COLLECTIVE SPECIFICATION AND VERIFICATION OF BEHAVIOR MODELS AND OBJECT-ORIENTED IMPLEMENTATIONS[*]

Qing Yi, Jianwei Niu and Anitha R. Marneni
University of Texas at San Antonio, One UTSA Circle, San Antonio, TX, U.S.A.

Keywords: Code generation, Modeling checking, Finite state machine.

Abstract: We present a finite-state-machine-based language, iFSM, to seamlessly integrate the behavioral logic and implementation strategies of object-oriented abstractions and prevent them from being out-of-sync. We provide a transformation engine which automatically translates iFSM specifications to lower-level C++/Java class implementations that are similar in style to manually written code. Further, we automatically verify that these implementations are consistent with their behavior models by translating iFSM specifications into the input language of model checker NuSMV.

1 INTRODUCTION

A large collection of development tools, e.g., Pathfinder, Metamill, UModel, among others (Balasubramanian et al., 2005; Kogekar et al., 2006) can automatically generate C++/Java code from models in various notations, such as class diagrams and statecharts. However, most of these tools require developers to manually complete the auto-generated code skeletons. Since the manual implementations are maintained separately from their higher-level design, they can easily become out-of-sync.

We introduce a finite-state-machine based language, iFSM, to seamlessly integrate software modeling notations such as HTS (Niu et al., 2003) with implementations using lower-level languages such as C++ and Java. A key contribution of iFSM is a concise mapping from the behavior models of arbitrary C++/Java classes, expressed using FSMs, to their implementations, expressed using a language that is independent of either C++ or Java. Such a mapping effectively unifies the design and implementation of OO classes to provide the following benefits.

- The behavior models and implementation strategies of object-oriented classes are unified and maintained together, and their consistency automatically verified via model checking techniques.

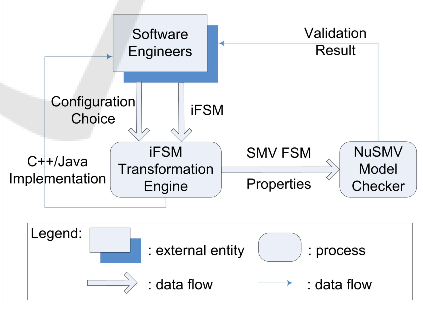

Figure 1: The iFSM framework.

- A single iFSM specification can be automatically translated to equivalent OO implementations in different programming languages and thus can conveniently support multi-lingual applications.
- The behavior models of C++/Java classes can be made available to compilers and potentially enable more aggressive optimization, which is the subject of our future work.

Figure 1 shows the work flow of our framework, where we use a single transformation engine to automatically translate iFSM specifications to C++/Java classes and to the input language of model checker NuSMV (Cimatti and et. al., 2002). The auto-generated C++/Java classes are similar in style to manually written code and therefore can be easily integrated with existing legacy code. The unification of behavior and implementation notations within iFSM

[*]This research is funded by the NSF through award CCF0747357 and CNS-0964710.

Yi Q., Niu J. and R. Marneni A..
COLLECTIVE SPECIFICATION AND VERIFICATION OF BEHAVIOR MODELS AND OBJECT-ORIENTED IMPLEMENTATIONS.
DOI: 10.5220/0003439300150024
In *Proceedings of the 6th International Conference on Software and Database Technologies* (ICSOFT-2011), pages 15-24
ISBN: 978-989-8425-77-5
Copyright © 2011 SCITEPRESS (Science and Technology Publications, Lda.)

```
aFSM(Iterator,dstate(Started),tparam(T),inherit()) {
  States: Started,Ended;
  Events: Advance:()->(); Reset:()->();
          ReachEnd:()->bool; Current:()->T;
  Trans:  Advance(): Started->(Started,Ended);
          Reset(): (Started,Ended)->Started;
          Current(): Ended->ERROR;
}
```

Figure 2: Example aFSM for an *Iterator* interface.

also allows their consistency to be automatically verified using model checking techniques.

2 THE iFSM LANGUAGE

Our iFSM language supports three main concepts, the abstract *FSM* (aFSM), which models the runtime behavior of an object-oriented class (see Section 2.1); the implementation FSM (iFSM), which extends the *aFSM* to additionally model implementation strategies (see Section 2.2); and *iFSM_class*, which specifies how to adapt the interface of automatically generated OO classes when translating iFSM to programming languages such as C++/Java (see Section 2.3).

2.1 The Abstract FSM

As illustrated by the *Iterator* interface in Figure 2, each aFSM has a default state (specified by the *dstate* attribute), a list of type parameters (specified by the *tparam* attribute), a list of base FSMs that the current FSM inherits (specified by the *inherit* attribute), and the following additional components.

