Extracting FSMs from Object-Z Specifications with History Invariants

Jun Sun
National University of Singapore
School of Computing
3 Science Drive 2, Singapore
sunj@comp.nus.edu.sg

Jin Song Dong
National University of Singapore
School of Computing
3 Science Drive 2, Singapore
dongjs@comp.nus.edu.sg

Abstract

Object-Z with history invariants can present precise and abstract models for complex systems. The system behavior patterns are often implicitly embedded within various state/operational constraints and history invariants. Without explicit system behavior representations, it is difficult to implement those abstract models. In this paper, we present a sound and systematic approach to automatically extract explicit system behaviors (as FSMs) from the abstract Object-Z specifications. Safety and liveness and additional crucial requirements for open systems are ensured.

Keywords: Object-Z, FSMs, Software Specification

1 Introduction

The notion of separation of concerns is a common technique to fight complexity in system development. A practical approach is to focus on system functionalities in the early stage of system design. An early stage abstract model typically contains a set of objects/classes, data variables and the associated abstract operations in each class. Those models can be perfectly documented as Object-Z [32, 28] specifications.

Object-Z with history invariants can present precise and abstract models for complex systems. A system design in Object-Z is relieved from behavioral aspects of the system by implicitly embedding system behavior patterns within state/operational constraints and, additionally, history invariants. However, without explicit system behavior representations, it is difficult to implement such abstract models. In this paper, we present a sound and systematic approach to automatically extract explicit implementable system behaviors as classic Finite State Machines (FSMs) from abstract Object-Z specifications. The ultimate goal of our work is to generate implementations from high-level designs in Object-Z automatically and, moreover, guarantee all critical requirements satisfied.

An Object-Z specification captures safety requirements by specifying class invariants and pre/post-condition for data operations. Liveness requirements are captured by history invariants. We generate FSMs that are guaranteed to satisfy both sets of requirements. Additionally, because Object-Z distinguishes external variables (variables attached with a question mark) from state variables, it can be used to model open systems. Crucial requirements for open systems are also to be ensured by the generated FSMs, i.e. the FSMs should not introduce fresh deadlocks and should work correctly in any environment. We call such FSMs as realizations of the Object-Z specification.

To handle Object-Z specifications modelling systems with infinite data space, we developed a predicate abstraction schema to build a raw FSM from an Object-Z specification. The number of abstract states are bounded by the number of predicates for abstraction. A weak abstract relation is used so that the abstraction is automated by general theorem provers like PVS [22] paying a reasonable price. The raw FSM is then refined to satisfy the additional requirements. Finally, an Object-Z specification is realized as an FSM with its transitions as guarded function calls. The soundness is proved by showing that there is a fair simulation relation from the realization to the specification. A tool is implemented in JAVA to experiment our method.

The reason why our approach is beneficial is twofold. Firstly, FSMs are more close to implementations than Object-Z specifications, i.e. FSMs are implementable. In our setting, a complete implementation of the system may be generated if the implementation of each operation in isolation (probably by other programmers) is supplied. This conforms one of the principles of object-oriented analysis and design, i.e. procedural thinking should be postponed as long as possible. Secondly, our realization is "minimally" restrictive so that further refinements are possible without breaking any of the requirements.

The rest of the paper is organized as follows. Section 2 introduces the Object-Z specification language. Section 3 presents a systematic way of building a finite state model from an Object-Z specification using predicate abstraction. Section 4 shows how to realize an Object-Z specification as an FSM. Section 5 proves the soundness of the approach. Section 6 briefly introduces a tool for experiments. Section 7 addresses related works and issues.

2 Object-Z

Z [32] is a state-based formal specification language based on the established mathematics of set theory and first-order logic. It has been used to specify a wide range of systems including transaction processing systems and communication protocols. A specification in Z typically consists of a number of state and operation schemas. A state schema groups together variables and defines the relationship that holds between their values. An operation schema defines the relationship between the 'before' and 'after' valuations of one or more state schemas. Object-Z [8] is an object-oriented extension of the Z language. It improves the clarity of large Z specifications through enhanced structuring. The main Object-Z construct is *class* definitions, which captures the object-oriented notion of a class by encapsulating a single state schema with all the operations which may affect its variables.

$[Package]$

Queue

$\upharpoonright(\mathrm{INIT}, Join, Leave)$

$expires : Package \rightarrow Boolean$

$items : \mathrm{seq}\, Package$

INIT

$items = \langle\ \rangle$

Join

$\Delta(items)$

$item? : Package$

$expires(item?) \Rightarrow items' = items$

$\neg\, expires(item?) \Rightarrow items' = items \frown \langle item? \rangle$

Leave

$\Delta(items)$

$item! : Package$

$items = \langle item! \rangle \frown items'$

An Object-Z class is represented syntactically as a named box with zero or more generic parameters. There may be local types and constant definitions, at most one state schema and associated initial state schema and zero or more operations. The declarations of the state schema are referred to as the state variables and the predicate as the class invariant. The class invariant restricts the possible valuations of the state variables. An initial schema identifies the possible initial valuations of the state schema. An operation is either an operation schema or a schema expression involving existing class operations and schema operators. The above is an Object-Z specification of a queue class. A package is modelled as a given type. The queue is modelled as a sequence of packages which is initially empty. Operations are provided to allow items to join or leave the queue on a first-in/first-out basis. This queue may be viewed as an incoming channel of a network router. A total function *expires* is used to tell whether a package is expired (by examining certain flag bit in the package). A package is enqueued and later forwarded if and only if it is not expired.

