# Automatic Generation of Implementations For Object-Oriented Abstractions *

Qing Yi  Jianwei Niu  Anitha R. Marneni  Jeyashree Lakshmipathy
University of Texas at San Antonio

## ABSTRACT

We present a general-purpose code transformation system, the POET system, for the purpose of automatic code generation from high-level behavior specifications of object-oriented abstractions to low-level efficient implementations in C++ and Java. In particular, we have developed an extended finite-state-machine-based language, iFSM, which models the behavior logic together with implementation details of arbitrary OO abstractions. We then use the POET system to automatically translate the behavior specifications to type-safe OO implementations in Java or C++. Finally, we use the POET system to automatically translate the behavior specifications to the input language of a model-checker (NuSMV) and apply model checking to validate the correctness of the specification. If the iFSM specification is correct, our approach can always generate a correct and type-safe implementation.

## 1. INTRODUCTION

Object-oriented programming (OOP) has become one of the most dominant programming paradigms today. In particular, a large collection of tools have been developed to effectively support the design of object-oriented software [44, 32, 26, 27, 28] and to automatically generate code skeletons from high level software design [1, 3, 2, 34, 6]. However, software developers are still required to provide implementation details for each OO abstraction by explicitly managing the control and data flow in lower level OOP languages such as C++ or Java. These implementations details, when coupled with the use of pointer based data structures (e.g., linked lists), are extremely difficult for compilers or static software verification tools to understand. As a result it is difficult to validate the correctness of abstraction implementations or to automatically improve their performance through compiler optimizations.

*This research is funded by the NSF through career award CCF0747357

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$10.00.

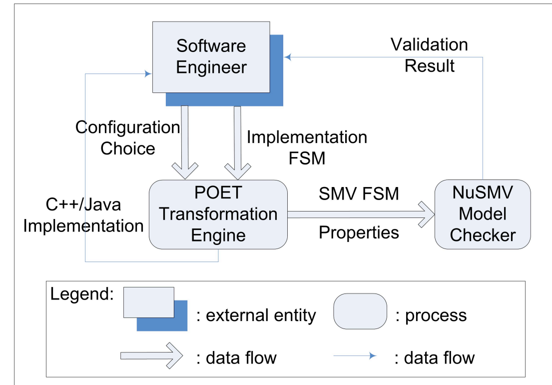


**Figure 1: The POET Translation System**

We introduce a general-purpose code transformation system, POET, for the purpose of automatic code generation from high-level behavior specifications of object-oriented abstractions to low-level efficient implementations in C++ or Java. In particular, we have developed an extended finite-state-machine-based language, the iFSM (Implementation FSM) language, which models the behavior logic together with implementation details of arbitrary OO abstractions. We then use the POET system to automatically translate the behavior specifications to type-safe OO implementations in Java or C++. Finally, we use the POET system to automatically translate iFSM specifications to the input language of a model-checker NuSMV [16], and apply model checking to validate the correctness of the specification. The structure of our translation system is shown as a data flow diagram in Figure 1.

Our goal is to automatically bridge the gap between software design using high-level specification languages (e.g., UML) and software implementation using lower level programming languages (e.g., C++,Java). Our iFSM language is essentially a UML state machine annotated with implementation specifications. An example iFSM specification is shown in Figures 3 and 4. A key insight of the language is that the infinite number of runtime states within an OO abstraction can be summarized using a finite number of summary states. Based on the abstraction of summary states, developers can express program control flow through state transitions in a FSM instead of directly managing control flow through branching instructions. The absence of branches and the explicit declaration of summary states allow implementation details of an abstraction to be much

easier to validate for correctness. At the same time, our code generation approach ensures that the auto-generated code is as efficient as those manually written by developers. Our contributions include the following.

- We present iFSM, an extended finite-state-machine-based language, to specify both the behavior logic and the implementation details of general-purpose OO abstractions.
- We introduce a general-purpose code transformation system, the POET system, which can be used to effectively translate high-level behavior specifications to low-level efficient implementations.
- We have used the POET system to automatically translate the iFSM language to efficient type-safe OO implementations in Java and C++. Our experimental results show that the auto-generated C++ implementations are as efficient as those manually written by developers.
- We have used the POET system to automatically translate the iFSM language to a model-checking language (NuSMV) and apply model checking to validate the correctness of the specifications.

In the following, Section 2 introduces the POET transformation system. Section 3 describes our iFSM language. Section 4 presents the algorithm for translating iFSM to C++/Java implementations. Section 5 presents the algorithm for translating the IFSM specification to a modeling checking language to validate the specification correctness. Section 6 presents experimental results. Section 7 presents related work. Section 8 concludes the presented work.

## 2. THE POET TRANSFORMATION ENGINE

Our POET transformation engine is built using an interpreted transformation language named POET [48, 49], which was designed specifically for the purpose of building ad-hoc translators between arbitrary languages (e.g. C/C++, Java) as well as applying transformations to programs in these languages. A preliminary design and implementation of the language have been used to successfully optimize the performance of several important linear algebra kernels in C [49, 50]. In this paper, we have used the POET language to build a system for automated code generation and validation.