- **A Finite Number of Control States** which categorize the runtime values that an arbitrary FSM object may have. Each object can stay in exactly one of the declared states at any time. For example, an object of the *Iterator* FSM in Figure 2 can stay either in state *Started*, which means the object is in normal working condition, or in state *Ended*, which means the object is no longer in operation.
- **A List of Events** which define the interface of an FSM object to communicate with the external world. Each event has a list of parameters and can optionally return a value. The *iterator* FSM in Figure 2 has four events, where *Advance* and *Reset* have no parameter and return nothing; *ReachEnd* takes no parameter and returns a boolean, and *Current* returns a type *T* value.
- **A List of Transitions** which model the change of states within an object as it responds to different events. Each transition has a triggering event, a set of source and destination states, and an optional boolean expression which specifies additional constraints required for the transition. For example, the *Advance* event in Figure 2 will trigger an *Iterator* object to transition from state *Started* to either *Started* or *Ended*, and the *Current* event will raise an exception (enter the *ERROR* state) if the object is in the *Ended* state.

```
aFSM(CloneFSM,dstate(),tparam(),inherit())
  {Events:Clone:()->ref(fsm(CloneFSM));}
iFSM(CountRefHandle,dstate(objNULL),tparam(T:CloneFSM),
      inherit(),init(),delete(reset)){
 Vars:   obj:ref(T)=null; count:ref(int)=null;
 States: objNULL: (count==null) -> (obj==null);
   objUnique: (count!=null&&val(count)==1) -> (obj!=null);
   objShared: (count!=null&&val(count)>1) -> (obj!=null);
 Events: build:(t:T)->(); const_ref:()->obj;
   modify:()->obj; reset:()->();
   copy:(that:fsm(CountRefHandle,T))->();
   EQ: (that:fsm(CountRefHandle,T))->(obj==that.obj);
 Actions:share:(that:ref(fsm(CountRefHandle,T)))->()
  {obj=that.obj;count=that.count;val(count)=val(count)+1;}
   init:(t:ref(T))->() {obj=t;count=new(int,1);}
   destroy:()->() { delete(count,obj);}
 Trans:build(t):objNULL->objUnique: {init(t.Clone());}
   build(t):objUnique->objUnique
              {destroy(); init(t.Clone());}
   build(t):objShared->objUnique
            {val(count)=val(count)-1;init(t.Clone());}
   reset():objUnique->objNULL {destroy();}
   reset():objShared->objNULL
      {val(count)=val(count)-1;obj=null;count=null;}
   copy(that) & !in_state(that,objNULL):
      objNULL->objShared {share(that);}
   copy(that) & !in_state(that,objNULL):
      objUnique->objShared {destroy();share(that);}
   copy(that) & !in_state(that,objNULL): objShared
       ->objShared {val(count)=val(count)-1;share(that);}
   copy(that) & in_state(that,objNULL):
                objUnique->objNULL { destroy();}
   copy(that) & in_state(that,objNULL): objShared->objNULL
        {val(count)=val(count)-1,obj=null,count=null;}
   modify():objShared->objUnique
        {val(count)=val(count)-1;init(val(obj).Clone());}
iFSM_class(CountRefHandle) {
   constructors: build,copy;
   access:       modify:protected;
   binding:      reset:dynamic;
   extra:        copy=>"operator="; EQ=>"operator=="
 }
```

Figure 3: Example iFSM for a reference counting C++ class.

In summary, each aFSM specifies the expected behavior of a C++ abstract class or a Java interface. They serve as interface specifications to support interactions between different software components.

2.2 The Implementation FSM

As illustrated by Figure 3, each iFSM extends the aFSM to additionally specify implementation details including the following.

Object Construction and Deletion. For example, at line 2 of Figure 3, $init()$ specifies that no parameter is required to build a *CountRefHandle* object, and $delete(reset)$ specifies that the *reset* event should be triggered when deleting a *CountRefHandle* object.

Member variables. For example, line 3 of Figure 3 declares two iFSM variables, obj, which points to a type T object, and *count*, which points to an integer. Both variables are initialized with $null$.

Conditions and Implications of Control States. Each iFSM control state s has two attributes, $condition(s)$, which evaluates to true if and only if the iFSM object is in the s state; and $imply(s)$, which is implied if the object is in the s state. For example, line 4 of Figure 3 specifies that a $CountRefHandle$ object is in the *objNULL* state if and only if *count==null* is satisfied at runtime. Further, if an object is in the *objNULL* state, then *obj==null* is guaranteed to hold.

Event Implementations. Each iFSM event can include its own local variables, control states, and transitions; that is., each event can contain an embedded iFSM to model any complex algorithm triggered by the event. Note that events cannot be nested inside one another, so local transitions within an event no longer have any triggering event (these are called *autonomous* transitions). In Figure 3, lines 7-8 contain all event definitions, all of which are simple enough that no embedded iFSMs are required.

Local Actions. which are defined similarly as events but are private members of the iFSM. Specifically, they cannot be used to as triggers of state transitions and cannot be invoked from outside the iFSM. The iFSM in Figure 3 has three actions, *share*, *init*, and *destroy*, defined at lines 9-11.

Transition Implementations. As illustrated by Figure 3 at lines 12-24, each iFSM transition contains a sequence of statements as its implementation. The statements currently supported by iFSM are listed in Table 1. Note that iFSM does not directly support any control-flow statements, so Table 1 has no if-conditionals or while-loops. This restriction is necessary to simplify the verification of iFSMs.