The various operations in a class are given a standard Z semantics, which is used to develop a transition-system semantics. The Z operation semantics is best viewed as describing a relation between initial and final states for each operation. The Z precondition of an operation schema describes the initial states for which there exists some final state satisfying the schema predicate. If the state schema of the class is denoted as *State*, and *inputs* (*outputs*) is the list of inputs (outputs) associated with the operation, then the precondition is defined as:

$$\mathrm{pre}\, Operation \\ \quad \widehat{=}\ \exists\, State';\ outputs \bullet Operation \setminus outputs$$

Similarly, the postcondition of an operation at a state ($State_a$) where the precondition is satisfied is defined as:

$$\mathrm{post}(Operation, State_a) \\ \quad \widehat{=}\ \exists\, State;\ inputs \bullet State_a \wedge Operation \setminus outputs$$

The operations of a class thus form a named collection of relations, which determines a transition system in which a given operation may fire exactly when its Z precondition is satisfied. The semantic model for the class consists of all the sequences of operations/events which can be performed by objects of the class. For instance, the class *Queue* effectively defines the state transition system in Figure 1.

The properties represented by a state transition system is referred as safety properties. They specify which state changes may occur but do not require that any state changes actually do occur. Properties which state that a state change, or an operation, must occur are referred to as liveness properties. Object-Z allows the specification of liveness properties by associating with each class a history invariant as a temporal logic formula[1]. The history invariant restricts the set of histories derived from the state of the class. In this work, we handle history invariants expressed in Linear-time

[1]History invariants are introduced in early versions of Object-Z language [27, 9]. However, it is not included in the works of G. Smith [28] for practical reasons. We believe that history invariants are an effective method to strengthen the weak process control logic of Object-Z.

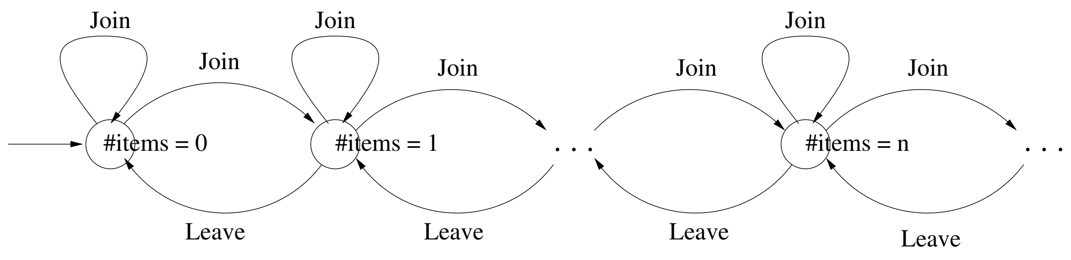

Figure 1. A Transition System Interpretation of *Queue*

Temporal Logic (LTL [23]). The history invariants other than standard LTL formulae appear in [27, 9] can be reframed in LTL by introducing auxiliary variables. The following class models a subclass of *Queue* where the queue is guaranteed to be empty eventually and the number of items in the queue is bounded by *max*. Informally, $\Box$ can be interpreted as 'always' and $\Diamond$ as 'eventually'.

FairBoundedQueue
Queue

$max : \mathbb{N}$

$\Box \Diamond \#items = 0$
$\Box \#items \leq max$

3 Predicate Abstraction

The state space of an Object-Z class may be infinite. For example, a *Queue* object may contain infinite number of items. However, an implementable control structure may only contain a finite number of control states. It restricts the behaviors of an object (specified by an Object-Z class) based abstract interpretations of the data variables. Figure 1 is an abstract interpretation of *Queue* objects as only the number of items (not the actually content) in the queue is concerned. We present a method to calculate an predicate abstraction of an Object-Z class. A weak abstract relation is used so that we may construct an abstract state graph automatically by paying a reasonable price. In the rest of the paper, we use state and predicate interchangeably.

Let $[Predicate]$ denote all possible predicates. Given a finite set $AP \subset Predicate$ for abstraction. We denote the set of abstract states as S_a.

$$S_a \mathrel{\widehat=} \{z : Predicate \mid \exists X : \mathbb{P} AP \mid z = \bigwedge (X \cup \{\neg e : Predicate \mid e \in AP \setminus X\})\}$$

Informally, an abstract state is a state where a subset of AP and the negation of the rest are true. We define a function $\mathcal{W}$ to calculate the weakest formula over AP that implies a predicate e, i.e. $\mathcal{W}(e)$ is the disjunction of all formulae $p \in AP$ where $p \Rightarrow e$.

$$\mathcal{W} : Predicate \rightarrow Predicate$$
$$\forall e : Predicate \bullet \mathcal{W}(e) = \bigvee \{x \in S_a \mid x \Rightarrow e\}$$

Informally, function $\mathcal{W}$ calculates the set of abstract states where a certain predicate is true. However, such a function in our context is undecidable. i.e. we may not be able to tell if a predicate is true at a state due to limited power of proving. Therefore, we define a function $\mathcal{S}$ to calculate all states where a certain predicate might be true.

$$\mathcal{S} : Predicate \rightarrow Predicate$$
$$\forall e : Predicate \bullet \mathcal{S}(e) = \bigvee (S_a \setminus \mathcal{W}(\neg e))$$

Function $\mathcal{S}$ works by pruning all states where the predicate is proved to be false. Thus, all states where the predicate is true are present in the result (probably with states where we are uncertain if the predicate is true). Function $\mathcal{S}$ is used to automatically construct an abstraction of an Object-Z specification.