Figure 2 shows the structure of our POET translator, which includes the following components.

```
<parameter fsmFiles token=(FileName) parse=LIST(STRING," ")
    default="" message="names of IFSM input files"/>
<parameter outputFile type=STRING default=""
    message="output file name for generated OO code"/>
<parameter outputLang type=STRING default=""
    message="language syntax to output the generated code"/>
<parameter smvFile type=STRING default=""
    message="output file name for generated SMV code"/>

<xform Driver pars=(input) genSMV=1 genOO=1>
......
</xform>
......
<input to=spec syntax="IFSM.code" from=fsmFiles />
<eval (ooCode,smvCode)=XFORM.Driver[genSMV=(smvFile!="");
                           genOO=(outputFile!="")](spec);
/>
<output to=(outputFile) cond=(outputFile!="")
            syntax=(outputLang) from=ooCode/>
<output to=(smvFile) cond=(smvFile!="")
            syntax=("SMV.code") from=smvCode/>
```

**Figure 2: The POET translator**

- Parameter declarations. The translator in Figure 2 starts with a number of external parameter declarations to control the configuration of code generation. Specifically, each keyword *parameter* declares a global variable, e.g., outputLang and outputFile, whose values can be re-defined by command-line configurations. A single POET translator can therefore be used to dynamically produce different output. This parameterization capability allows different software implementations be manufactured on demand based on different feature requirements.
- Transformation routine definitions. In Figure 2, the declaration of parameters is followed by a sequence of transformation routine definitions, e.g., the *Driver* routine. Each POET transformation routine takes a collection of input parameters, e.g., the abstract syntax tree (AST) representation of an input code, and returns the analysis or transformation results. They are defined and invoked similar to imperative functions in C and receives full programming support to accomplish their tasks. In particular, they can use built-in pattern matching operations together with conditionals, loops, and recursive functions to process and build both atomic values (e.g., integers, strings) and compound data structures (e.g., lists, tuples, hash tables, AST objects). Note that AST nodes are built-in data structures in POET. The built-in support for internal representation of programs makes it much easier to build program translators in POET than using a general-purpose language such as C/C++/Java.
- Processing input files and outputing results. The POET translator in Figure 2 uses a single *input* command to parse all the input programs (i.e., the IFSM files), build an abstract syntax tree (AST) representation of the entire input code, and then store the AST into a user-defined variable (the *spec* variable). The translator then invokes the *Driver* routine to traverse the AST and apply various code analysis and transformations. Finally, the transformation results (stored in variables *ooCode* and *smvCode*) are unparsed to external files using the output language syntax and the SMV language syntax respectively.

As illustrated by the input and output commands in Figure 2, POET uses syntax definitions obtained from external files to discover the structure of the input code and to unparse output code with correct syntax. This built-in support for dynamically parsing and unparsing programs in different languages makes it extremely convenient to build ad-hoc source-to-source translators. Concepts from different languages can be mixed within a single AST, and the syntax of AST nodes does not need to be specified until the transformation results are unparsed to external files.

The POET language is ideal for expressing arbitrary sequences of program transformations to the source-level rep-

resentation (i.e., the Abstract Syntax Tree) of an input code. The full programming support for customized transformations distinguishes POET from most other existing transformation languages, which rely on pattern-based rewrite rules to support definition of new transformations.

POET is designed to be a code transformation language that can be easily used to build customized source-to-source translators. In this paper, we have used the language to build a code generation system that automatically translate high-level abstraction specifications to low level efficient implementations in languages such as C++/Java.

# 3. THE IFSM LANGUAGE

Our iFSM (Implementation Finite State Machine) language was initially designed from adapting the HTS state machine language by Niu et al [40]. The goal of iFSM is to precisely describe the implementation details of an object-oriented abstraction as well as the interface of the OO abstraction. The following describes each aspect in more detail.

## 3.1 Finite State Machine Specification

The goal of the FSM specification is to precisely define the internal operational logic of an OO abstraction without explicitly managing any control flow branches. As illustrated by the example in Figures 3 and 4, each FSM is essentially a UML finite state machine annotated with implementation specifications. Each FSM specification includes the following components.

- Variables, where each variable has a name, a type, and an optional default value. The supported variable types include boolean, integer, external type name, and pointer of other types (e.g., the $ref(T)$ type for variable *obj* in Figure 3). Each external type name, e.g., the type $T$ in Figure 3, is expected to be the name of an external FSM. However, it could also remain unknown to the iFSM translator as long as the internal structure of the type is never exploited. This decision allows the iFSM language to be easily extensible. For example, $float$ or *char* can be simply used as unknown type names. Each variable in the FSM specification will be translated into a member variable in the OO abstraction implementation.
- Control states, where each state has a name and an associated boolean expression (defined using the *cond* attribute in Figure 3). Each FSM control state can represent a potentially infinite number of individual states that an OO object may stay at runtime. A FSM object is considered to be in a control state, e.g., the objIsNULL state in Figure 3, if and only if the associated boolean expression (e.g., *obj==null* && *count==null* for the $objIsNULL$ state) evaluates to true. Note that at any time, only one of the boolean expressions associated with FSM states can be evaluated to true; that is, a FSM object can stay at exactly one control state at any point in time.
- Actions, where each action has a list of input parameters and is associated with a sequence of statements. Actions are treated as macros in the FSM specification and they return nothing. They will be translated into private member methods that return void in C++/Java classes.

```
<FSM name=CountRefHandle init=objIsNULL >
  <variable name=obj type=ref(T) init=null/>
  <variable name=count type=ref(int) init=null/>

  <state name=objIsNULL cond=(obj==null && count==null)/>
  <state name=objIsUnique
      cond=(obj!=null && count!=null && val(count)==1)/>
  <state name=objIsShared
      cond=(obj!=null && count!=null && val(count)>1)/>

  <action name=init pars=(t:ref(T)) body=(...)/>
   ......
  <event name=build pars=(t:val(T))/>
  <event name=modify pars=() return=(obj) />
   ......
  <transition from=objIsNULL to=objIsUnique event=(build)
     action=(init(t.Clone()))/>
  ......
</FSM>
```

**Figure 3: FSM specification for a reference-counting abstraction**

- Events, where each event has a list of parameters and an optional return value. Each event will be translated into a public or protected member method of the generated abstraction implementation, where the statements and control flow inside the method body are specified through state transitions of the FSM.
- State transitions, where each transition is declared with a set of source states, a set of destination states, a set of triggering events, an optional boolean condition, and a sequence of statements that modify private variables of the FSM abstraction. Each transition will be translated into an if-statement in the method body of each event that triggers the transition.

