], is the trace produced on Module *Y* for the execution which created $\overline{a}$ over the Observable variables of *Y*. The two modules *X* and *Y* are *trace equivalent*, $X \simeq^{Ref} Y$, if $X \preceq^{Ref} Y$ and $Y \preceq^{Ref} X$.

The answer to the refinement problem (*X*,*Y*) is YES if X refines Y and otherwise NO.

A. Refinement Verification for Metropolis

Building on the generic refinement definition given in section II, we have defined the refinement problem in Metropolis with a discussion on the syntactic conditions and trace definition.

1) Syntatic Conditions: [2] frames the refinement conditions in terms of the *reactive modules* [1] syntax and puts requirements on their variable structures for each model to be compared. For the similar syntactic conditions for Metropolis models, given $X \preceq^{Ref} Y$, are that $Y_{inputs} \subseteq X_{inputs}$ and $Y_{outputs} \subseteq X_{outputs}$. Essentially this simply requires that *X* have all of *Y*'s inputs and outputs (if not more). This could be viewed as simply a naming issue if you require the same order and number of corresponding inputs and outputs for each model.

2) Trace Definition: As mentioned previously, a trace, $\overline{a}$ is considered a sequenced set of observable values for a finite execution of the module. In the case of Metropolis, the key observable values that we are concerned with are *function calls to media*. This paper will refer to a trace consisting of function calls to media as a $Trace_M$, where the "M" stands for "Metropolis". Due to the semantics of Metropolis, processes must communicate strictly via media. Ultimately the behavior of a process can be characterized by the sequence by which it makes these calls. Syntactically this results in an interface call attached to a particular port. Figure 2 shows the process and medium interaction. This shows what is observable to the rest of the system is this process' use of the media interface.

In order to characterize the Metropolis trace, $TraceM$, the key structure to be obtained from the model is the control flow graph concerning the ways in which these sets of observable events can occur. Once this structure is created *State Equivalence* concepts such as *Bisimilarity* and *Similarity* [2] could be used to determine refinement. A $Trace_M$ can be obtained by traversing this structure described next.

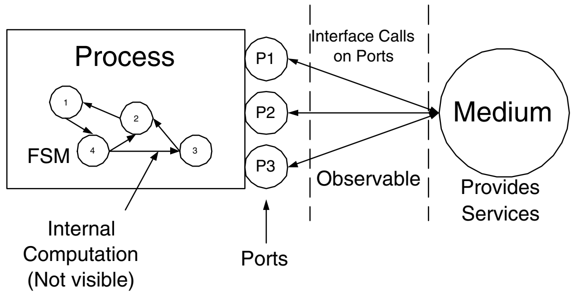

Fig. 2. Function Calls to Media as Observable Variables

B. Control Flow Automata in Metropolis

The key structure in this investigation is the Control Flow Automaton (CFA) representation of a Metropolis Model. Metropolis has an *Action Automata* specification underlying it [5] but this provides much more information than is required and its structure is not suited to use in our refinement scenario. A CFA is defined as a very much like [8]. It is a tuple <*Q*, q_0, **X**, Op, $\rightarrow$>.

Q is a finite set of control locations. These will be determined by the Metropolis model structure. q_0 is the initial control location, **X** is a set of variables, and Op are operations which denote (1) function calls to media (2) basic blocks of instructions starting (3) basic block of instructions ending. This ending and beginning notion is taking from the *Action Automata* semantics. A basic block is taken in the traditional sense, meaning a section of code in which there is no conditional execution which could result in a different execution sequence. A basic block simply could be viewed abstractly as a function call. It is for this reason that the start and end are denoted. This way, the CFA could be augmented with the body of the function call, inserted inside the beginning and end portions.

An edge (q, Op, q') is a member of a finite set of edges and the transition relationship, $\rightarrow$, is defined as (Q $\times$ Op $\times$ Q). A edge makes a transition based on the Op present, q $\rightarrow^{Op}$ q'.

Ideally an CFA is created which represents the model and corresponding automata are created which represent the state of variables in the automata. For example a model may have a loop which is checks the value of a particular variable. The CFA would have a variable, $v \in$ **X**, which has an automata which can be queried as to the value of that variable to determine what edges can be transitioned. For the purposes of this project, these automata are not formally defined nor are they automatically generated. Figure 4 shows one possible representation that could be used to show the incrementing of an integer with a functional range of 0 to 2.

Figures 3 and 4 demonstrate a code snippet and the resulting Metropolis CFA as defined in this paper.

Once a CFA is defined, a $Trace_M$ is nonempty word $\overline{a}_{1...n}$ over the alphabet of Q control locations such that $a_i \rightarrow a_{i+1}$ for all 1 $\leq$ i $\leq$ n.