Let $AP \mathrel{\widehat=} \{\#items = 0, \#items \leq max\}$. The set of abstract states is (assuming $max > 0$):

$$S_a \mathrel{\widehat=} \{\#items = 0, max \geq \#items > 0, \#items > max\}$$

The abstract initial state of *Queue* class is $\mathcal{S}(\text{INIT}) \mathrel{\widehat=} \#items = 0$. We calculate an abstraction of an operation by abstracting the precondition and postcondition. The precondition is replaced by $\mathcal{S}(\text{pre}\, Operation)$, i.e. all abstract states where the operation might be applied. For instance, the abstract precondition of operation *Leave* is:

$$\begin{aligned}
&\mathcal{S}(\text{pre } Leave)\\
&\quad \mathrel{\widehat=} \mathcal{S}((\exists\, items' : \text{seq}\, Package;\ item! : Package \bullet\\
&\qquad\quad items = \langle item! \rangle \frown items') \setminus \{item!\}) \quad \text{[def. of pre]}\\
&\quad \mathrel{\widehat=} \bigvee (S_a \setminus \mathcal{W}((\forall\, items' : \text{seq}\, Package;\ item! : Package \bullet\\
&\qquad\quad items \neq \langle item! \rangle \frown items') \setminus \{item!\})) \quad \text{[def. of } \mathcal{S}]\\
&\quad \mathrel{\widehat=} \bigvee (S_a \setminus \{\#items = 0\}) \quad \text{[def. of } \mathcal{W}]\\
&\quad \mathrel{\widehat=} \#items > 0
\end{aligned}$$

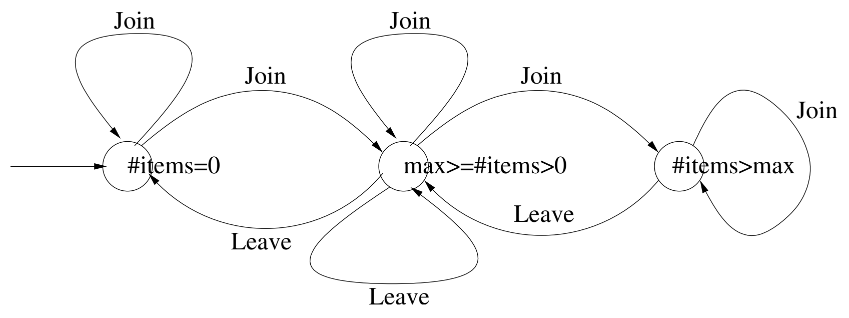

Figure 2. An Abstract Transition System Interpretation of *Queue*

Therefore, operation *Leave* is applicable only at the two abstract states where $\#items > 0$. For each abstract state $s_a : S_a$ satisfying the abstract precondition, we calculate the abstract postcondition as $\mathcal{S}(\text{post}(Operation, S_a))$. For example, the postcondition of operation *Leave* at the abstract state where $max \geq \#items > 0$ is:

$$\begin{aligned}
&\text{post}(Leave, max \geq \#items > 0)\\
&\quad \mathrel{\widehat{=}} \mathcal{S}((\exists\, items : \text{seq}\, Package \bullet\\
&\qquad \#items \leq max \land \#items > 0 \land\\
&\qquad items = \langle item! \rangle \frown items') \setminus \{item!\})\ \text{[def. of post]}\\
&\quad \mathrel{\widehat{=}} \bigvee S_a \setminus \mathcal{W}((\forall\, items : \text{seq}\, Package \bullet\\
&\qquad \#items > max \lor \#items \leq 0 \lor\\
&\qquad items \neq \langle item! \rangle \frown items') \setminus \{item!\})\ \text{[def. of } \mathcal{S}]\\
&\quad \mathrel{\widehat{=}} \bigvee S_a \setminus \{\#items' > max\} \qquad \text{[def. of } \mathcal{W}]\\
&\quad \mathrel{\widehat{=}} max \geq \#items' \geq 0
\end{aligned}$$

We abstract every operation in the class. The abstraction of the *Queue* class defines the state transition system in Figure 2. Note that abstraction introduces non-determinism and spurious sequence of operations. For example, a *Join* operation at the state in the middle may result in a state where the number of items in the queue is larger than *max* or no larger than *max*. The abstraction can be automated with the help of *PVS* [22]. Despite the prove power, an abstract state transition system covers all possible sequences of operations of the concrete one.

4 Generating Finite State Machines

An FSM is an abstract machine that has only a finite constant amount of memory. It can be viewed as a flattened UML State Diagram. There are finite many states and each state has transitions to states. Transitions are triggered by observable events. Additionally, there are one or more initial states and final states.

Definition 4.1 *An FSM is a 6-tuple $(S, T, I, F, \Sigma, \mathcal{L})$ where S is a finite set of states, $T : S \times \Sigma \times S$ is a labelled transition relation, $I : \mathbb{P}\, S$ is a set of initial states, $F : \mathbb{P}\, S$ is a set of final states, Σ is the alphabet and $\mathcal{L} : S \to \mathbb{P}\, S$ is a labelling function which labels a state with a set of observations (predicates).*

The language accepted by an FSM contains all traces of the state machine which end up with a final state. An FSM is a realization of an Object-Z specification if the initial schema is satisfied, every operation is engaged with its precondition/postcondition fulfilled and the history invariants are satisfied. For open systems, two additional requirements are crucial.

A_1: The FSM should not introduce any fresh deadlocks.

A_2: The FSM is not allowed to restrict the actions of the environment.

Both requirements have been discussed in various works of control theory [26, 17]. The first requirement is commonly referred as nonblocking. The second requirement is essential for systems constantly interacting with its environment. Informally, it requires that the state machine should be able to function correctly regardless of the environment. In this section, we present a systematic way of generating FSMs that satisfy both the Object-Z specification and the two additional requirements.

4.1 Generating Raw State Machines

Our method begins with constructing a finite Büchi automaton from the history invariant[2]. An efficient tool to convert LTL formulae into optimized Büchi automata is Somenzi and Bloem's Wring [30]. For example, Figure 3 shows the Büchi Automata constructed from the LTL formulae in the *FairBoundedQueue* class. Both states are initial states. The state labelled with $\#items = 0$ is a final state. Note that transitions are not labelled.

[2]In general, our implementation allows terminating behaviors. This is different from the languages accepted by Büchi automata. We keep the name Büchi Automata so as to honor works on generating finite state realization from temporal logic formulae. However, we treat it as finite state machines with no label on the transitions.