The key idea of FSM specification is to use a finite number of control states to categorize the infinite number of different values that each member variable of an FSM object may have at runtime. Each public/protected method of a C++/Java class can be expressed as a parameterized event in the FSM. If the C++/Java method modifies private variables of the object, such modifications are modeled as transitions triggered by the corresponding event. The control flow within the body of each C++/Java method is expressed through the source states and the conditions associated with the transitions. The algorithm for translating FSMs to C++/Java classes is presented in Section 4.

Our iFSM language can conveniently express the implementation details of a large collection of OO abstractions. While the current language does not yet support the expression of loops, recursive functions may be expressed by generating new events within transitions. Our future work intends to model loops through the definition of container and iterator concepts.

## 3.2 Abstraction Interface Specification

While each FSM specification can precisely summarize the implementation details of an OO abstraction, the composition of abstractions requires additional details such as the access control and dynamic binding of member methods and the inheritance relations between abstractions. To support parametric polymorphism (e.g., C++ templates), abstractions may additionally have type parameters.

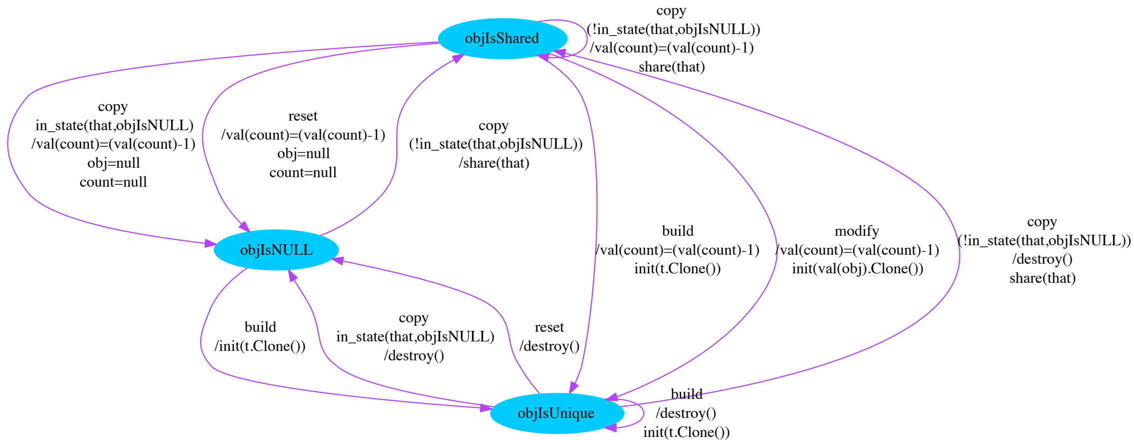


**Figure 4: The graphical representation of FSM in Figure 3**

The abstraction interface specification in Figure 5 illustrate how to specify these OO properties in our iFSM language. In particular, an arbitrary number of *FSM_class* definitions can be built from each FSM. Each *FSM_class* specification includes the name of the current class, the name of the underlying FSM, and a set of optional components including:

- A set of type parameters, which makes the class a C++ template or Java generics;
- A set of base classes that the current class inherits from;
- A set of event names to be used to build constructor methods of the class;
- A single event name to be used to build the destructor method of the class (C++ only);
- A set of extra events which comprise additional public/protected methods of the current class. None of these extra methods is allowed to modify variables of the class.
- The access control (i.e., public, protected, or private) for each event. By default, all events are translated into public methods.
- The static or dynamic binding of methods. The default binding of methods is dynamic when generating Java classes and is static when generating C++ classes.
- Additional method names for events, and whether to rename the methods for certain events.

The goal of the abstraction interface specification is to allow the flexibility of easily adapting abstraction interfaces for different needs, specifically for easy integration of the auto-generated classes with existing code. The iFSM language also supports an *invoke* declaration that allows the abstraction interface specifications to be embedded within existing C++/Java code.

# 4. GENERATING IMPLEMENTATIONS

We have used the POET translator shown in Figure 2 to automatically translate iFSM specifications such as those in Figures 3 and 5 to efficient implementations of C++/Java classes. Our translation mapping strategy includes the following.

```
<FSM_class name=CountRefHandle FSM=CountRefHandle
    parameter=(T)
    constructors=(initialize,build,copy)
    destructors=(delete)
    access=(modify:protected)
    extra_events={
       <event name=ConstRef pars=() return=(val(obj))/>
       <event name=EQ pars=(that:val(CountRefHandle))
                      return=(obj==that.obj)/>
    }
    rename=(reset=>"Reset",
            EQ=>"operator==",
            modify=>"UpdatePtr")
/>
```

**Figure 5: An abstraction interface specification**

- Each FSM variable is translated into a private member variable in the resulting class.
- Each FSM action is translated into a private member method that returns void in the resulting class.
- Each FSM event is translated into a public or protected member method in the resulting class.
- Each FSM state transition is translated into an if-statement in the method body of its triggering event.

Figure 6 shows some fragments of the auto-generated C++ class from the FSM specifications in Figures 3 and 5. In particular, the *build* method of the C++ class corresponds to the *build* event in Figure 3, and the three transitions triggered by the *build* event (see Figure 4) have been translated into the three if-statements inside the *build* method.

The details of our code generation algorithm is shown in Figure 7, which contains three routines, *Driver*, *GenClassImpl*, and *GenMethodFromEvent*. The following explains each routine in detail.