Autonomous Transitions not triggered by any event. Each autonomous transition t is controlled by a boolean attribute *repeat* which specifies whether t should be triggered repetitively until the triggering condition no longer holds. If local to an event *ev*, an autonomous transition t can be triggered by three options, *pre*, which triggers t before evaluating any transition triggered by *ev*; *post*, which triggers t after evaluating all transitions triggered by *ev*; and *loop*, which evaluates t as part of each *LOOP()* statement

Table 1: iFSM Expressions and Statements.

type expressions	
fsm(n,targs)	an aFSM/iFSM with name n and type params $targs$
ref(t)	a reference (pointer) type to values of type t
array(t,s)	an array of size s and element type t
expressions	
+,-,*,/,%,	arithmetic operators
==,>=,<=,!=,>,<	comparison operators
&&, ||, !	boolean operators (*and*, *or*, and *not*)
$f(args)$	invoke action/event f using *args* as parameters
$new(t,o)$	a new object of type t and initialized with value o
$new\_array(t,n,o)$	a new array of n type t items, each initialized with o
$a[s]$	the element at subscript s of array a
$a.b$	the attribute b of an FSM/iFSM object a
val(p)	the value of the memory referenced by address p
ref(v)	the memory address associated with variable v
in_state(x,y)	whether the *aFSM/iFSM* object x is in state y
statements	
$delete(p)$	free the memory referenced by p
$except(x)$	raise an exception x
$m = exp$	assign a new value *exp* to memory expression *m*.
$LOCAL(x : t = i)$	create a local variable x with type t and initial value i
$LOOP()$	iteratively evaluate autonomous transitions

(see table 1) invoked by a transition triggered by *ev*.

Exception Handling and Debugging Support, which can be associated with each event, action, or transition. We omit their explanations here due to space constraints.

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

Figure 4: Auto-generated C++ code from Figure 3.

Table 2: Mapping iFSM to object-oriented C++/Java classes.

iFSM component	C++/Java component
iFSM variables	private member variables in C++/Java classes
iFSM actions	private member functions in C++/Java classes
Nested iFSMs	inner classes nested inside C++/Java classes
iFSM events	public/protected methods in C++/Java classes
event variables	local variables of C++/Java methods
event transitions	top-level if-conditionals in C++/Java methods
auto. transitions	loops or nested if-conditionals in C++/Java methods

2.3 Interface Adaptation for C++/Java

Each iFSM component is designed to correlate FSM notations with OO implementation details, and the translation mappings are shown in Table 2. In particular, All aFSM events are translated to dynamically-bound public methods, All iFSM actions and events are translated to statically-bound (i.e., non-virtual) methods in C++ but dynamically-bound (i.e., non-static) methods in Java, as these are the default method bindings in C++ and Java. The default access control and binding of each method can be overridden, as illustrated by the `iFSM_class` at lines 25-30 of Figure 3, via the following interface adaptations.

- Extra class constructors. In Figure 3, line 26 specifies that two extra constructors should be defined for the *CountRefHandle* iFSM based on state transitions triggered by the *build* and *copy* events.
- Alternative access control. In Figure 3, line 27 specifies that the event $modify$ should be made *protected* instead of *public*.
- Alternative method binding. In Figure 3, line 28 specifies that the event *reset* should be made dynamically bound.
- Alternative names for existing events. In Figure 3, line 29 specifies that an extra name “operator=” should be used for event *copy*, and an extra name “operator==” should be used for event EQ.

The `iFSM_class` specification provides flexibility for users to easily adapt the interface of an auto-generated C++/Java class for different needs. Figure 4 shows a portion of the C++ code automatically generated from the iFSM specification in Figure 3. The code generation is discussed in more detail in Section 3.

2.4 Expressiveness of the Language

As it stands, our iFSM language serves as a proof-of-concept in collectively specifying both software behavioral designs and their OO implementations. Table 1 shows the set of expressions and statements currently supported by the language, which are a small subset of those in C++/Java and are expected to be extended when used to model larger and more complex software systems beyond what we have studied.

While incomplete, iFSM has the potential to conveniently specify arbitrary general-purpose OO classes. To demonstrate this potential, we have used the language to fully specify a large and complex C++ class which supports the parsing capability of a research compiler project (see Section 5.1). Although compilers and language interpreters are typically sequential and do not need to deal with concurrent evaluation of components, they are among the most challenging software to build, and their implementations typically feature extremely complex and delicate control logics that are easily broken when the code needs to be modified for maintenance (e.g., bug fixing) or for functionality enhancement. We found that by explicitly specifying its behavioral logic, the new generated code has better structure and is easier to read and understand. Further, since the behavior model of class implementations are made explicit, their consistency can be more easily verified (see Section 4).

3 AUTO-GENERATING C++/JAVA CODE

To automatically translate iFSM specifications to C++/Java class implementations, we use the transformation engine shown in Figure 1 which is implemented using POET (Yi et al., 2007), an interpreted program transformation language designed for building ad-hoc translators between arbitrary languages (e.g. C/C++, Java) as well as applying transformations to programs in these languages. Our iFSM translator can be configured via command-line parameters to dynamically produce output in C++, Java, or the input language of the NuSMV model checker. Figure 4 shows a portion of the C++ class automatically generated by our transformation engine from Figures 3.