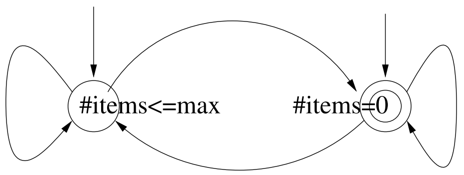

Figure 3. A Büchi Automata Example

Definition 4.2 *A Büchi automata is a 5-tuple $(S, T, I, F, \mathcal{L})$ where S is a finite set of states, $T : S \times S$ is a transition relation, $I : \mathbb{P}\, S$ is a set of initial states, $F : \mathbb{P}\, S$ is a set of final states and $\mathcal{L} : S \to \mathbb{P}\, S$ is a labelling function which labels a state with a set of predicates.*

Meanwhile, a raw FSM is constructed from an Object-Z specification as discussed in section 3. We require that the predicates for abstraction include propositions in the history invariants and the initial schema. Every state in the raw state machines is a final state[3]. The product of the state machine and the Büchi automata is then constructed.

Definition 4.3 *A state machine $(S, T, I, F, \Sigma, \mathcal{L})$ is a product of a state machine $(S_s, T_s, I_s, F_s, \Sigma_s, \mathcal{L}_s)$ and a Büchi automata $(S_b, T_b, I_b, F_b, \mathcal{L}_b)$ if it satisfies the following condition:*

- $S \mathrel{\widehat=} \{(s_s, s_b) : S_s \times S_b \mid \bigwedge \mathcal{L}_s(s_s) \Rightarrow \bigwedge \mathcal{L}_b(s_b)\}$
- $T \mathrel{\widehat=} \{((s_s^1, s_b^1), e, (s_s^2, s_b^2)) : S \times \Sigma \times S \mid (s_s^1, e, s_s^2) \in T_s \wedge (s_b^1, s_b^2) \in T_b\}$
- $I \mathrel{\widehat=} \{(i_s, i_b) : I_s \times I_b \mid \bigwedge \mathcal{L}_s(i_s) \Rightarrow \bigwedge \mathcal{L}_b(i_b)\}$
- $F \mathrel{\widehat=} \{(f_s, f_b) : I_s \times I_b \mid \bigwedge \mathcal{L}_s(f_s) \Rightarrow \bigwedge \mathcal{L}_b(f_b)\}$
- $\Sigma \mathrel{\widehat=} \Sigma_s$
- $\mathcal{L} \mathrel{\widehat=} \mathcal{L}_s$

Informally, a state in the Büchi automata is unified with a state in the state machine if their labelling is consistent. Note that because all predicates in the history invariant are used for abstraction, the consistency of two states is a straightforward existence check, i.e. whether the set of predicates labelled with a state is a subset of those of the other state. A state of the product is an initial state if and only if it is unified by two initial states. A labelled transition in the raw state machine is allowed in the product if and only if there is a transition between the same starting state and ending state in the Büchi Automata. For instance, Figure 4 is the product of the state machine in Figure 2 and the Büchi automata in Figure 3.

[3]In Object-Z semantics, an object may wait infinitely long before engaging an enabled operation.

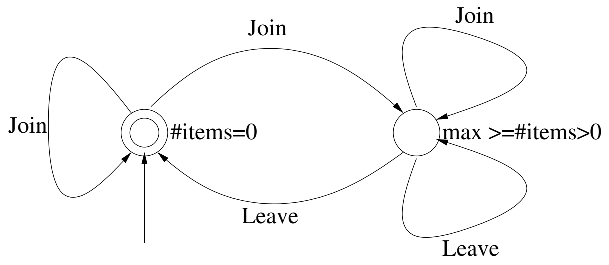

Figure 4. Product of the FSM and Automata

4.2 Pruning Raw State Machines

The product of the raw state machine and the Büchi automata satisfies the Object-Z specification with the history invariant. However, it may not be a valid realization of the Object-Z specification. There are two sources of possible errors. Firstly, because of requirement A_2, any realization satisfies the specification by assuming behaviors of the environment is not valid. For example, a possible realization of *FairBoundedQueue* is the state machine in Figure 5. By requiring that all *item*? from the environment are expired, the queue remains empty all the time and, therefore, satisfies the history invariant. We prune such realizations from our state machine by pruning states and transitions violating requirement A_1 and A_2. Secondly, abstraction introduces spurious sequences of events/operations, e.g. an operation may be engaged at states where its precondition is not satisfied or an invocation of the operation may lead to different states nondeterministically. To solve the problem, each transition is equipped with a guard condition (if necessary) in the last step.

The operations enabled at a state are partitioned into two sets, controllable operations and uncontrollable operations. An operation is uncontrollable at a state if its postcondition depends on environmental inputs. For example, the *Join* operation at the initial state of the state machine in Figure 4 is uncontrollable. An operation at a state is controllable if it is not uncontrollable. Formally, A_1 and A_2 are:

$$\begin{array}{l}
\mathbf{A_1} : \forall s_s^1, s_s^2 : S_s;\ s_b^1, s_b^2 : S_b;\ e : \Sigma \bullet \\
\quad (s_s^1, e, s_s^2) \in T_s \Rightarrow \\
\qquad (\exists e' : \Sigma;\ (s_s^3, s_b^3) : S \bullet \\
\qquad\quad ((s_s^1, s_b^1), e', (s_s^3, s_b^3)) \in T) \\
\mathbf{A_2} : \forall s_s^1, s_s^2, s_s^3 : S_s;\ s_b^1, s_b^2 : S_b;\ e : \Sigma \bullet \\
\quad ((s_s^1, s_b^1), e, (s_s^2, s_b^2)) \in T \wedge (s_s^1, e, s_s^3) \in T_s \wedge \\
\qquad e \text{ is uncontrollable at the state} \Rightarrow \\
\qquad\quad (\exists s_b^3 : S_b \bullet ((s_s^1, s_b^1), e, (s_s^3, s_b^3)) \in T)
\end{array}$$