**The Driver routine** takes a single parameter, *input*, which contains a set of FSM (an example of which is shown in Figure 3) and abstraction interface (an example of which is shown in Figure 5) specifications. This routine examines each FSM specification and applies type checking to make

```
template <class T>
class CountRefHandle
{
    private:
      int* count; T* obj;
    private:
      void destroy() {...}
      void share(const CountRefHandle& that) {...}
      void init(T* t) {...}
    public:
    ......
      void build(const T& t)
      {
        if (obj!=0&&count!=0&&(*count)>1) {
          (*count) = -1+(*count);
          init(t.Clone());
        }
        else if (obj!=0&&count!=0&&(*count)==1)
        {
          destroy();
          init(t.Clone());
        }
        else if (obj==0&&count==0) {
          init(t.Clone());
        }
        return ;
      }
   ......
};
```

**Figure 6: Skeleton of auto-generated C++ class implementation**

sure that each FSM is properly defined. After constructing a symbol table for each FSM, it then takes each abstraction interface specification from *input*, finds the FSM specification for the abstraction, generates a class implementation based on both specifications, and returns the resulting class implementations.

**The GenClassImpl routine** is invoked by the *Driver* routine to generate C++/Java class implementations from three input parameters, $symTab$, the symbol table that contains definitions of all relevant FSMs; $fsm$, the FSM specification for the current class; and *impl*, the abstraction interface specification for the current class. It generates the class implementation via the following steps.

1. Initialize the class body (*clsBody* in Figure 7) to include a private member variable declaration for each FSM variable.
2. Extend the class body to include a private method definition for each FSM action. All action methods return *void*, as illustrated in Figure 6.
3. Generate C++/Java statements for autonomous transitions that are not triggered by any event. These transitions need to be evaluated at the end of all event method bodies.
4. Compute relevant information for generating event methods. In particular, *eventMap* maps each event name to all the transitions triggered by it, and $accessMap$ maps each event name to its encapsulation control (e.g.,protected vs. public).
5. Extend the class body to include constructor methods. Each constructor is generated from an FSM event by invoking the $GenConstructorFromEvent$ routine,

```
Driver(input)
  /*type checking and symbol table construction*/
  for each FSM spec. $fsm$ in $input$
     symTab[fsm.name] = GenSymTabFromFSM(fsm);
  res = null;
  for each abstraction spec. $impl$ in $input$
     fsm=find_FSM(symTab,impl);
     res=Append(res,GenClassImpl(symTab[fsm],fsm,impl));
  return res;

GenClassImpl(symTab, fsm, impl)
1.  /*generate member variable decls */
   clsBody=GenMemberVars(symTab,fsm);
2.  /*generate a private method for each FSM action*/
   clsBody=GenMethodFromActions(clsBody,symTab,fsm);
3. /*transitions not triggered by any event*/
   autoTrans=ImplAutonomousTrans(symTab,fsm);
4. /* map event names to relevant info.*/
   eventMap=MapEvent2Transitions(fsm)
   accessMap=MapEvent2AccessCtrl(impl);
5. for each event name nm in impl.constructors
     clsBody=GenConstructorFromEvent
       (clsBody,nm,symTab,eventMap,autoTrans,accMap);
6. if (impl has named destructor event nm)
     clsBody=GenDestructorFromEvent
       (clsBody,nm,symTab,eventMap,autoTrans,accMap);
7. for each event name nm in fsm
     clsBody=GenMethodFromEvent
       (clsBody,nm,symTab,eventMap,autoTrans,accMap);
8. clsBody=PostProcess(clsBody,impl);
9. return ClassImpl#(impl.name,impl.typeParams,clsBody);

GenMethodFromEvent(clsBody,name,symTab,
                   eventMap,autoTrans,accMap)
  evDef=symTab[name]; //find def. of event
  symTab=PushSymTabFromParams(evDef.pars,symTab);
  body=null;
  for each transition t in eventMap[name]
     t_cond=GenConditionFromTrans(symTab,t);
     t_body=GenStmtsFromTrans(symTab,t);
     body=GenIfElseStmt(symTab,t_cond,t_body,body);
  body=Append(body,autoTrans);
  method=Method#(name,evDef.pars,body,evDef.return);
  return Append(clsBody,AccessCtrl#(method,accMap[name]));
```

**Figure 7: Algorithm for generating class implementations**

which is similar to the $GenMethodFromEvent$ routine in Figure 7 except that all transitions triggered by the constructor event must start from the default state of the $fsm$ (e.g., the default state of the FSM in Figure 3 is objIsNULL).