Naturally the potential for a CFA to be quite large is a concern. As you will see in the description of the backend it is bounded by the nodes in the Abstract Syntax Tree created by Metropolis compilation which could be very large. However this can be reduced further by heuristic grouping of nodes to create control locations as will be shown.

III. CFA BACKEND

The Metropolis Design Environment is designed around the concept of a *meta-model* as mentioned previously. This allows for the initial model to be decomposed into an intermediate representation and then fed to a number of different tools called *backends*. This is demonstrated roughly in the structure shown in figure 5. As you can see the model is parsed into an *Abstract Syntax Tree* (AST) and that AST is interpreted by the backends to generate another representation with semantics for another tool while maintaining some relationship to the original model. The creation of a backend to generate a CFA as described earlier was the primary work of this paper.

The CFA backend traverses the AST and identifies the *nodes* of the AST. It is composed of two files:

- CFABackend.java
- CFACodegenVisitor.java

CFABackend.java is what is called when the backend is *invoked* and actually writes to various files the results of the *visitor functions*. The file CFACodegenVisitor.java actually contains the visitor functions. The visitor functions traverse the AST and determine what should happen at each type of node. There are over **160 different node types** that can make up an AST. It is in these functions that the CFA structure is determined. In particular this is true when visiting what this project introduces as *Grouping Node Types*. Each AST node generates its own *location* structure. Groups of these belong to a *group location structure*. Each group location structure each contains **exactly** one node which is a member of the *Grouping Node Types*. These sets of group location structures with one unique node of the Grouping Node Types are what constitute a control location, *Q*, in the CFA. All of this is stored in an internal list structure which can be traversed itself. It is this heuristic grouping which prevents the size of the CFA from being *O(AST Nodes in Model)* and rather ***O(Grouping Node Types in Model)*** which is substantially smaller in practice. In order to have this reduction the *Grouping Node Types* are currently defined as:

- Structure Nodes - these include ProcessDeclNode, CompileUnitNode

```
//sample code snippet
process example {
  port Read port1;
  port Write port2;
void thread(){
   int x = 0
      while (x<2){
        port1.callRead();
        x++;}
      port2.callWrite();
 }
}
```

Fig. 3. Metro Code Example

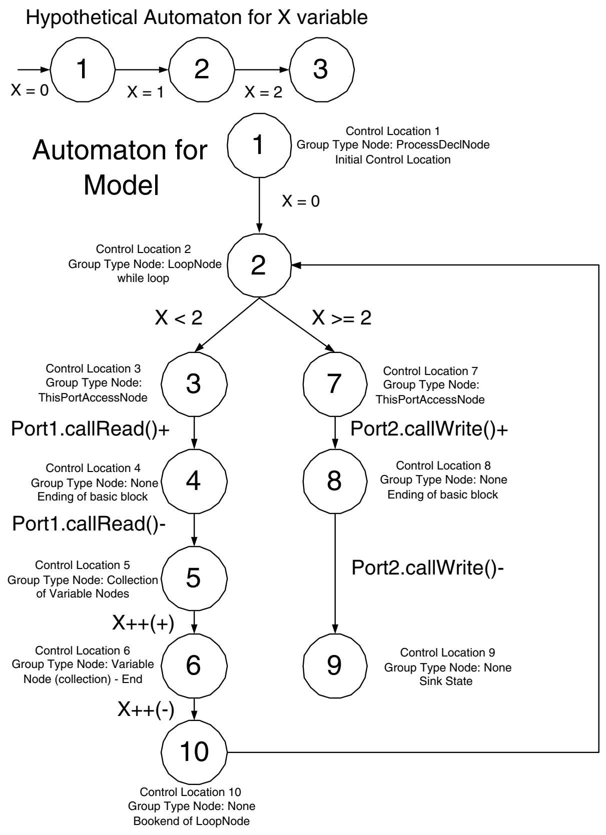
Hypothetical Automaton for X variable
1
2
3
X = 0
X = 1
X = 2
Automaton for Model
Control Location 1 Group Type Node: ProcessDeclNode Initial Control Location
X = 0
Control Location 2 Group Type Node: LoopNode while loop
X < 2
X >= 2
Control Location 3 Group Type Node: ThisPortAccessNode
Control Location 7 Group Type Node: ThisPortAccessNode
Port1.callRead()+
Port2.callWrite()+
Control Location 4 Group Type Node: None Ending of basic block
Control Location 8 Group Type Node: None Ending of basic block
Port1.callRead()-
Port2.callWrite()-
Control Location 5 Group Type Node: Collection of Variable Nodes
X++(+)
Control Location 6 Group Type Node: Variable Node (collection) - End
Control Location 9 Group Type Node: None Sink State
X++(-)
Control Location 10 Group Type Node: None Bookend of LoopNode

Fig. 4. Resulting CFA for Code Example

- Control Nodes - these include AwaitStatementNode, AwaitGuardNode, LoopNode
- Variable Nodes - these include ThisPortAccessNode

Also, the CFA internal structure is able to be created in one pass through the Metropolis Model code so the running time it is ***O(Nodes Traversed with Visitor Functions)*** where visitor functions $\leq$ AST nodes types in code.

