State Exploration with Multiple State Groupings

Colin Campbell and Margus Veanes

Microsoft Research, Redmond, WA, USA
colin,margus@microsoft.com

Abstract. Exploration algorithms are relevant to the industrial practice of generating test cases from an abstract state machine whose runs define the predicted behavior of the software system under test. In this paper we describe a new exploration algorithm that allows multiple state grouping functions to simultaneously guide the search for states that are interesting or relevant for testing. In some cases, our algorithm allows exploration to be optimized from exponential to linear complexity. The paper includes an extended example that illustrates the use of the algorithm with the Spec Explorer tool developed as Microsoft Research.

1 Introduction

The problem we are addressing here arises in the context of test case generation from model programs. This is a two step process: one first produces a finite transition system that encodes interesting or relevant runs of the model program (a process called ”FSM generation”) and then one traverses the transition system to produce test cases. A model program is a particular form of an ASM: a set of guarded update rules and an implicit input program that chooses any enabled rule in each step.

Although a model program is expressed as a finite text, it typically induces a transition system (an ”unwinding” of the model program) with a very large or even infinite state space. Hence, FSM generation can benefit from various methods that reduce the state space but preserve the ability of the generated test cases to discern behavioral differences.

One such method is to use state-based expressions to group states. The basic approach of using state groupings for FSM generation [7] in the context of test case generation was developed in [7] where groups are called hyperstates. Any two states where the grouping expressions evaluate to the same values are considered indistinguishable with respect to the grouping and are said to belong to the same group.

One limitation of the approach described in [7] is that sometimes a single state grouping may not be adequate. This situation arises if the model program has several parts that correspond to logically independent sub-models each of which induces a large state space on its own. The total state space induced by the whole program is then a Cartesian product of the state spaces induced by the sub-models. If one uses a single grouping that captures relevant properties of all the sub-models, then the generated FSM has a state group for each combination of the values. However, one is typically only interested in states where the different values of grouping expressions for the different sub-models are present, rather than the combination of all the possible values. To overcome this limitation we extend the approach described in [7] to use multiple state groupings.

We first provide some basic definitions in order to clarify the problem. We then describe the use of multiple state groupings in the context of the exploration algorithm and provide an example to illustrate the idea.

2 Preliminaries

A *transition system* M is defined by the components $(S, s^{\text{init}}, A, R)$, where S is a set of *states*, $s^{\text{init}} \in S$ is the *initial state*, A is a set of *actions*, and $R \subseteq S \times A \times S$ is a *transition relation*. A tuple $(s, a, t) \in R$ is a *transition* with *source* s, *label* a, and *target* t. The set of *enabled transtions* from a state s is the set of all transitions in R whose source is s. We say that an action $a \in A$ is *enabled* in a state $s \in S$ if a is the label of some enabled transiton from s. In order to identify a component of a transition system M, we index that component by M, unless M is clear from the context.

A *model program* P is an implicit definition of a transition system by using a finite collection of *guarded update rules* or *ASM rules*. A guarded update rule in P is defined as a parameterized *action method* using AsmL [10] or Spec# [2], similar to the way methods are written in normal programming languages like C#. The program declares a finite vocabulary of model *variables*. A *model state* is a mapping of those variables to concrete values, i.e., a first-order state or an ASM state. The execution of a single step of an action method is an ASM step [9]. The ASM semantics of the core of AsmL is explained in [6].

An action method m in P takes arguments $\boldsymbol{v}_{\text{in}}$ and produces a return value v_{ret}; the types of $\boldsymbol{v}_{\text{in}}$ and v_{ret} are specified by the method signature. Typically the methods of a model program P create objects and use unbounded data structures, like integers, strings, sets, sequences and maps. An action method m is associated with a state-dependent predicate $\textit{require}_m[\boldsymbol{v}_{\text{in}}]$, called the *enabling condition* of m.

The transition system $M_P = (S, s^{\text{init}}, A, R)$ defined by a model program P is a complete unwinding or expansion of P as explained next. The initial state s^{init} is the initial model state given by the initial assignment of variables to values as declared in P. The transition relation R is the smallest relation that is closed under method execution:

- Given a state s, an action method m of P, and input arguments $\boldsymbol{v}_{\text{in}}$ for m, there is a transition $(s, \langle m, \boldsymbol{v}_{\text{in}}, v_{\text{out}} \rangle, t) \in R$ provided that
  - $\textit{require}_m[\boldsymbol{v}_{\text{in}}]$ holds in s, and
  - some execution of $m(\boldsymbol{v}_{\text{in}})$ in s returns the value v_{out} and produces updates on s that yield the sequel state t.