6. Extend the class body with a destructor method if necessary; that is, if the interface specification *impl* includes a destructor event name. The routine *GenDestructorFromEvent* is similar to the *GenMethodFromEvent* routine except that the destructor method has a different name.
7. Extend the class body to include a member method for each event in $fsm$. This is done by invoking the *GenMethodFromEvent* routine (also defined in Figure 7).
8. Post process the class body to satisfy any additional specifications in *impl*.
9. Generate a complete class implementation and return the result. The *ClassImpl#(name,typeParams,clsBody)*

```
GenNuSMV(symTab,fsm)
1. /*gather memories referenced in fsm states*/
   traceThis=MemoryRefsInStates(symTab,fsm);
2. /*gather the conditions that trigger each transition*/
   condMap=MapTrans2Conditions(symTab,fsm);
3. /* compute alias info. of pointer variables */
   for each pointer variable x in traceThis
     tracePtr[x]=GatherModInTranstions(symTab,fsm,x);
     aliasMap[x]=GatherAliasedVars(tracePtr[x]);
4. /* compute mod info. from transitions */
   for each memory ref x in traceThis
     modMap[x]=MapModInTransitions(symTab,fsm,x,aliasMap);
5. /* gather external references in modMap and condMap*/
   traceExtern=
     GatherExternRefs(modMap) ∪ GatherExternRefs(condMap);
6. /* generate SMV variable declarations*/
   smvVars=traceThis ∪ traceExtern ∪ fsm.eventNames ∪ "state";
   smvBody=GenVarDeclInSMV(symTab,fsm,smvVars);
7. /* generate initialization of SMV variables */
   initVars=traceThis ∪ "state";
   smvBody=GenVarInitInSMV(smvBody,symTab,fsm,initVars);
8. /* generate state transitions */
   cases={Case#(condMap[t],t.to) ∀ t ∈ fsm.transitions}
   smvBody=GenAssignInSMV(smvBody,"state",cases);
9. /* generate variable modifications */
   for each x in traceThis
     cases={GenCaseInSMV(symTab,x,modMap[x][t],tracePtr[x],
                              condMap[t]) ∀ t ∈ fsm.transitions}
     smvBody=GenAssignInSMV(smvBody,x,cases);
10. /* generate properties */
   for each s in fsm.states
     GenPropertyInSMV(smvBody,symTab,s.name,s.cond);
```

**Figure 8: Algorithm for generating SMV code**

notation constructs an AST node named *ClassImpl* which can be unparsed with proper syntax in either C++ or Java. For details on the construction and unparsing of AST nodes, see Section 2.

**The GenMethodFromEvent routine** of the algorithm takes several parameters, including *clsBody*(the current class body), event name, *symTab*(the symbol table), *eventMap* (which maps each event name to transitions triggered by it), *autoTrans* (statements that implement transitions not triggered by any event), and *accMap* (which maps each event name to its access control). The routine first finds the event declaration from *symTab* and extends the symbol table with new type information on the event parameters. It then builds the body of the event method by generating a if-statement for each transition triggered by the event. In particular, the if-condition for each transition $t$ considers both the boolean expressions associated with each source states of $t$ and any additional constraints expressed in the *cond* attribute of $t$. The true branch of the if-Statement is generated from the *action* attribute of $t$, and the false branch includes the remaining statements that implement other triggered transitions. Finally, the method body is extended with statements in *autoTrans*, and a method declaration with proper access control is appended at the end of the given *clsBody*.

The algorithm in Figure 7 is correct because it accurately reflects the semantics of our iFSM languages.

# 5. VALIDATING CORRECTNESS

The key idea of our iFSM language is to provide a higher

```
traceThis={obj,count,val(count)};
tracePtr[count]={(build,new),(copy,that.count),(modify,new)}
tracePtr[obj]={(build,t.Clone),(copy,that.obj),(modify,obj.Clone)}
aliasMap[count]={CountRefHandle.count};
aliasMap[obj]={CountRefHandle.obj}

/*trans1: from objisNULL to objIsUnique triggered by build*/
modMap[obj][trans1]=(build,t.Clone());
modMap[count][trans1]=(build,new);
modMap[val(count)][trans1]=1;
condMap[trans1]={count==null && obj==null}

/*trans2: from objisNULL to objIsShared triggered by copy*/
modMap[obj][trans2]=(copy,that.obj);
modMap[count][trans2]=(copy,that.count);
modMap[val(count)][trans2]=(copy,val(that.count))+1;
condMap[trans2]={count==null && obj==null && (copy,that.obj)!=null
   && (copy,that.count)!=null && (copy,val(that.count))>=1}

/*trans3: from objisUnique to objIsNULL triggered by reset*/
modMap[obj][trans3]=0;
modMap[count][trans3]=0;
condMap[trans3]={obj!=null && count!=null && val(count)==1}
```

**Figure 9: Partial evaluation results of the Figure 8 algorithm for the Figure 4 FSM specification**

level concept over program control flow in the implementations of object oriented abstractions. In particular, the declaration of control states explicitly categorizes the infinite runtime states of an OO object so that conditional branches within a method implementation can be expressed as state transitions triggered by external events. The explicit categorization of runtime states and the absence of arbitrary branches not only make the semantics of the implementation much easier to understand, they allow various properties of the OO abstractions to be more easily validated. As a proof of concept, we show how to automatically validate the correctness of control state abstractions in FSM specifications. We are working on automatically validating other properties of OO abstractions, e.g., the aliasing of pointers and the shapes of data structures.

To validate the correctness of a FSM specification, we use another code generation algorithm, shown in Figure 8, to automatically translate iFSM specifications into the input language for a model checker, NuSMV [16]. Both code generation algorithms are implemented using the POET system, discussed in Section 2.

The goal of the code generation algorithm in Figure 8 is to validate the correctness of all the state transitions in an iFSM specification. In particular, it tries to validate that

> For each state transition *trans* declared to go from one control state *state*1 to another *state*2, if the runtime state of the memory satisfies both the boolean expression associated with *state*1 and the *cond* expression associated with *trans*, then after modifying memory by evaluating the actions of *trans*, the runtime state of memory satisfies the boolean expression associated with the *state*2.

Our algorithm therefore must generate SMV code that separately implement each transition through the tracing of control states and through direct modifications to memory. It must then use LTL (Linear Temporal Logic) properties to reconcile the results of implementing each transition via both approaches.

The code generation routine *GenNuSMV* in Figure 8 takes two parameters, *symTab*, the symbol table generated by the *driver* routine in Figure 7; and $fsm$, the finite state machine specification to validate. The algorithm follows the validation strategy through the following steps.