A. Visual Representation

The first and most trivial result of the CFA backend is a simple visual representation as shown in figure 6

This is simply for debugging purposes and allows the user to see not only what the structure of the CFA is but also examine what individual nodes compose a *control location*. This information can be used to redefine what a *Grouping Node Type* is and also see the effects of different heuristic choices for grouping. The *Group* field is an integer identification of what group this is. In turn this corresponds to a control location, *Q* in the CFA. The *Parents* field is a set of integers which define which groups are the parents of this group. *Types* is a set of integers which are associated which each node to identify it (as defined by the AST node types). The *Inputs* field denotes what input variables must be required to transition from this group. The *Outputs* field denotes which output variables will be present (go “high”) when you transition from this node. *Misc* is used to hold such information as if arithmetic nodes are visited (i.e. a PlusNode denoting a possible incrementing of a variable) or other information used to build the CFA. *Names* is simply a list of Strings which indicates what types nodes make up this group location (corresponding to the type field; easier to read). And finally the *Cond Code* field indicates which type of conditional node was visited for the group (i.e. LoopNodes, AwaitStatementNodes, etc) and is internally defined to identify the branching structure of the CFA. The "arrow" like symbols are used where there are multiple children. This can be produced in one pass of the internal list structure of the CFA or ***O(CFA Control Locations)***.

B. Finite State Machine Representation

The second more functional result of the CFA backend, is that it produces a Finite State Machine representation of the CFA. The inputs to the finite state machine represent information provided by other automata to the CFA model (such as the variable automata) and the outputs are the function calls to media. This is formatted as a KISS representation. An example of KISS is shown in figure 7.

This format was chosen for two reasons: (1) It is easily produced from the internal list structure (2) it can be read by various tools such as SIS [7]. SIS in turn can produce other formats such as BLIF, PLA, EQN, etc. Of particular interest is BLIF whose close relative EXLIF can be read by FORTE [10] as will be described in section IV

Once the initial data structure is created by the backend the algorithm to create a KISS file is as shown in figure 8

The running time of this algorithm is ***O(2 * CFA Structure Groups)***. Essentially you have to traverse the structure once to create the lists of inputs and outputs. Then you must traverse it again to actually generate the KISS file based on that information. Each line of KISS requires that you examine the input and output lists completely to see if they contain input or output at that location.

C. Reactive Module Representation

The third and final result of the *CFA backend* is a *"reactive module"* [1] file. This is a modeling language for describing the behavior of hardware and software systems. This is produced as an additional benefit of the backend for three reasons: (1) It is very inexpensive to create a *reactive module* which models an *FSM*. (2) It allows for non-deterministic behavior which is not allowed by *KISS* models provided to *SIS*. (3) It can be read by tools such as *MOCHA* [3]. *MOCHA* allows a rich set of model checking algorithms to be run on the *CFA* model that are useful both for refinement and other verification tasks.

The first point for making this representation was that it was inexpensive to do from the FSM representation. Figure 9 gives the algorithm to do so.

This can be done in one pass of the KISS file. The variable declaration initializations for the module are simply from the *KISS* input (.i), output (.o), and state (.s) declarations. The *init* command is simply another listing of the variables. The largest part of the file, the *update* commands, correspond one-to-one with each line in the *KISS* body. The running time of this is naturally ***O(KISS file body)***.

The second reason for using this representation, non-determinism, is inherent in the fact that multiple guards may be *true*. Also inherent is that the union of all guard commands does not have to equal the entire space of the inputs (i.e. it can be partially specified). Naturally, KISS currently has deterministic behavior so it will result in a reactive module with deterministic behavior. However, there is nothing preventing a reactive module from being produced from a KISS file which would not run in SIS. A CFA could be produced that has non-deterministic behavior simply with a modification to the backend.

The third and final reason, *MOCHA*, will be discussed in section V.

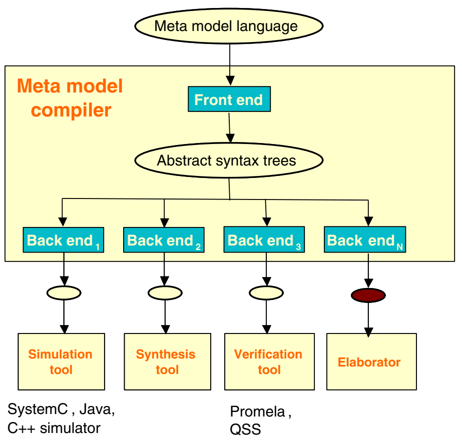

Fig. 5. Metropolis Compiler Structure

```
Group: 3
Parents: 2
Types: 12
Inputs: in1
Outputs: #can be blank
Misc: #can be blank
Names: LoopNode
Cond Codes: 1
      |     |
      V     V
```