3.1 The Code Generation Algorithm

Figure 5 shows our algorithm for translating iFSMs to C++/Java classes. The algorithm takes two parameters, a list of aFSM/iFSM declarations (*ifsm_decls*) and a list of interface adaptations (*adapt_decls*). It first applies type checking to verify that all aFSM/iFSMs in *ifsm_decls* are properly defined and constructs a symbol table for each FSM in the process. The algorithm then takes each interface adaptation from *adapt_dcls*, finds the corresponding iFSM, and generates an object-oriented class accordingly.

```
GenClassImpl(ifsm_decls, adapt_decls)
  globalTable = TypeCheck(ifsm_decls);
  for each (ifsm,adapt) in adapt_decls
     symTab= lookup_symbol_table(globalTable, ifsm);
     clsBody = empty; /*body of generated class*/
1.   generate_member_variable_decls(clsBody, symTab, ifsm);
2.   generate_private_actions(clsBody, symTab, ifsm);
3.   for (each inner iFSM m nested inside ifsm):
       gen_inner_class(clsBody,symTab,m,adapt);
4.   /* map event names to relevant info.*/
     accMap=map_event_to_accessCtrl(adapt);
     bndMap=map_event_to_binding(adapt);
     trMap=map_event_to_transitions(ifsm);
     autoMap=map_event_to_autoTransitions(ifsm);
5.   gen_default_constructor(clsBody, symTab,ifsm);
     for (each event ev in extra_constructors_of(adapt)):
       gen_constructor_from_event(clsBody,symTab,ev,trMap[ev],
                          autoMap[ev],accMap[ev],bndMap[ev]);
6.   if (ifsm has a destructor event ev)
        gen_destructor_from_event(clsBody,symTab,ev,trMap[ev],
                          autoMap[ev],accMap[ev],bndMap[ev]);
7.   for (each event ev defined in ifsm)
        gen_method_from_event(clsBody,symTab,ev,trMap[ev],
                          autoMap[ev],accMap[ev],bndMap[ev]);
8.   for (each additional method wrapper (ev,name) in adapt)
        gen_method_wrapper(clsBody, ev, name);
9.   output_class_impl(name_of(ifsm),type_param_of(ifsm),
                             inherit_by(ifsm),clsBody);
```

Figure 5: Generating class implementations.

The *GenClassImpl* routine in Figure 5 essentially follows the mapping rules shown in Table 2 to translate each iFSM to a corresponding C++/Java class. In particular, after setting up the symbol table properly, steps (1-2) of the algorithm translate iFSM variables and actions; step(3) recursively invokes the algorithm to translate nested iFSMs; steps(4-7) translate events and transitions into class member functions; and steps (8-9) post-process the class body (based on interface adaptations) and unparse the final class implementation with proper syntax in either C++ or Java.

The main task of the algorithm is translating events to class member functions. Here step(4) precomputes two associative maps, *trMap*, which maps each event to the state transitions triggered by it, and *autoMap*, which maps each event to the pertinent autonomous transitions. Steps (5-7) then combine such information with additional interface adaptation information (*accMap* and *bndMap*) to translate each event *ev* to a member function or a constructor/destructor of the class. Specifically, an if-else-branch is generated for each transition *t* triggered by *ev*, where the if-condition considers both the source states and any additional constraints associated with *t*, the true-branch includes all statements associated with *t*, and the else branch includes implementations of other transitions triggered by *ev*.

Most iFSM statements can be translated to C++/Java in a straightforward fashion. A special case is the *LOOP* statement (see Table 1), which is translated to a sequence of loops and if-statements. In particular, for each *loop*-triggered autonomous transition *t* within the current event, a *while* loop is generated if the source and destination states of *t* are different or if the *repeat* attribute of *t* is set to true; otherwise, an if-statement is generated for *t*. Code generation for *pre*- and *post*-triggered autonomous transitions (see Section 2.2) are supported in a similar fashion, except that these transitions are evaluated at the entry and exit of the corresponding event or action.

3.2 Correctness and Profitability

Our algorithm loyally follows the translation rules shown in Table 2, which define the operational semantics of the iFSM specifications. Consequently, the generated code is guaranteed to be correct if the input is known to be correct. However, if the input iFSM contains a semantic error, e.g., dereferencing a null-pointer or accessing an array out-of-bound, the error appears accordingly in the generated code. To alleviate this problem, we translate iFSM specifications to the input of a model checker, NuSMV, to automatically detect semantic errors in iFSM specifications.

The goal of our iFSM language is to raise the level of abstractions so that developers can focus on the behavior design of OO classes and then explicitly map their design to concrete implementations. The design and implementation are collectively specified and maintained together so that they never become out-of-sync. As illustrated by Figure 4, our auto-generated code is similar in style to hand written code and is easily understandable by both human and compilers, so it can be seamlessly integrated with existing code and can benefit from the same level of automatic performance optimization by compilers.