1. Gather memories referenced in boolean expressions associated with the control states of $fsm$. These memory stores must be traced in the model checking language (the SMV language) to validate the correctness of transitions. Therefore a SMV variable must be created for each of them. For example, in the FSM specification in FIgure 3, the memory stores referenced in the control states include $obj$, *count*, and *val*(*count*) (the dereference of the pointer variable *count*). These memory references are collected into the variable *traceThis*, shown in Figure 9.
2. Gather the triggering conditions for each transition. Specifically, the algorithm builds an associative table (*condMap*) that maps each transition $t$ to its triggering conditions, which include both the boolean expressions associated with the source states of $t$ and any additional constraints expressed in the *cond* attribute of $t$. The triggering condition is computed by calling the *GenConditionFromTrans* function as shown in the *GenMethodFromEvent* routine of Figure 7. Figure 9 shows the resulting *condMap* for some of the transitions in Figure 3.
3. Compute pointer aliasing information. First, for each pointer variable $x$ in *traceThis*, gather all the values that have been assigned to $x$ by each transition defined in $fsm$. Each value is represented using a pair $(ev, exp)$, where $ev$ is the triggering event of the transition, and $exp$ is the new value assigned to $x$. If the transition has no triggering event, a pair must be created for each event of the $fsm$. The resulting *tracePtr* table for both the *count* and $obj$ variables in Figure 3 is shown in Figure 9. The algorithm then further examines each *tracePtr* collection to extract all the member variables of FSMs that can be aliased with $x$. The result is shown in the *aliasMap* table in Figure 9.
4. For each memory reference $x$ in *traceThis*, compute the new value assigned to $x$ by each transition. In particular, the *MapModInTransitions* routine builds an associative table for $x$ by going over each transition $t$ in $fsm$ and mapping $t$ to the last value assigned to $x$ by statements in *t.action* (the action attribute of t). Note that this routine collects only the last value assigned to $x$ by each transition; that is, it does not trace intermediate modifications within transactions (e.g., modifying $x$ to be null before giving it another value). Some of the resulting *modMap* is shown in Figure 9 for various transitions.

   Because no branching instruction is allowed inside transitions, if a variable $x$ is modified within the transition, a unique expression can be determined to be the new value for $x$ if there is no ambiguity on modifying memories that may be aliased to $x$. The aliasing ambiguity is determined by examining *aliasMap*. If an ambiguity exists, a set of new values for $x$ is returned if all potential modifications can be captured. The validation fails if an aliasing ambiguity cannot be precisely modeled.
5. Gather all the external expressions that have been used to as new values for variables in *traceThis* (i.e., expressions in *modMap*) or have been used to build triggering conditions of transitions (i.e., expressions in *condMap*). An SMV variable need to be created for each of these expressions, which are collected into the *traceExtern* variable in Figure 8.
6. Generate SMV variable declarations. In particular, a SMV variable is generated for each expression in *traceThis* or *tranceExtern*. Further, a SMV variable is generated for each event name, and a special variable named *state* is generated to explicitly trace the state transitions.
7. Initialize SMV variables that need to be traced. Note that we categorize the SMV variable generated above into two groups. The first group includes the special *state* variable and variables in *traceThis*. These variables need to be initialized, and their values will be controled by transition relations. The rest of variables, which comprise the second group, are not controlled by state transitions and are therefore left uninitialized so that the model checker will enumerate all possible values for them.
8. Generate state transitions. In particular, the *state* variable is modified based on the transition declarations in $fsm$.
9. Generate memory modifications. In particular, each variable $x$ in *traceThis* is modified based on transition declarations in $fsm$; i.e., based on the values stored in $modMap[x]$.
10. Generate LTL (Linear Temporal Logic) properties to validate state transitions. In particular, a property is generated for each control state $s$ to validate that at any point of runtime, the *state* variable equals to $s$ if and only if the boolean expression associated with $s$ evaluates to true.

The algorithm in Figure 8 shows how to validate a single FSM specification. For multiple related FSMs, SMV code can be generated that collectively model their behavior through the interaction of events. Using our iFSM language, the aliasing of a pointer variable can be easily resolved if the variable is never passed outside for modification. We expect this case to be true almost universally in reality, as keeping critical variables private is the standard practice of object oriented programming.

The SMV code generated from our FSM specifications is rather simple when compared with those required for validating typical software design properties. However, we argue that validating the correctness of software implementations is an extremely hard problem and is in general not solvable. Our approach has made the problem easier to tackle by allowing developers to explicitly declare abstractions of control states and the transitions between control states. Through the FSM specification, we no longer need to handle program control flow and instead only need to reason about each transition independently. The separation of concern makes it possible to precisely determine the possible new values for each variable.

Our algorithm in Figure 8 aims to validate that the transition declarations of a FSM specification are consistent with its state declarations. It does not guarantee the complete

correctness of FSM specifications in terms of other properties. Software developers may specify additional properties in our iFSM language for validation. We are working on supporting such capabilities.

## 6. EXPERIMENTAL RESULTS

We have used our iFSM language to specify the implementations of several object oriented abstractions and have produced correct implementations for each abstraction in both Java and C++ when applicable. In particular, these abstractions include:

- Two finite state machine abstractions which are contrived to test our code generation algorithm. Both Java and C++ code are generated for these abstractions.
- A reference counting based smart pointer abstraction manually written in C++ for a compiler project [47]. A FSM specification was manually written for this abstraction, and a new C++ implementation is automatically generated. The FSM specifications and the auto-generated C++ class are shown in Figures 3,4,and 6. A Java class is not generated for this abstraction because Java does not allow explicit memory management by developers.
- Two iterator classes in C++ taken from the same compiler project. The first is a single item iterator class that provides an iterator interface for a single item; the second is a multi-iterator class which combines two iterator interfaces into a single one. Both C++ and Java classes are auto-generated for these abstractions.