Fig. 6. CFA Visual Representation

```
#KISS File
.i 3 #input count
.o 4 #output count
.s 2 #state count
.p 2 #next state equations
#inputs current_state next_state outputs
010 s1 s2 0101
000 s2 s1 1010
.e
```

Fig. 7. KISS Representation

IV. FORTE

Prior to the integration of *Reactive Modules* into the *CFA Backend*, the project was targeting a tool called FORTE. FORTE [10] is a tool provided by Intel Corporation which is a collection of several tools. These are *Functional Language*

```
Input: CFA Data Structure, D
Output: KISS File
//create unique inputs list (1)
∀ group locations $i \in D$ {
  for each input $j \in i$ {
   if $j \notin$ unique input list IL, add j}}
//same procedure as (1) for outputs using a different list
//Create the declarations section
Simply output the size of the input and output lists for the .i and .o
portions. The .s is the size of the structure D and .p is how many lines
you process when done making the body.
//Create the Body
//input portion of the kiss line (2)
∀ group locations, $i \in D$ {
  ∀ elements, $e \in IL$ {
   if $e \in i$ write 1
  else write 0}
Output current_group and child_group
//same procedure for output values as (2) for kiss line
}
```

Fig. 8. Algorithm for KISS construction

```
Input: KISS File
Output: Reactive Module File
create new Module <filename>
//lists of the external, interface, and private variables
 for ∀ $i \in$ FSM Output
   new interface variable
 for ∀ $i \in$ FSM Input
   new external variable
 for ∀ $i \in$ FSM State
   new private variable
create new atom cfa
 controls <each interface variable, each private variable>
 reads <each external variable, each private variable>
 init
<all interface and private variables = false except first state variable>
  update
<for each line of the KISS representation the guard is the appropriate
input = true and that current state = true, the result is the next state
variable = true and the appropriate outputs = true>
```

Fig. 9. Algorithm for Reactive Module construction

```
//sis commands
read_kiss <filename>
state_minimize
state_assign <nova> or <jedi>
source script.rugged
write_blif <filename>
```

BLIF to EXLIF Manual Edits
- Remove start_kiss, end_kiss, and kiss code embedded in file
- Remove external don't care section (.exdc)
- Add to the .latch definitions a clk signal and the type of flop it is (rising, falling)
- Remove the .latch_order and .code portions

Fig. 10. SIS Commands and EXLIF requirements

```
Input: Two EXLIF Models, A and R with State Space $\Sigma_A$ and $\Sigma_R$
Output: Answer to the Refinement Question (R, A)
Let $q_A \in \Sigma_A$ and $q_R \in \Sigma_R$
Given a set of states, the set S, $S^C$ is $(\Sigma_A \cup \Sigma_R) \setminus S$
Let $\vec{x}$ be a vector of inputs common to both A and R
Let $\vec{y}_A(q_A, \vec{x})$ be a vector of outputs for A given the state $q_A$ and the inputs $\vec{x}$
Let $\vec{y}_R(q_R, \vec{x})$ be a vector of outputs for R given the state $q_R$ and the inputs $\vec{x}$
Let $E_n$ be a set of sets of states reachable in n input sequences
Let $\sigma$ be a set of sets $\{(q_A, q_R) \mid q_A \in \Sigma_A, q_R \in \Sigma_R\}$
Let $Tr(q_A, \vec{x}, q_A')$ = true if there is a transition from $q_A$ to $q_A'$ under input $\vec{x}$
Let $pre(\sigma) = \{(q_A, q_R) \mid \exists \vec{x} : Tr(q_A, \vec{x}, q_A') \cap Tr(q_R, \vec{x}, q_R') \cap (q_A', q_R') \in \sigma\}$
//Start of Algorithm
$E_0 = \emptyset$
$E_1(q_A, q_R) = \forall \vec{x}, \vec{y}_A(q_A, \vec{x}) \bigodot \vec{y}_R(q_R, \vec{x})$
k = 0
do {
  k = k + 1
  $E_{k+1}(q_A, q_R) = E_k(q_A, q_R) \setminus preE_k^C(q_A, q_R)$
until $(E_{k+1} = E_k)$
if $(\forall q_R \in \Sigma_R, \exists q_A$ such that $(q_A, q_R) \in E_k)$
 Return YES
else
  Return NO
```

Fig. 11. Algorithm for FORTE Refinement Check

(FL), *Symbolic Trajectory Evaluation* (STE), *FSM Logic Data model*, and some circuit drawing tools. FORTE works on circuit descriptions of models. This is was a major factor in influencing the decision to reduce the CFA into a FSM representation.