We use a polynomial time algorithm to prune states and transitions violating either of the requirements from the product. A state is pruned if and only if it is a fresh deadlock state. Transitions at a state labelled with the same operation

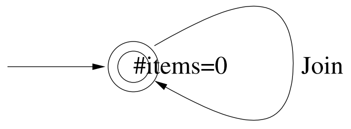

Figure 5. An Invalid Realization

are pruned at once if the operation is uncontrollable and A_2 is violated. Let *Current* be a set of immediate successor states of a state, *Path* be the set of states in the path from an initial state to the state (inclusive). *Path* is used to guarantee convergence of the procedure. Let *Product*, *Raw* be the two state machines respectively. Additional, *Done* holds states that are not to be pruned.

```
procedure Prune (Current, Path, Product, Raw)
 1:  if Current is empty
 2:    return true                                   [Case 1]
 3:  else
 4:    if all states in Current are in either Path or Done
 5:      return true                                 [Case 2]
 6:    else
 7:      pick a state s from Current and not in Path or Done
 8:      for all uncontrollable events e at s
 9:        if e at s satisfies $A_2$
10:          continue with line 8
11:        else
12:          prune all transitions labelled with e
13:        endif
14:      endfor
15:      if s satisfies $A_1$
16:        continue with line 21
17:      else
18:        prune s from Product
19:        return false                              [Case 3]
20:      endif
21:      let Children be the immediate successors of s
22:      Add s to Path
23:      if Prune (Children, Path, Product, Raw) is true
24:        add s to Done and continue with line 4    [Case 4]
25:      else
26:        continue with line 8                      [Case 5]
27:      endif
28:    endif
29:  endif
```

Our algorithm takes a Depth-First-Search strategy. Initially, *Current* is the set of initial states, *Path* is an empty set. There is a realization of the Object-Z specification if and only if the pruned state machine has at least one initial state and one reachable final state. The correctness of the algorithm is an immediate consequence of the fact that a state is not pruned if it satisfies both requirements and all reachable states from it are not pruned. The algorithm converges because the states and transitions are finite and the size of *Done* is monotonically increasing.

If we specify the history invariant for *Queue* as $\square(\#items = 0)$, the product of the raw state machine and the Büchi automata is the FSM in Figure 5. After the pruning there is no initial and final state left (the transition is pruned because of violation of A_2 and the state is pruned because of violation of A_1). Therefore, we conclude that there is no realization for such a specification.

4.3 Calculating Guard Condition

The last step is to calculate a proper guard condition for each transition. A guarded transition can be engaged only when its guard condition is satisfied. A guard condition guarantees that an operation is applied only when its precondition is satisfied. Moreover, part of a nondeterministic choice may get pruned in the pruning process. The remaining transitions are, therefore, constrained by restricting its postcondition. This is not directly implementable. Thus, a state guard is use to make sure that a transition is engaged only when it will reach the desired postcondition.

Let $\mathcal{WP}$ denote the weakest precondition operator introduced in [6]. Given an operation *Operation* and a source state s_a and a state s_b that can be reached from s_a by engaging *Operation*, the weakest precondition is defined as[4]:

$$\begin{aligned}&\mathcal{WP}(Operation, s_a, s_b)\\&\quad\widehat{=}\ (\exists\, State';\ outputs \bullet Operation) \wedge\\&\qquad(\forall\, State';\ outputs \bullet (Operation \wedge s_a) \Rightarrow s_b)\end{aligned}$$

The first part of the condition guarantees the termination of the operation. The second part guarantees the postcondition. For example, the guard condition for *Join* operation at the initial state of the state machine in Figure 4 to remain at the same state is:

$$\begin{aligned}&\mathcal{WP}(Join, \#items = 0, \#items = 0)\\&\ \widehat{=}(\exists\, items' : \mathrm{seq}\, Package \bullet\\&\qquad(expires(item?) \Rightarrow items' = items) \wedge\\&\qquad(\neg\, expires(item?) \Rightarrow items' = items \frown \langle item? \rangle)) \wedge\\&\quad(\forall\, items' : \mathrm{seq}\, Package \bullet (\#items = 0 \wedge\\&\qquad(expires(item?) \Rightarrow items' = items) \wedge\\&\qquad(\neg\, expires(item?) \Rightarrow items' = items \frown \langle item? \rangle)) \Rightarrow\\&\qquad\quad \#items' = 0) \qquad\qquad [\text{def. of } \mathcal{WP}]\\&\ \widehat{=}\ \forall\, items' : \mathrm{seq}\, Package \bullet\\&\qquad((expires(item?) \wedge items' = \langle\,\rangle) \vee\\&\qquad(\neg\, expires(item?) \wedge items' = \langle item? \rangle)) \Rightarrow\\&\qquad\quad \#items' = 0 \qquad\qquad [\text{one-point rule}]\\&\ \widehat{=}\ \forall\, items' : \mathrm{seq}\, Package \bullet \#items' = 0 \vee\\&\qquad expires(item?) \vee items' = \langle item? \rangle\\&\ \widehat{=}\ expires(item?)\end{aligned}$$

[4]A similar problem on the weakest precondition semantics of Z is addressed in [3]

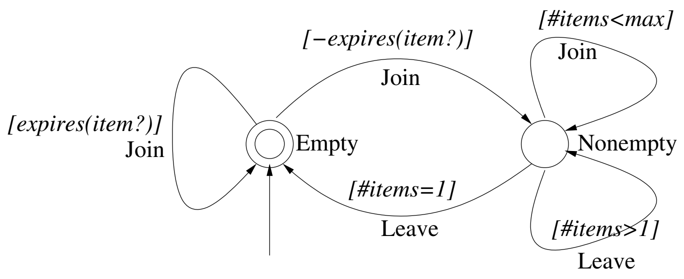

Figure 6. A Realization of *FairBoundedQueue*

Thus, the transition is guarded with $expires(item?)$. If the weakest precondition turns out to be *false*, it means that there is no way that we can guarantee that the transition ends up with the desired state. This is normally due to internal nondeterminism, i.e. some information is not present at the abstract level. Such transitions are pruned. The pruned state machine for *FairBoundedQueue* is in Figure 6. Note that states are labelled with names to improve readability.