4 VERIFYING iFSM SPECIFICATIONS

A key objective of iFSM is to unify the behavior and implementation of an OO class so that their consistency can be readily verified. In particular, the control states of each iFSM categorize the different values that an iFSM object may have at runtime. As the object goes through various modifications triggered by event invocations, the modification of iFSM variables must conform to the declared state transitions. We use NuMSV (Cimatti and et. al., 2002), a BDD-based general-purpose model checker, to trace modifications to the iFSM variables as different events are invoked and relevant state transitions are triggered.

```
   VAR
1: state : {objNULL,objUnique,objShared};
   obj : 0..3; count : 0..3; val_count : -21..20;
2: copy_that_obj : 0..1; copy_that_count : 0..1;
   val_copy_that_count : -21..20;
3: EQ: boolean; const_ref: boolean; reset: boolean;
   modify: boolean; copy: boolean; build: boolean;

   ASSIGN
4: init(state) := objNULL; init(count) := 0; init(obj) := 0;
5: next(state) := case
6:  build : objUnique;
7:  reset : objNULL;
8:  copy&(!(copy_that_count=0))&(state=objNULL) : objShared;
9:  copy&!(copy_that_count=0)&(state=objUnique) : objShared;
10: copy&!(copy_that_count=0)&(state=objShared) : objShared;
11: copy&(copy_that_count=0)&(state=objUnique) : objNULL;
12: copy&(copy_that_count=0)&(state=objShared) : objNULL;
13: modify&(state=objShared) : objUnique;  1 : state;
   esac;
14:next(obj) := case
15: build&(obj=0)&(count=0) : 1;
16: build&(obj!=0)&(count!=0)&(val_count>=1) : 1;
17: reset&(obj!=0)&(count!=0)&(val_count>=1) : 0;
    copy&(!(copy_that_count=0))&(count=0) : 2;
18: copy&!(copy_that_count=0)&(count!=0)&(val_count=1) : 2;
19: copy&(copy_that_count=0)&(count!=0)&(val_count=1) : 0;
20: copy&!(copy_that_count=0)&(count!=0)&(val_count>1) : 2;
21: copy&(copy_that_count=0)&(count!=0)&(val_count>1) : 0;
22: modify&(obj!=0)&(count!=0)&(val_count>1):3;  1 : obj;
   esac;
23:next(val_count) := case ...... esac;
24:next(count) := case ...... esac;

25:LTLSPEC G(state=objNULL -> obj=0&count=0)
26:LTLSPEC G(obj=0&count=0 -> state=objNULL)
27:LTLSPEC G(state=objUnique -> obj!=0&count!=0&val_count=1)
28:LTLSPEC G((count!=0)&val_count=1 -> state=objUnique)
29:LTLSPEC G(state=objShared -> obj!=0&count!=0&val_count>1)
30:LTLSPEC G((count!=0)&(val_count>1) -> state=objShared)
```

Figure 6: Auto-generated NuSMV input from Fig. 3.

Figure 6 shows the result of translating the *CountRefHandle* iFSM in Figure 3 to the NuSMV input for verification.

4.1 Translating iFSM to NuSMV

Our translation from iFSM to NuSMV aims to simultaneously simulate the state transitions and the corresponding iFSM variable modifications so that their agreement can be verified using temporal logic properties. Figure 6 shows the translation result from the iFSM specifications in Figure 3.

A key translation step from iFSM to NuSMV is to convert all iFSM values to integers with explicit lower/upper bounds, so that NuSMV can check all values against the proposed properties. To accomplish the conversion, we associate a unique integer with each unknown external memory reference (e.g., an event parameter or the result of a *new* operator). We then use these integers as values for iFSM variables that have non-integer types. For example, at line 15 of Figure 6, when responding to event *build*, the *next* value for obj is set to 1 because the expression $t.Clone()$ used to modify obj at lines 12-14 of Figure 3 has been associated with integer 1. For iFSM variables that already have an integer type but can hold an unlimited number of different values, we impose an artificial bound configured via command-line options. In Figure 6, this artificial bound is set to be 20. So both variables *val_count* and *val_copy_that_count* have a value range $-21..20$, where -21 is used to represent all values beyond $-20..20$. The translation approximation, as defined above, could potentially cause NuSVM to report failure with a counter example that does not exist in reality, thus making our verification conservative.

The rest of the translation simply maps each iFSM constituent to a corresponding NuSMV component. As illustrated by Figure 6, the resulting SMV code contains the following components.

Variables. Four groups of SMV variables are declared, illustrated at lines 1-3 of Figure 6.

- The $state$ variable, which is used to keep track of the control states of an iFSM object;
- The *internal* variables, each corresponding to a memory reference used inside the definition of an iFSM control state. In Figure 6, these variables are obj, $count$ and *val_count* (which represents $val(count)$) used at lines 4-6 of Figure 3.
- The *external* variables, each corresponding to a memory reference used in modifying the *internal* variables. In Figure 6, these variables are $copy\_that\_obj$, $copy\_that\_count$, and $val\_copy\_that\_count$, which correspond to $that.obj$, $that.count$, and $val(that.count)$ used by the event *copy* at lines 17-23 of Figure 3.
- The *event* variables, each corresponding to an iFSM event and used to keep track of the random invocation of each event. In Figure 6, these variables are *EQ*, *const_ref*, *reset*, *modify*, *copy*, and *build*, which are the events declared in Figure 3.

Initializations. The *state* variable is initialized with the default state of the iFSM, and the *internal* variables are initialized with their default iFSM values, as illustrated by line 4 of Figure 6. The *external* and *event* variables are not initialized, so that NuSMV will enumerate all possible values for them.

State Modification. The $state$ variable is modified based on which event is being invoked, the values of event parameters, and the previous value of $state$, as illustrated by lines 5-13 of Figure 6.