Our goal is to show that the auto-generated class implementations perform as well as alternative implementations manually written by professional C++/Java developers. Since all of our manual implementations are in C++, we compare the performance of auto-generated C++ code with their manually written counter parts. Figure 10 shows the result of comparison.

We have manually written a C++ driver function to measure the performance of each C++ class. For each class, the driver builds a large array of class objects and then invoke public methods of the objects a large number of times (proportional to the array size). Two executables are generated for each class by linking the same C++ driver with different class implementations: the auto-generated implementation and the manually written implementation.

We have compiled all the C++ code using g++ 4.2.0 with -O2 option. The elapsed time of each executable is measured on an Intel 2.16 GHz Core2Duo processor with 1GB 667MHz memory and 4MB L2 cache. The performance results are reported in Figure 10.

From Figure 10, we see that all the auto-generated C++ code perform similarly as the manually written one. In particular, the auto-generated iterator class implementations consistently performed slightly better than the manually written ones, while the auto-generated reference counting class implementation performed slightly worse than the manually written one. The differences are minor and are likely caused by random factors in the compiler.

## 7. RELATED WORK

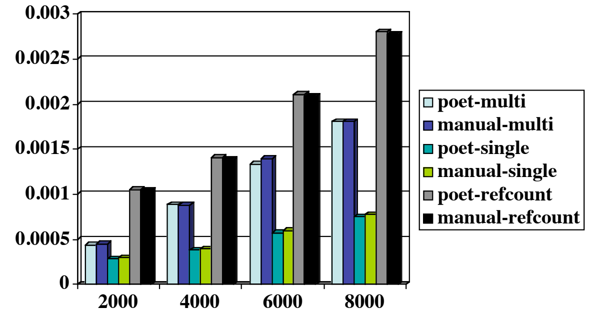


**Figure 10: Performance (elapsed time in seconds) of abstraction implementations when used in arrays of different sizes**

We present a code generation approach which aims to fill the gap between high-level software design and low-level efficient implementations. While numerous projects [33, 35, 45] have automatically produced program implementations from finite state machine specifications, these implementations target finite state machine abstractions in specialized domains, e.g., embedded systems. In contrast, our goal is not generating implementations of finite state machine abstractions. Instead, we use a finite number of control states to abstract the infinite number of runtime states within general-purpose OO abstractions. We then use state transitions to specify the implementation of general-purpose OO abstractions without explicitly managing control flow.

Program transformation tools have long been used to analyze, modify, reshape, and optimize existing code, including re-documenting/re-implementing code, reverse engineering, changing APIs, and porting to new platforms [10, 4, 24]. A number of translation systems [7, 9, 39, 23, 25] can automatically generate programs from formal specifications such as system design model [43], mathematical formulations [22, 9], reflection of metadata and code [21], design patterns [14] and dataflow graphs [41]. Several general-purpose transformation languages and systems have been developed [31, 20, 13, 5] and some have been widely adopted [13, 5]. These tools and systems mostly rely on pattern-based transformation rules coupled with application strategies [39, 23, 13, 5, 31, 25]. Although these rules are convenient to use and easy to learn, they are limited in their capability to express arbitrary program transformations. Our work is based on a more complete transformation language, POET, which supports compound data structures, arbitrary control flow, and recursive functions. We focus on combining program transformation with software verification technology to ensure both correctness and efficiency of generated code.

Generative programming is a software engineering paradigm that focuses on effective configuration and integration of customized reusable software [18, 46, 8] and automatic adaptation of software components [29] in the context of product-line and model-driven architectures [12, 44]. Our research also aims at automatic software construction, but we focus on generating implementations of individual object-oriented abstractions instead of integration and adaptation of existing software components.

Model driven development [44, 32] captures important aspects of a software system through models [26, 27, 28] before producing executable code [1, 3, 2, 34, 6]. There have been

research work [30, 36, 1, 3, 2] and successful commercial tool suites, including IBM Rational Rose, Telelogic SDT, and i-Logix STATEMATE, that support aspects of Model Driven Engineering, such as software modeling and code generation. However, most of these systems provide only the simplest analysis capabilities, such as consistency checking and simulation, and most produce only skeletons of C++/Java code. Our work provide tools that combine UML diagrams with additional semantic descriptions to fully automate the generation of object-oriented implementations.

Formal methods have then be used widely to validate the correctness of both system design and implementations [11, 17, 42, 37, 15, 38, 19]. We use the NuSMV [16] model checker to validate the abstraction of state transitions in our iFSM language.

# 8. CONCLUSIONS

This paper presents a general purpose code transformation system to automate the generation of efficient implementations from high-level specifications of object oriented abstractions. Our approach aims to automatically bridge the gap beteen software design using high-level specification languages (e.g., UML) and software implementation using lower level programming languages (e.g., C++,Java). Our current iFSM language uses a finite number of summary states to categorize the infinite number of different values that each member variable of an OO abstraction may have. Through the specification of state transitions, developers no longer need to explicitly manage program control flow in low level OO pogramming languages such as C++ and Java. We have shown that this approach produces implementations that are as efficient as those manually written by developers and that the absence of branches allows implementation details of the abstraction to be easily validated for correctness.

# 9. REFERENCES


[1] Implementing uml statechart diagrams. Pathfinder Solutions. www.PathfinderMDA.com.

[2] Metamill 4.0. www.metamill.com.

[3] Umodel. Altova Inc. http://www.altova.com/.

[4] R. Akers, I. Baxter, M. Mehlich, B. Ellis, and K. Luecke. Re-engineering c++ component models via automatic program transformation. In *Twelfth Working Conference on Reverse Engineering*. IEEE, 2005.