Once a model had been created as a KISS file, that KISS file was given to the SIS tool. This was used to create a BLIF file with the script in figure 10. This representation was very similar to the EXLIF file format used by FORTE. Some simple modifications allowed this to be converted to EXLIF and in turn read by FORTE also shown in figure 10. These manual edits could be worked into a Perl script in the future.

Once the models are converted to EXLIF files FORTE can begin to process them for refinement. The algorithm is in figure 11.

The running time for such an algorithm is approximately ***O(m • n)*** where *n* is the number of states and *m* is the transitions. This algorithm and corresponding code was not created as part of this project but supplied by Intel.

V. MOCHA

Since the CFA Backend produces a *Reactive Module* MOCHA can be used to do refinement checking. However, this requires some manual preparation of the file produced by the backend. [3] describes refinement as a *trace inclusion* problem. This amounts to:

1) For every initial state, *s* of *X*, the projection of *s* to the variables of *Y* is an initial state of *Y*.
2) For every reachable state of *s* of *X*, if *X* has a transition from *s* to *t* then *Y* has a matching transition.

The search can be done symbolically or emuneratively with MOCHA. In the case that the test fails it generates a counterexample of a trace on *X* which is not a trace of *Y*. This may be computationally complex. Therefore some restrictions are placed on the modules, to verify $X \preceq^{Ref} Y$.

1) The module *Y* has no private variables
2) Every interface variable of *Y* is an interface variable of *X*.
3) Every external variable of *Y* is an external variable of *X*.

Recalling our requirements for refinement, the 2nd and 3rd conditions are already met. However, a module created with the CFA Backend will have private variables representing states. The solution for this is to create a *Witness Module*, *W*. This is a module whose interface variables are the private variables of *Y*. Also, *W* should not contain any of the external variables of *X*. In turn, a module, *Y'*, will be created with those private variables declared as interface variables. Once this is the case then $X||W \preceq^{Ref} Y'$ as shown in [1]. The procedure is naturally:

1) Create *Y'* from *Y* by changing private variables to interface.
2) Define a *Witness Module*, *W*, whose interface variables are the private variables of *Y* but exclude the observable variables of *X*.
3) Check $X||W \preceq^{Ref} Y'$ with MOCHA

Since this not automatic this is a potential bottleneck in the flow, since the creation of a *Witness Module* requires creativity on the part of the user. In addition the parallel composition is also manual.

VI. Results

In order to demonstrate a *proof of concept* for our methodology, we assembled the previously described components into a complete flow as shown in figure 12.

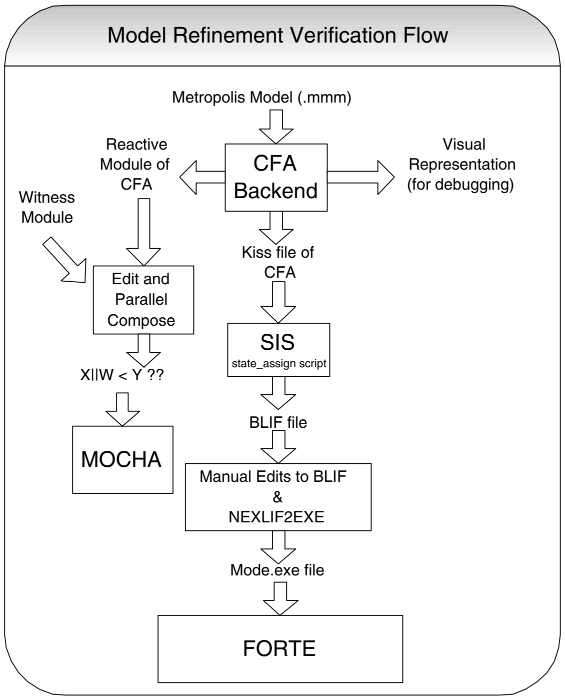

Fig. 12. Refinement Verification Flow

As you can see from figure 12, the process begins with a Metropolis Model. Using the metropolis compilation engine you can simply run it through the CFA Backend automatically. This will return a *reactive module file*, and *KISS file*, and a *visual representation*. The reactive module is fed to Mocha but first it must be augmented with *witness module* manually to do refinement checking on it. This was described in section V. The visual representation is simply for viewing. The main trunk of the flow requires that you submit the KISS file to SIS. The script in figure 10 is run to assign state encoding and logic to the symbolic states in the KISS file. This can then be written out in BLIF format. Then the slight manual edits as described previously have to be done to the BLIF file to convert it to EXLIF for FORTE. Finally you run NEXLIF2EXE (provided by FORTE) to convert the EXLIF to an executable format for FORTE.