Internal Variable Modifications. The next value of each *internal* variable is computed based on the im-

```
GenNuSMV(ifsm,symTab)
1. traceThis = memory refs used in states_of(ifsm);
2. /* compute alias info. of pointer variables */
   for (each pointer variable x in traceThis):
     tracePtr[x] = stmts that modify x in transitions_of(ifsm);
     aliasMap[x] = memory refs aliased to x by stmts in tracePtr[x];
3. /*compute conditions and side effects of transitions*/
   for (each t in transitions_of(ifsm)):
     condMap[t] = condition(t) && condition(src_of(t));
     for (each memory ref x in traceThis):
       modMap[t][x]=stmts in t that modify x or aliasMap[x]
4. traceExt=empty; /*external memory refs to be traced by SMV*/
   for (each t in transitions_of(ifsm)):
     traceExt ∪= external_memory_refs(condMap[t]);
   for (each ref x in traceThis):
     traceExt ∪= external_memory_refs(modMap[t][x]);
5. /* generate SMV variable declarations*/
   for (each x in traceThis ∪ traceExt ∪ events(ifsm) ∪ {"state"}):
     gen_SMV_variable_declaration(symTab,ifsm,x);
6. /* generate initialization of state and internal variables */
   for (each x in traceThis ∪ {"state"}) :
     gen_SMV_variable_initialization(symTab,ifsm,x));
7. cases = empty; /* generate SMV state modification */
   for (each t in transitions_of(ifsm)):
     append_SMV_mod_case(cases,condMap[t],dest_states(t));
   gen_SMV_variable_modification("state",cases);
8. /* generate internal variable modifications */
   for (each memory reference x in traceThis):
     cases = empty;
     for (each t in transitions_of(ifsm)):
       append_SMV_mod_case(cases,condMap[t],modMap[t][x]);
     gen_SMV_variable_modification(x,cases);
9. for each s in states_of(ifsm) /* generate properties */
     gen_SMV_property(name_of(s),condition(s));
```

Figure 7: Algorithm for generating SMV code.

plementations of event-triggered transitions (i.e., how each transition modifies the *internal* variables), as illustrated by lines 14-24 of Figure 6.
LTL (Linear Temporal Logic) Properties to verify. This is discussed in Section 4.3.

4.2 The Translation Algorithm

Figure 7 shows our algorithm for translating iFSM specifications to NuSMV input. The algorithm takes two parameters, *ifsm*, the input iFSM specification to verify, and *symTab*, the symbol table of *ifsm*. The translation process includes the following steps.
Step(1): Collect Internal Variables, which are variables used in boolean expressions associated with the iFSM states. Save the result to variable *traceThis*.
Step(2): Compute Pointer Aliasing Information. For each pointer variable x in *traceThis*, extract all the memory references that can be aliased with x.
Step(3): Analyze Event-triggered Transitions. Here *condMap* maps each transition t to a boolean expression that controls its evaluation, and for each variable x in *traceThis*, $modMap[t][x]$ maps t to the new values it could assign to x. Note that *modMap* collects only the last value assigned to each *internal* variable without tracing intermediate modifications.
Step(4): Collect External Variables, which include all the external memory references that are used in expressions inside *condMap* or *modMap*. The collection is saved into the *traceExt* variable in Figure 7.
Step(5): Create SMV Variable Declarations. In addition to *state*, a variable is declared for each iFSM event and each item in *traceThis* or *tranceExt*.
Step(5-9): Generate SMV Code for Variable Initialization, Modification, and LTL Properties, as discussed in Sections 4.1 and 4.3.

4.3 Verifying Properties of iFSM

As illustrated by lines 25-30 of Figure 6, we generate an LTL property for each control state s to verify that at any time, the *state* variable equals to s if and only if the boolean expression associated with s in the original iFSM specification evaluates to true. Because the auto-generated SMV code separately simulates the state transitions and the corresponding iFSM memory modifications, the LTL properties essentially reconcile the results of both simulations, thereby verifying their consistency. Once the LTL properties are confirmed by the NuSMV model checker, the input iFSM is guaranteed to satisfy the following constraints.

1. The boolean expressions associated with different control states are mutually exclusive. Otherwise, the LTL properties would imply that the *state* variable have two different values simultaneously, which is a contradiction.
2. An iFSM object cannot possibly enter any state beyond those explicitly declared in the iFSM. Otherwise, since the SMV *state* variable always has a valid *enum* value, say s, the property relevant to $state = s$ would have failed the verification.
3. Immediately after initializing all member variables, an iFSM object is guaranteed to enter its declared default state. Otherwise, the LTL properties pertinent to the iFSM default state will fail.
4. After evaluating the implementation of each iFSM transition t, the resulting new values for the memory satisfy the boolean constraints associated with one of the declared destination states of t. If this is violated, the SMV verification will fail immediately as the value of the *state* variable no longer agrees with the values of the *internal* variables.

In summary, our verification algorithm will detect errors that cause an iFSM object to violate its declared runtime behavior. For example, if *val_count* in Figure 6 becomes < 1 at any point, e.g., due to the programmer forgetting to examine $val(count)$ before decrementing it, the verification will report the path that causes *val_count* to become out-of-sync.