4.4 A Case Study

We use a multiplexer example to show how our method works for composed classes. A multiplexer is made up of three bounded queues, two as incoming channels and one as outgoing channel. It can be viewed as a network router which gets packages from different sources and forwards those that are not yet expired. All packages in the incoming channels are eventually forwarded to the outgoing channel.

Multiplexer

$\upharpoonright(\text{INIT}, Join_1, Join_2, Transfer_1, Transfer_2, Leave)$

$input_1, input_2 : FairBoundedQueue$
$output : Queue$

INIT

$input_1.\text{INIT} \wedge input_2.\text{INIT} \wedge output.\text{INIT}$

$Join_1 \mathrel{\widehat=} input_1.Join$
$Join_2 \mathrel{\widehat=} input_2.Join$
$Transfer_1 \mathrel{\widehat=} input_1.Leave \parallel output.Join$
$Transfer_2 \mathrel{\widehat=} input_2.Leave \parallel output.Join$
$Leave \mathrel{\widehat=} output.Leave$

$\Box \# output.items \leq output.max$

The history invariants include those inherited from the *FairBoundedQueue*. The predicates for abstraction include those in the history invariant and initial schema. It is:

$$AP \mathrel{\widehat=} \{\#input_1.items = 0, \#input_2.items = 0, \\ \#input_1.items \leq input_1.max, \\ \#input_2.items \leq input_2.max\}$$

Only operations defined or promoted in this class are concerned. For operations composed using operation operators, the process of calculating preconditions and postcondition can be simplified by considering the structure of an operation (refer to chapter 14 in [32]). Note that an uncontrollable operation may become controllable when the object composes with other objects. For example, operation *output.Join* is initially uncontrollable (at all states) when we consider *Queue* class along. It becomes controllable as in operation *Transfer* because all packages from either of the incoming channels are not expired. The final FSM realized from *Multiplexer* is in Figure 7. The step-by-step construction is omitted due to the limit of space.

5 Soundness

A state machine is a realization of an Object-Z specification if and only if it satisfies the following condition:

- All operations are engaged when its precondition and postcondition are satisfied. $[A_3]$
- All possible sequence of operations satisfies the history invariant. $[A_4]$
- A_1 and A_2.

A_3 is guaranteed by guarding each transition with a condition stronger than its precondition (the weakest precondition). In the process of pruning the product, all fresh deadlock states and transitions violating A_2 are pruned. It is straightforward to verify that both A_1 and A_2 are satisfied. To prove A_4, we show that there is a fair simulation relation from our realization to the product of the state transition system defined by an Object-Z specification and the Büchi automata representing its history invariant (the specification).

Definition 5.1 *Let $M_i \mathrel{\widehat=} (S_i, T_i, I_i, F_i, \Sigma_i, \mathcal{L}_i)$ where $i \in \{1,2\}$ be two state machines. A total relation $\mathcal{R} : S_1 \rightarrow S_2$ is a fair simulation from M_1 to M_2 if it satisfies the following condition:*

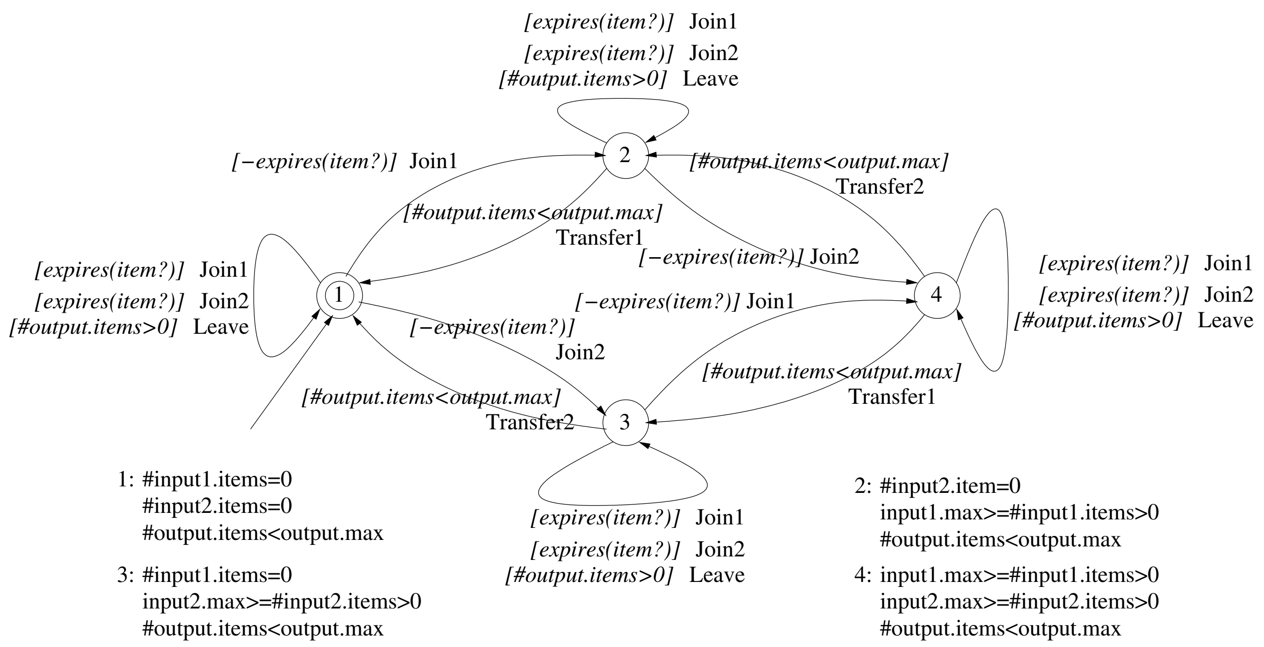

Figure 7. A Realization of *Multiplexer*

D_1: $\forall s : I_1 \bullet \mathcal{R}(s) \in I_2$
D_2: $\forall (s_1, e, s_2) \in T_1;\ s'_1 : S_2 \mid \mathcal{R}(s_1) = s'_1 \bullet$
$\quad \exists s'_2 : S_2 \bullet (s'_1, e, s'_2) \in T_2 \wedge \mathcal{R}(s_2) = s'_2$
D_3: $\forall s : F_1 \bullet \mathcal{R}(s) \in F_2$