Refinement	MOCHA Result	FORTE Result
(Test,Test2)	YES	Pending
(Test2, Test)	NO	Pending
(XX, XX2)	NO	NO
(XX2, XX)	NO	NO

TABLE I
Refinement Checking Outcomes

This flow was demonstrated using a file from the Metropolis examples distribution, *XX.mmm*, and a small example file created solely for this project, *Test.mmm*. These files were compared with modified versions, *XX2.mmm* and *Test2.mmm*, which should not be a refinement and one which should be a refinement respectively. The two "XX" files are shown in the ***appendices***. Also in the *appendices* are an example of a snippet of the visual representation, reactive module code for one of the modules, and the KISS and BLIF code generated. This gives a feel for the various representations. The "Test" files are simply not provided as to not overwhelm with information.

The results (Table I) was that the two files, *XX.mmm* and *XX2.mmm*, were indeed found to not be a refinement, (XX, XX2) = NO. This was verified by both with MOCHA and FORTE. The two files, *Test.mmm* and *Test2.mmm*, were found to be a refinement, (Test, Test2) = YES, by MOCHA. The results for FORTE for these files are still pending as of the writing of this report. While these are trivial models, this is encouraging for two reasons (1) The flow works from start to finish and (2) It can indeed begin to identify potential refinements and models which are not refinements.

As mentioned, the FORTE flow contained code provided from Intel. It is due to this limited access that the results have not yet been attained for the "Test" files. The results for the other files were as expected and increase our confidence in the flow.

The overall coding effort was primarily in the CFABackend.java and CFACodegenVisitor.java files. There is $\sim$ **1000 lines of code** between the two. They are built right into the existing Metropolis infrastructure so they can be run like any other current backend.

VII. Conclusions

The conclusion of this project is that there is now a flow in place to check the refinement of Metropolis Models. This flow was successfully shown on a model included with the Metropolis distribution package. Currently this flow only works on very simplistic models but in order to remedy this only the CFA Backend component needs to be made more robust. The tool chain will function, from the more robustly modeled CFA, correctly and should need only minor adjustments. The flow is also nicely automatic for a large portion. A small script should be able to take care of the edits needed to the BLIF file. The only major obstacle to complete automation is the creation of the *Witness Module*. This is a small tradeoff in return for the power and future usefulness of the MOCHA tool.

This project began with not only the absence of a tool chain but also a methodology. Regardless of the performance of this initial flow, the methodology is now in place and can be built upon and made more robust to handle issues as they will inevitably arise. This methodology hopefully has as its foundation sound model checking practices while at the same time taking advantages of heuristics and abstractions to target this problem domain.

VIII. Future Work

Primarily the future work will be concerned with the development of the CFA internal data structure. This is not only the most complex and semantically difficult of all the portions of the flow but it also is the foundation for the entire flow. The *FSM* and *Reactive Module* creation works acceptably and correctly provided that the CFA structure is robust and correct. Currently the CFA backend has a limited number of visitor functions. This needs to increase if more complex modules (having more complex and diverse AST nodes) are going to be examined. This initial flow was created with the example model in mind and the visitor functions and heuristics reflect this. This will be the bulk of the future work. However, it would also be interesting to explore some more of the features, particularly relating to non-determinism, associated with the reactive modules. Metropolis has the notion of non-determinism and naturally this will not work with the current FSM structure. Instead on adding some constraint to the FSM, perhaps MOCHA and reactive modules could exploit this. Naturally, the next step for larger models is to get this to work for the TTL and YAPI libraries.

Acknowledgements

Thanks to Felice Balarin, John Moondanos, Roberto Passerone, Harry Hsieh, and Yoshi Watanabe for their valuable input and guidance without which this project would not have been possible.

References

[1] Alur, R., and Henzinger, T. A. Reactive modules. *Formal Methods in System Design: An International Journal 15*, 1 (July 1999), 7–48.

[2] Alur, R., and Henzinger, T. A. *Hierarchical Verification*. Draft, Mar 2003, ch. 8.

[3] Alur, R., Henzinger, T. A., Mang, F. Y. C., Qadeer, S., Rajamani, S. K., and Tasiran, S. MOCHA: Modularity in model checking. In *Computer Aided Verification* (1998), pp. 521–525.

[4] Balarin, F., Hsieh, H., Jurecska, A., Lavagno, L., and Sangiovanni-Vincentelli, A. Formal verification of embedded systems based on cfsm networks. In *Proceedings of the 33rd annual conference on Design automation conference* (1996), ACM Press, pp. 568–571.

[5] Balarin, F., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A., Sgroi, M., and Watanabe, Y. Modeling and designing heterogeneous systems. Tech. rep., University of California, Berkeley.

[6] de Kock, E. A., Smits, W. J. M., van der Wolf, P., Brunel, J.-Y., Kruijtzer, W. M., Lieverse, P., Vissers, K. A., and Essink, G. Yapi: application modeling for signal processing systems. In *Proceedings of the 37th conference on Design automation* (2000), ACM Press, pp. 402–405.

[7] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and Sangiovanni-Vincentelli, A. SIS: A system for sequential circuit synthesis. Tech. rep., University of California, Berkeley, 1992.