5 EVALUATION

Our experimental evaluation aims to confirm two hypotheses regarding iFSM: (1) the language has the potential to conveniently specify most general-purose C++/Java classes, and (2) the auto-generated C++/Java class implementations are comparable to manually written code in terms of readability and efficiency. To verify these hypotheses, we have taken a number of existing manually written C++ classes and generated iFSM specifications for them. We then use our iFSM transformation engine to automatically generate equivalent C++/Java implementations from the iFSM specifications. This approach enables us both to look into the expressiveness of the language in terms of specifying randomly chosen existing C++ classes and to compare the quality of the auto-generated code with existing manual software implementations.

5.1 A Use Case Study

To verify the expressiveness of our iFSM language, we have taken a large, complex C++ class from an open-source compiler project, POET (Yi et al., 2007). We choose the POET project because its adaptive parser includes extremely complex control flow that was difficult to understand without documentation.

The single C++ class we took from POET is named *ParseMatchVisitor* and contains 490 lines of C++ code. This class inherits from two base classes and uses the visitor pattern to dynamically match syntax descriptions of an arbitrary language with a given input token stream. We have used iFSM to fully specify the behavioral logic and implementation strategies of this class, and have regenerated an equivalent class implementation from the iFSM. The resulting auto-generated C++ code has 540 lines and has confirmed many of our expectations of the iFSM language.

First, when using iFSM to specify each class method, we are required to consider both the overall side effects and the different runtime situations that may occur when invoking the method. Each situation is then specified using a state transition. The separate definitions of transitions and the explicit specification of their source and destination states serve to clearly document the semantic intention of each transition. The end result is a successful unification of the implementation with its higher-level behavioral design.

Second, cross-cutting concerns are made more explicit by iFSM. In particular, after specifying the beginning and ending states of all transitions, common traits of different events become easily noticeable. For example, all events of *ParseMatchVisitor* must return empty if the input stream is empty, and when the leading input token is matched against a designated syntax, all events must advance the token pointer.

Third, iFSM imposes some programming constraints which offer better coding structure. In particular, since each iFSM transition includes a straight line of statements as implementation, and *LOOP()* is the only statement that can introduce additional control flow, at most two levels of branches can be directly nested inside one another in the auto-generated code. If a deeper nesting of control-flow is required, additional methods must be introduced. Since complex control flow is a main source of confusion which reduces program readability, the auto-generated code is easier to understand than the original code.

When integrated within the POET project, the performance difference between the auto-generated *ParseMatchVisitor* class and the manually written one is indiscernible (below 0.01%). Therefore the coding structure difference had minimal performance impact.

5.2 Performance Comparison

Besides the use case study, we have used iFSM to generate a number of smaller C++/Java classes, including the $CountRefHandle$ class in Figure 3, two iterator classes named *SingleIterator* (which supports the iterator interface in Figure 2 for a single item) and $MultiIterator$ (which unifies two iterator interfaces into a single one), and two container classes named $Matrix$ and *SinglyLinkedList*. Both Java and C++ code are generated for each iFSM except for $CountRefHandle$, where only C++ code is generated because Java does not support memory deletion. All iFSMs are based on existing manually written C++ code. Therefore we compare the performance of auto-generated C++ code with that of the manual implementations. Figure 8 shows the result of comparison.

To test each C++ class, we construct a large number of class objects and then invoke the public methods of each object a constant number of times. Therefore, the runtime of each class implementation is proportional to the object container size, e.g., a matrix of size 500*500 or a singly-linked list with 500*500 items. We compiled all the C++ code using g++ 4.2.0 with -O2. The elapsed time of each evaluation is measured on an Intel 2.16 GHz Core2Duo processor with 1GB memory and 4MB L2 cache.

From Figure 8, the auto-generated C++ classes

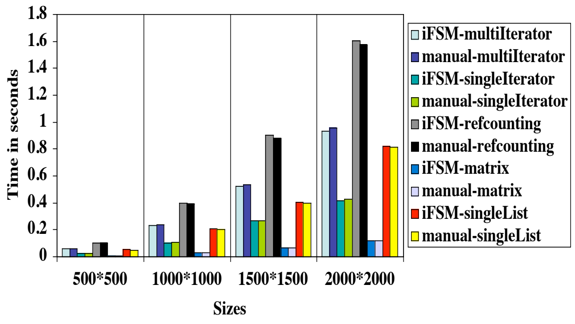

Figure 8: Performance using different input sizes.

performed similarly as the manually written ones. In particular, the auto-generated iterator classes performed slightly better than the manually written ones, while the auto-generated reference counting and singly-linked list classes performed slightly worse. The matrix classes performed almost identically.

Since the differences in performance are minor between the iFSM-generated class implementations and the manually written ones, they are likely caused by random factors in the compiler. The main benefit to gain from the iFSM specifications is the automatic verification of consistency between implementation details and behavior design, and the potential of the iFSM compiler utilizing the behavior information to improve the efficiency of their interactions.

6 RELATED WORK

Model-driven development (Kleppe et al., 2003) captures important aspects of a software system through models (Goguen and Burstall, 1992; Gray et al., 2001) before producing lower-level implementations of the system (Balasubramanian et al., 2005; Kogekar et al., 2006). In particular, FSM-based notations have long been used in previous research to model the dynamic behavior of large reactive systems (Harel, 1987; Harel and Naamad, 1996), and many research projects have automatically produced code to simulate the behavior of various state machine models (Knapp and Merz., 2002; Prout et al., 2008; Whalen, 2000).