Informally, D_1 states that there is an initial state in M_2 for every initial state in M_1. D_2 states if the FSM can engage an event at a state in M_1, M_2 should be able to simulate the transition at the corresponding state. D_3 guarantees that all final states in M_1 are simulated in M_2. A similar definition appears in [7] and later development can be found in [12]. If there is a fair simulation relation from M_1 to M_2, then M_2 fair trace-contains M_1, i.e. it is possible to generate by M_2 every fair sequence of operations that can be generated by M_1. The notion of fair trace-containment is robust with respect to LTL. Therefore, we may conclude that A_4 is satisfied.

Theorem 5.2 *Let $M_c \mathrel{\widehat{=}} (S_c, T_c, I_c, F_c, \Sigma_c, \mathcal{L}_c)$ be the product of the (concrete) transition system determined by the transition system semantics of Object-Z specification and the Büchi automata representing the history invariant. Let $M_a \mathrel{\widehat{=}} (S_a, T_a, I_a, F_a, \Sigma_a, \mathcal{L}_a)$ be a realization constructed using our method. M_c fairly simulates M_a.*

Proof. We claim that the following total relation is a fair simulation relation from M_a to M_c.

$$\mathcal{R} \mathrel{\widehat{=}} \{(a, c) : S_a \times S_c \mid c \text{ is a state where } \mathcal{L}(a) \text{ is true}\}$$

D_1 is an immediate consequence of the fact that the initial condition is included in the predicates for abstraction. In the abstraction process, an abstract state is identified as an initial state if and only if the initial condition is satisfied. Because the weakest condition calculated in the last step is stronger than the precondition, an operation is engaged only when its predication is satisfied. An operation may reach a successor state only if the postcondition is satisfied at the successor state ($e \Rightarrow \mathcal{S}(e)$ for all e : *Predicate*), i.e. there is a corresponding transition in M_c. Thus, D_2 is true. An state in M_a is a final state if it satisfies the fair constraint. All simulating states of the state satisfies the fair constraint (definition of $\mathcal{R}$). Thus, D_3 is true. Therefore, we conclude that M_c fairly simulates M_a.□

6 Automaton

Our method is automated by building an experiment tool in JAVA. The inputs are an Object-Z class specification in its XML representation [31], along with an optional set of predicates for abstraction. By default, the predicates include those in the history invariant and initial schema. The predicate abstraction is automated with the help of Prototype Verification System (PVS [22]). Lemmas are generated automatically from the Object-Z specification for calculating the abstract initial schema, precondition and postcondition of each operation. In general, the number of lemmas are exponential to the number of predicates for abstraction. A number of tricks are used to reduce the number of abstract state, e.g. removing false state by considering co-relation between the predicates. Therefore, the number of lemmas are reduced. PVS is invoked in batch mode to prove the lemmas automatically without user interaction because it is unlikely that a user would like to prove the lemmas interactively for complex systems. To further speed up the abstriction so as to handle complex systems, a more loop-free proving strategy than *grind* (the highest-level command in PVS) is used to prove each lemmas in a limited amount of time.

A raw state machine is constructed from the proving result. It is then composed with the Büchi automata generated from Wring [30]. The product is pruned using our pruning algorithm. If there is at least one initial state and at least one reachable final state left, the pruned state machine is equipped with guard conditions and presented to users as a realization. However, computing the weakest precondition involves eliminating dashed variables. Variable elimination in our context is in general undecidable. Yet an interesting enough subset is decidable where there is no nonlinear integer arithmetic and no shielded variables occurring inside uninterpreted terms. PVS is currently lack of such a procedure. However, we can always use PVS to prove-check a manually constructed stronger guard.

More features on connecting our tool to existing tool supports for FSM-like structures will be offered. For example, we plan to generate XMI [10] representation of our state machines so that they can be exchanged and visualized using tools like Rational Rose [16]. We may also generate codes for Rhapsody [11] so that we may simulate the model and synthesize working code from the Object-Z specification if the implementation of each operation is supplied (and tested by checking the precondition and postcondition) by the user.

7 Discussion

The contribution of our work are twofold. Firstly, we developed a systematic method to abstract an Object-Z specification on a class base. Such a method is useful for verification of Object-Z specification as well. Secondly, we developed an effective way of realizing an Object-Z specification as FSMs. By treating each transition as a function call and implementing each operation in isolation, we may generate executable code from the specification. Moreover, an experimental tool is developed to realize the method.

A less restrictive state machines would allow more possible further refinement of the model. Our method works by pruning those sequences of operations that fails the specification and the additional requirements. Therefore, it is naturally 'minimally' restrictive. However, a minimum restrictive state machine may not exist. An example can be found in [17].

Our work contributes to the transformation from formal specifications to programs [14]. It is related to works on abstraction and controller synthesis. Abstraction techniques are now widely considered useful and even necessary for a successful verification. It has been discussed in various works on model-checking softwares, e.g. Graf's work on property preserving abstractions for transition systems [15] and Ball's work on abstraction of C programs[1]. Though partially inspired by Graf's work, our abstraction schema is highly coupled with Object-Z semantics. The abstraction schema is closely related to the work in [29], where Smith and Winter proposed a similar predicate abstraction for totalized Z specifications. Their aim is to verify safety temporal properties of Z specification. The difference between their abstraction and ours is that our predicate abstraction applies to Object-Z specification (therefore, do not have to assume operations to be totalized) and, more importantly, is automated by PVS. The latter is essential for complex systems.