[8] Henzinger, T. A., Jhala, R., Majumdar, R., Necula, G. C., Sutre, G., and Weimer, W. Temporal safety proofs for systems code. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV)* (2002), Lecture Notes in Computer Science 2404, Springer-Verlag, pp. 526–538.

[9] Hsieh, H., Balarin, F., Lavagno, L., and Sangiovanni-Vincentelli, A. Syncronous approach to functional equivalence of embedded system implementations. *IEEE Transactions On Computer-Aided Design of Integrated Circuits and Systems 20*, 8 (Aug 2001), 1016–1033.

[10] Naor, N., Lerman, Y., and Kessler, M. Forte/fl user guide. Tech. rep., Intel Corporation, Jan 2003.

[11] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. CIL: Intermediate language and tools for analysis and transformation of C programs. *Lecture Notes in Computer Science 2304* (2002), 213–228.

[12] Strehl, K., and Thiele, L. Symbolic model checking of process networks using interval diagram techniques. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)* (San Jose, California, 8–12, 1998), pp. 686–692.

Appendix I
Metropolis Models

```
package test;

process XX {

    port IntReader port0;
    port IntWriter port1;

    public XX(String name) {}
    void thread() {
        int w = 0, r = 0;

        while (w < 30) {
            block(Outer) {
                await {
                  (port1.nspace() > 0;
             port1.intWriter; port1.intWriter){
                    port1.writeInt(w);
                    w = w + 1;
                    }
                  (port0.num() > 0;
                   port0.intReader; port0.intReader)
                          r = port0.readInt();
                }
            }
        }
    }
}


package test;

process XX2 {

    port IntReader port0;
    port IntWriter port1;

    public XX2(String name) {}
    void thread() {
        int w = 0, r = 0;

        while (w < 30) {
            block(Outer) {
                await {
                  (port1.nspace() > 0;
            port1.intWriter; port1.intWriter){
                       port1.writeInt(w);
                       port0.readInt();
                       w = w + 1;
                      }
                     (port0.num() > 0;
                      port0.intReader; port0.intReader)
                      r = port0.readInt();
                }
            }
        }
    }
}
```

APPENDIX II
VISUAL REPRESENTATION SNIPPET FOR XX.MMM

```
Group: 5
Parents: 4 5 5 5
Types: 128 25 51 152
Inputs:
Outputs: writeInt port1
Misc:
Names: AwaitGuardNode BlockNode ObjectFieldAccessNode
       ThisPortAccessNode
Cond Codes: 0 0 0 0
|
V


Group: 6
Parents: 5 6
Types: 152 51
Inputs:
Outputs: port1_end writeInt_end
Misc:
Names: ThisPortAccessNode-End ObjectFieldAccessNode-End
Cond Codes: 0 0
|
V
```

APPENDIX III
REACTIVE MODULE

```
//Reactive Module of Metro CFA
module XX is

external thread_input, w_input, r_input : bool

interface r, w, writeInt, port1, port1_end, writeInt_end,
readInt, port0, port0_end, readInt_end : bool

private s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s11 : bool

atom cfa controls s0, s1, s2, s3, s4, s5, s6, s7, s8, s9,
s11, r, w, writeInt, port1, port1_end, writeInt_end,
readInt, port0, port0_end, readInt_end

reads thread_input, w_input, r_input, s0, s1, s2, s3, s4,
      s5, s6, s7, s8, s9, s11

init
[] true -> s0′ := false; s1′ := true; s2′ := false;
 s3′ := false; s4′ := false; s5′ := false; s6′ := false;
 s7′ := false; s8′ := false; s9′ := false; s11′ := false;
 r′ := false; w′ := false; writeInt′ := false;
 port1′ := false; port1_end′ := false; writeInt_end′ := false;
 readInt′ := false; port0′ := false;
 port0_end′ := false; readInt_end′ := false;

update
[] s0 = true -> s0′ := true;
[] s1 = true -> s1′ := true;
[] thread_input = true  & s1 = true -> s2′ := true;
r′ := true ; w′ := true ;
[] w_input = true  & s2 = true -> s3′ := true;
[] s3 = true -> s4′ := true;
[] s4 = true -> s5′ := true; writeInt′ := true ;
port1′ := true ;
[] s5 = true -> s6′ := true; port1_end′ := true ;
writeInt_end′ := true ;
[] w_input = true  & w_input = true  & w_input = true
& s6 = true -> s7′ := true;
[] r_input = true  & s5 = true -> s8′ := true;
readInt′ := true ; port0′ := true ;
[] s8 = true -> s9′ := true; port0_end′ := true ;
readInt_end′ := true ;
[] s8 = true -> s11′ := true;
```