[5] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

[6] K. Balasubramanian, A. S. Krishna, E. Turkay, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt. Applying model-driven development to distributed real-time and embedded avionics systems. international journal of embedded systems. *special issue on Design and Verification of Real-time Embedded Software*, April 2005.

[7] R. Balzer, N. Goldman, and D. Wile. On the transformational implementation approach to programming. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 337–344, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[8] D. S. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. In *International Conference on Software Reuse*, pages 117–136, 2000.

[9] F. L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations-computer-aided, intuition-guided programming. *IEEE Trans. Softw. Eng.*, 15(2):165–180, 1989.

[10] I. D. Baxter. Using transformation systems for software maintenance and reengineering. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 739–740, Washington, DC, USA, 2001. IEEE Computer Society.

[11] H. Berg, W. Boebert, W. Franta, and T. Moher. *Formal Methods of Program Verification and Specification*. Prentice Hall, 1982.

[12] J. Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[13] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.16. a language and toolset for program transformation. *Science of Computer Programming*, 2007.

[14] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Syst. J.*, 35(2):151–171, 1996.

[15] S. Chaki, E. Clarke, and A. Groce. Modular verification of software components in c. *Transactions of Software Engineering*, 1(8), Sept. 2004.

[16] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[17] E. Clarke and J. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, Dec. 1996.

[18] J. C. Cleaveland. Building application generators. *IEEE Software*, 5(4):25–33, 1988.

[19] M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 57–68, New York, NY, USA, 2002. ACM.

[20] M. Erwig and D. Ren. A rule-based language for programming software updates. *SIGPLAN Not.*, 37(12):88–97, 2002.

[21] M. Fähndrich, M. Carbin, and J. R. Larus. Reflective program generation with patterns. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*,

pages 275–284, New York, NY, USA, 2006. ACM Press.

[22] M. S. Feather. A system for assisting program transformation. *ACM Trans. Program. Lang. Syst.*, 4(1):1–20, 1982.

[23] P. Freeman. A conceptual analysis of the draco approach to constructing software systems. *IEEE Trans. Softw. Eng.*, 13(7):830–844, 1987.

[24] Y. Futamura, Z. Konishi, and R. Glück. Wsdfu: program transformation system based on generalized partial computation. *The essence of computation: complexity, analysis, transformation*, pages 358–378, 2002.

[25] D. Garlan, L. Cai, and R. L. Nord. A transformational approach to generating application-specific environments. In *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 68–77, New York, NY, USA, 1992. ACM Press.

[26] J. A. Goguen and R. M. Burstall. Institutions: abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

[27] J. Gray, T. Bapty, and S. Neema. Handling crosscutting constraints in domain-specific modeling. In *Communications of the ACM*, pages 87–93, October 2001.

[28] M. Groe-Rhode, F. P. Presicce, and M. Simeoni. Formal software specification with refinements and modules of typed graph transformation systems. *J. Comput. Syst. Sci.*, 64(2):171–218, 2002.

[29] C. Haack, B. Howard, A. Stoughton, and J. B. Wells. Fully automatic adaptation of software components based on semantic specifications. In *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, pages 83–98, London, UK, 2002. Springer-Verlag.

[30] D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 34(1):53–60, 2001.

[31] S. S. Huang, D. Zook, and Y. Smaragdakis. Statically safe program generation with safegen. In *Generative Programming and Component Engineering*, 2005.

[32] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture Practice and Promise.* Addison Wesley, 2003.

[33] A. Knapp and S. Merz. Model checking and code generation for uml state machines and collaborations. In *Proc. 5th Wsh. Tools for System Design and Verification*, pages 59–64, 2002.

[34] A. Kogekar, D. Kaul, A. Gokhale, P. Vandal, U. Praphamontripong, S. Gokhale, J. Zhang, Y. Lin, and J. Gray. Model-driven generative techniques for scalable performability analysis of distributed systems. In *In Proceedings of the NSF NGS Workshop, International Conference on Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 2006.

[35] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc.* O'Reilly & Associates, 1992.

[36] S. Maoz and D. Harel. From multi-modal scenarios to code: Compile LSCs into AspectJ*. ACM Press, 2005.

[37] S. Narayanan and S. A. M. Simulation. verification and automated composition of web services. In *In Proceedings of 11th World Wide Web Conference*, pages 77–88, 2002.

[38] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[39] J. Neighbors. Software construction using components, 1980.

[40] J. Niu, J. M. Atlee, and N. A. Day. Composable semantics for model-based notations. In *the 10th SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-10)*, pages 149–158. ACM Press, 2002.

[41] H. Oh and S. Ha. Efficient code synthesis from extended dataflow graphs for multimedia applications. In *Design Automation Conference*, 2002.

[42] S. OWRE, J. RUSHBY, and N. SHANKAR. Pvs: A prototype verification system. In *In Eleventh International Conference on Automated Deduction (CADE)*, 1992.

[43] I. Sander and A. Jantsch. Transformation based communication and clock domain refinement for system design. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 281–286, New York, NY, USA, 2002. ACM Press.

[44] R. Soley and O. S. S. Group. Model driven architecture, 2000.

[45] A. Wasowski. On efficient program synthesis from statecharts. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 163–170, New York, NY, USA, 2003. ACM.

[46] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999.

[47] Q. Yi. *Transforming Complex Loop Nests For Locality.* PhD thesis, Rice University, 2002.

[48] Q. Yi. The poet language manual, 2008. www.cs.utsa.edu/ qingyi/POET/poet-manual.pdf.

[49] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. In *Workshop on Performance Optimization for High-Level Languages and Libraries*, Mar 2007.

[50] Q. Yi and C. Whaley. Automated transformation for performance-critical kernels. In *ACM SIGPLAN Symposium on Library-Centric Software Design*, Montreal, Canada, Oct. 2007.