Runtime behavior modeling, however, is only one aspect of software development. Unless the behavioral notations are correlated with other aspects of software implementation, e.g., data structures and algorithms, the behavioral notations are merely artifacts of the software design phase and have to be kept separate from complete implementations of software systems. Working towards similar goals, Poizat *et. al.* have developed a semi-automated approach to generate Java code from both data and behavioral models (Poizat et al., 1999). Our iFSM language offers a way to unify modeling notations and implementation strategies so that they can be maintained together, and their agreement can be automatically verified.

By unifying behavior modeling with domain-specific implementation specifications, previous research has produced efficient finite-state-machine implementations in some specialized domains, e.g., embedded systems (Wasowski, 2003) and lexer/parser generation (Levine et al., 1992). Our work is different from these domain-specific code generators in that we target general-purpose object-oriented C++/Java code, and we automatically verify the consistency between the implementation specifications and the corresponding behavioral design.

Program transformation tools have long been used to analyze and modify existing software implementations, including re-documenting/re-implementing code, reverse engineering, and porting to new platforms (Baxter et al., 2004; Futamura et al., 2002). Several general-purpose transformation languages and systems have been developed (Huang et al., 2005; Erwig and Ren, 2002; Bravenboer et al., 2008; Bagge et al., 2003) and some have been widely adopted (Bravenboer et al., 2008; Bagge et al., 2003). These tools and systems do not use modeling notations and are typically not concerned with the consistency between software design and implementation. We focus on combining program transformation with software verification to better support both the correctness and the efficiency of generated code.

Formal methods have been widely used to verify both software designs and implementations (Clarke et al., 1999; Owre et al., 1992; Chaki et al., 2004; Necula, 1997; Das and et. al., 2002; Kawaguchi et al., 2009). A number of projects can effectively verify important properties or generate test cases of software systems via analyzing the source code (Beyer et al., 2004; Das and et. al., 2002; Chaki et al., 2004). While more appealing, directly verifying low-level software implementations is in general extremely challenging due to the unlimited memory references dynamically modified by user applications. Our iFSM language explicitly categorizes the unlimited memory modifications using a finite number of runtime state transitions. As a result we can more readily bridge the semantic gap between the behavior properties and the implementation details. For example, if a variable *x* is modified within a transition, a small set of unique expressions can be determined to be the new values for *x*, which is typically not possible when directly verifying programs in lower-level languages such as C++/Java. The iFSM annotations may be embedded inside existing implementations to enable more effective verification, but this belongs to our future work.

7 CONCLUSIONS

We present a high-level specification language, iFSM, to effectively unify the behavior design of object-oriented classes with detailed implementation strategies. We have automatically generated efficient C++/Java code from iFSM specifications and have automatically verified their consistency via model checking techniques.

REFERENCES

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

Balasubramanian, K., Krishna, A. S., Turkay, E., Balasubramanian, J., Parsons, J., Gokhale, A., and Schmidt, D. C. (2005). 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*.

Baxter, I., Pidgeon, P., and Mehlich, M. (2004). Dms: Program transformations for practical scalable software evolution. In *Proceedings of the International Conference on Software Engineering*. IEEE Press.

Beyer, D., Chlipala, A. J., and Majumdar, R. (2004). Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 326–335.

Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. (2008). Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*.

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

Cimatti, A. and et. al. (2002). NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *CAV*, volume 2404 of *LNCS*.

Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. MIT Press.

Das, M. and et. al. (2002). Esp: path-sensitive program verification in polynomial time. In *PLDI '02*, pages 57–68.

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

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

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

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

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Comp. Prog.*, 8(3).

Harel, D. and Naamad, A. (1996). The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333.

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

Kawaguchi, M., Rondon, P., and Jhala, R. (2009). Type-based data structure verification. In *PLDI '09*, pages 304–315.

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

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

Kogekar, A., Kaul, D., Gokhale, A., Vandal, P., Praphamontripong, U., Gokhale, S., Zhang, J., Lin, Y., and Gray, J. (2006). 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.

Levine, J. R., Mason, T., and Brown, D. (1992). *Lex & Yacc*. O'Reilly & Associates.

Necula, G. C. (1997). Proof-carrying code. In *POPL'97*, pages 106–119.

Niu, J., Atlee, J. M., and Day, N. A. (2003). Template semantics for model-based notations. *IEEE Transactions on Software Engineering*, 29(10):866–882.

Owre, S., Rushby, J., and Shankar, N. (1992). PVS: A prototype verification system. In *CADE*.

Poizat, P., Choppy, C., and Royer, J.-C. (1999). From informal requirements to coop: A concurrent automata approach. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II*, pages 939–962.

Prout, A., Atlee, J. M., Day, N. A., and Shaker, P. (2008). Semantically configurable code generation. In *MoDELS*, pages 705–720.

Wasowski, A. (2003). 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. ACM.

Whalen, M. W. (2000). High-integrity code generation for state-based formalisms. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 725–727, New York, NY, USA. ACM.

Yi, Q., Seymour, K., You, H., Vuduc, R., and Quinlan, D. (2007). POET: Parameterized optimizations for empirical tuning. In *Workshop on Performance Optimization for High-Level Languages and Libraries*.