APPENDIX IV
KISS FILE

```
#Kiss File for XX.mmm
#Generated by CFA Backend
.i 3
.o 10
.s 11
.p 9
# Variable Order
# thread w r  *** r w writeInt port1 port1_end
# writeInt_end readInt port0 port0_end readInt_end
100 s1 s2 1100000000
010 s2 s3 0000000000
000 s3 s4 0000000000
000 s4 s5 0011000000
000 s5 s6 0000110000
010 s6 s7 0000000000
001 s5 s8 0000001100
000 s8 s9 0000000011
000 s9 s11 0000000000
.e
```

APPENDIX V
BLIF FILE

```
.model XX.kiss
.inputs IN_0 IN_1 IN_2
.outputs OUT_0 OUT_1 OUT_2 OUT_3 OUT_4 OUT_5
         OUT_6 OUT_7 OUT_8 OUT_9
.latch    v5.0 LatchOut_v3   0
.latch    v5.1 LatchOut_v4   0
.start_kiss
.i 3
.o 10
.p 13
.s 4
.r S0
000 S0 S1 0000000000
010 S0 S0 0000000000
100 S0 S0 1100000000
000 S1 S2 0011000000
010 S1 S0 0000000000
100 S1 S0 1100000000
000 S2 S0 0000110000
001 S2 S3 0000001100
010 S2 S0 0000000000
100 S2 S0 1100000000
000 S3 S0 0000000011
010 S3 S0 0000000000
100 S3 S0 1100000000
.end_kiss
.latch_order LatchOut_v3 LatchOut_v4
.code S0 00
.code S1 11
.code S2 01
.code S3 10
.names LatchOut_v3 LatchOut_v4 [39] [30]
111 1
.names v5.0 LatchOut_v3 [39] [32]
001 1
.names LatchOut_v3 [40] [36]
11 1
.names IN_2 LatchOut_v3 [40] v5.0
1-- 1
-01 1
.names [30] LatchOut_v3 [40] v5.1
1-- 1
-01 1
.names IN_0 OUT_0
1 1
.names IN_0 OUT_1
1 1
.names [30] OUT_2
1 1
.names [30] OUT_3
1 1
.names [32] OUT_4
1 1
.names [32] OUT_5
1 1
.names IN_2 OUT_6
1 1
.names IN_2 OUT_7
1 1
```

```
.names [36] OUT_8
1 1
.names [36] OUT_9
1 1
.names IN_0 IN_1 [39]
00 1
.names LatchOut_v4 [39] [40]
01 1
.exdc
.inputs IN_0 IN_1 IN_2 LatchOut_v3 LatchOut_v4
.outputs v5.0 v5.1 OUT_0 OUT_1 OUT_2 OUT_3 OUT_4
         OUT_5 OUT_6 OUT_7 OUT_8 OUT_9
.names IN_0 IN_1 IN_2 LatchOut_v3 LatchOut_v4 v5.0
11--- 1
1-1-- 1
-11-- 1
--11- 1
--1-0 1
.names IN_0 IN_1 IN_2 LatchOut_v3 LatchOut_v4 v5.1
11--- 1
1-1-- 1
-11-- 1
--11- 1
--1-0 1
.names IN_0 IN_1 IN_2 LatchOut_v3 LatchOut_v4 OUT_0
11--- 1
1-1-- 1
-11-- 1
--11- 1
--1-0 1
.names IN_0 IN_1 IN_2 LatchOut_v3 LatchOut_v4 OUT_1
11--- 1
1-1-- 1
-11-- 1
--11- 1
--1-0 1
.names IN_0 IN_1 IN_2 LatchOut_v3 LatchOut_v4 OUT_2
11--- 1
1-1-- 1
-11-- 1
--11- 1
--1-0 1
.names IN_0 IN_1 IN_2 LatchOut_v3 LatchOut_v4 OUT_3
11--- 1
1-1-- 1
-11-- 1
--11- 1
--1-0 1
.names IN_0 IN_1 IN_2 LatchOut_v3 LatchOut_v4 OUT_4
11--- 1
1-1-- 1
-11-- 1
--11- 1
--1-0 1
.names IN_0 IN_1 IN_2 LatchOut_v3 LatchOut_v4 OUT_5
11--- 1
1-1-- 1
-11-- 1
--11- 1
--1-0 1
.names IN_0 IN_1 IN_2 LatchOut_v3 LatchOut_v4 OUT_6
11--- 1
1-1-- 1
-11-- 1
--11- 1
--1-0 1
.names IN_0 IN_1 IN_2 LatchOut_v3 LatchOut_v4 OUT_7
11--- 1
1-1-- 1
-11-- 1
--11- 1
--1-0 1
.names IN_0 IN_1 IN_2 LatchOut_v3 LatchOut_v4 OUT_8
11--- 1
1-1-- 1
-11-- 1
--11- 1
--1-0 1
.names IN_0 IN_1 IN_2 LatchOut_v3 LatchOut_v4 OUT_9
11--- 1
1-1-- 1
-11-- 1
--11- 1
--1-0 1
.end
```