Our work is also related to works on deriving an automata representation from Z/Object-Z for specification-based testing [5, 19, 13, 20, 21]. Dick and Faivre in [5] derives an automata representation from a Z specification for generating test cases. Murray in [19] formally derives an FSM of an Object-Z specification for the same purpose. However, their works focus on extract a finite set of behaviors for testing (partial coverage) and therefore, are more straightforward. Our work focuses on extracting implementable FSMs from Object-Z specification. On the contrary, we guarantee all behaviors are properly constrained. Also, our abstraction is to facilitate the extraction of the behavior patterns.

Our work is also inspired by works on controller synthesis both from computer science and control-theoretic perspective. The problem of synthesizing controllers is to find a controller that restricts the behavior of a given process in order to satisfy given constraints on sequences of actions executed by the process. The line of work goes back to the realization problem formulated by Church [4] and later solved by Büchi and Landweber [2]. During the past decade, there has been a vigorous revival of this area both from computer science and control-theoretic perspectives. Various problems associated with partial observability, controllability and hierarchical control have been addressed as evidenced in [24, 25, 18]. However, previous works on controller synthesis are all based on automata-like structures with no or trivial data states. Our work applies to applications with complicated data structures and functional requirements.

References

[1] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[2] J. R. Büchi and L. H. Landweber. Solving Sequential Conditions by Finite State Strategies. *Trans. on American Math. Soc.*, 138:295–311, 1969.

[3] A. Cavalcanti and J. Woodcock. A Weakest Precondition Semantics for Z. *The Computer Journal*, 41(1):1–15, 1998.

[4] A. Church. Logic, Arithmetic and Automata. In *Proceeding of International Congr. Math.*, 1960.

[5] J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In *Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pages 268–284. Springer-Verlag, 1993.

[6] E.W. Dijkstra. *A Discipline of Programming*. International Series in Computer Science. Prentice-Hall, 1976.

[7] D. L. Dill, A. J. Hu, and H. Wong-Toi. Checking for Language Inclusion Using Simulation Preorders. In *Computer Aided Verification*, pages 255–265, 1991.

[8] R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z*. Cornerstones of Computing Series. Macmillan, March 2000.

[9] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. Technical Report 94-45, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, Australia, 1994. Also in a special issue of *Computer Standards and Interfaces* on Formal Methods and Standards, September 1995.

[10] OMG Group. Xmi: Xml metadata interchange. http://www-4.ibm.com/software/ad/standards/xmi.html, 2000.

[11] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *Computer*, 30(7):31–42, 1997.

[12] T. A. Henzinger, O. Kupferman, and S. K. Rajamani. Fair Simulation. In *International Conference on Concurrency Theory*, pages 273–287, 1997.

[13] R. M. Hierons. Testing from a Z Specification. *Software Testing, Verification and Reliability*, 7:19–33, 1997.

[14] S. Liu and C. Ho-Stuart. Semi-automatic Transformation from Formal Specifications to Programs. In Alex Stoyenko, editor, *Proceedings Second IEEE International Conference on Engineering of Complex Computer Systems*, pages 506–513. IEEE CS, October 1996.

[15] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design*, 6:11–44, Jan 1995.

[16] Rational Ltd. Rational rose. http://www-306.ibm.com/software/rational/, 2004.

[17] P. Madhusudan and P. S. Thiagarajan. Branching Time Controllers for Discrete Event Systems. *Theoretical Computer Science*, 274:117–149, 2002.

[18] P. Madhusudan and P.S. Thiagarajan. A Decidable Class of Asynchronous Distributed Controllers. In L. Brim, P. Janar, M. Ketinsky, and A. Kuera, editors, *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR 2002)*, Lecture Notes in Computer Science, pages 145–160. Springer-Verlag Heidelberg, 2002.

[19] L. Murray, D. A. Carrington, I. MacColl, J. McDonald, and P. A. Strooper. Formal Derivation of Finite State Machines for Class Testing. In Jonathan P. Bowen, Andreas Fett, and Michael G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 42–59. Springer-Verlag, 1998.

[20] A. J. Offutt and S. Liu. Generating Test Data from SOFL Specifications. Technical Report ISSE-TR-97-02, Department of Information and Software Systems Engineering, George Mason University, 1997.

[21] J. Offutt, S. Y. Liu, A. Abdurazik, and P. Ammann. Generating Test Data From State-based Specifications. *The Journal of Software Testing, Verification and Reliability*, 13(1):25–53, 2003.

[22] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[23] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer science*, pages 46–57, 1977.

[24] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proceedings of the 16th ACM Symposium Principles of Programming Languages (POPL 1989)*, pages 179–190, 1989.

[25] A. Pnueli and R. Rosner. Distributed Reactive Systems are Hard to Synthesis. In *Proceedings 31st IEEE Sypm. on Foudation of Computer Science*, pages 746–757, 1990.

[26] P. J. Ramadge and W. M. Wonham. The Control of Discrete Event Systems. *Special Issue on Discrete Event Dynamic Systems*, 36(1):81–98, Jan 1989.

[27] G. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, University of Queensland, 1992.

[28] G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.

[29] G. Smith and K. Winter. Proving Temporal Properties of Z Specifications Using Abstraction. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina Waldén, editors, *ZB 2003*, volume 2651 of *Lecture Notes in Computer Science*, pages 260–279. Springer, 2003.

[30] F. Somenzi and R. Bloem. Efficient Büchi Automata from LTL Formulae. In *Computer-Aided Verification*, pages 248–263, 2000.

[31] J. Sun, J. S. Dong, J. Liu, and H. Wang. A Formal Object Approach to the Design of ZML. *Annals of Software Engineering*, 13:329–356, 2002.

[32] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall International, 1996.