The set S is the smallest set of states closed under R, and A is the set of all labels of transitions in R. If the return value of a method m is (of type) *void*, indicating absence of an explicit return value, the action is denoted by $\langle m, \boldsymbol{v}_{\text{in}} \rangle$. When $\boldsymbol{v}_{\text{in}}$ is the empty sequence we abbreviate $\langle m, \boldsymbol{v}_{\text{in}} \rangle$ by m.

We make the assumption that all methods are void in this paper. This will not affect the algorithm described in the following section where the structure of labels is irrelevant. In AsmL, execution of a method call $m(\boldsymbol{v}_{\text{in}})$ in a state s may be nondeterministic, in which case M_P is *nondeterministic*, i.e. there are several transitions from s with the same label but different targets [6,10].

A *grouping* function G for M_P is a function from states of M_P to concrete values defined by a state-dependent expression.[1] A grouping expression may use state variables and functions defined in the model program but may not have any side effects (i.e. may not change any of the state variables). Given a state s, the value $G(s)$ is called the *G-label* of s. Two states are *G-equivalent* if they have the same G-label. A G-equivalence class of states is called a *G-group*. A grouping G is *finite* if its range is finite, i.e. there are only finitely many distinct G-groups. For example a predicate can be used as a finite grouping, whereas a grouping `x.Length` where `x` is a state variable of type sequence is not finite, unless the length of `x` in S_{M_P} is bounded.

3 FSM generation with multiple state groupings

Groupings provide a way of defining "what is an interesting state partition" from a given testing point of view. The usage of state groupings in FSM generation is that states with the same grouping label are considered to be indistinguishable with respect to the given grouping. This allows the statespace of the model to be collapsed to a finite state machine (FSM) with respect to a given finite grouping. Test cases can then be generated from the FSM using known techniques. The basic algorithm with a single grouping function was introduced in [7], where grouping labels are called hyperstates. The new insight presented here is that a tester may provide several state groupings and associate each one with an integer-valued bound expression. In the following sections we illustrate the use of multiple groupings on a sample model and show why this is a practically useful pruning technique that cannot, in general, be achived with a single grouping.

To define a grouping function formally, consider a model program P and let $M = M_P = (S, s^{\text{init}}, A, R)$. Let G be a given grouping function for M. A G-group of S is denoted by $[s]_{/G}$, where s is a representative of the group. G induces the partitioning $M_{/G} = (S_{/G}, [s^{\text{init}}]_{/G}, A, R_{/G})$ of M, where $S_{/G}$ is the set of all G-groups and

$$R_{/G} = \{([s]_{/G}, a, [t]_{/G}) : (s, a, t) \in R\}.$$

We are now ready to describe the FSM generation algorithm with multiple groupings. Let P be the given model program and let $(G_i)_{1 \leq i \leq k}$ be the given grouping functions for M_P. In addition there is a given *bound* function B_i associated with each grouping G_i that maps a state to a non-negative integer or the value ∞ indicating absence of bound for the given state. The value of $B_i(s)$ is called the *G_i-bound* of state s. It is assumed that for any integer n, $n < \infty$. The use of bounds is explained below.

The algorithm keeps a set of states that have been discovered so far (`States`), a frontier of states that have not been fully explored yet (`Frontier`), a set of transitions included in the generated FSM (`Transitions`), and a set of transitions that have been traversed (`Traversed`). Initially, the state s^{init} has been discovered but not fully

[1] We sometimes say *grouping* to either mean the function or the expression defining the function, depending on the context.

explored, no transitions have been included so far in the generated FSM, and no transitions have been traversed yet.

We describe the exploration algorithm using AsmL as pseudo-code.

```
structure Transition
  Source as State
  Label  as Action
  Target as State

var Transitions as Set of Transition = {}
var Traversed   as Set of Transition = {}
var States      as Set of State = {s^init}
var Frontier    as set of State = {s^init}
```

The following helper functions are used:

- $\texttt{Count}(i, s) = \#(\texttt{States} \cap [s]_{/G_i})$ returns the number of states in $[s]_{/G_i}$ that have been discovered so far.
- $\texttt{Bound}(i, s) = B_i(s)$ evaluates the G_i-bound of state s.

The algorithm repeats the following step until the frontier is empty. Upon completion the outcome of the algortitm is an FSM given by the subset `Transitions` of R and the subset `States` of S. The groupings are enumerated from 1 to k.

```
choose s in Frontier
  choose tr in R - Traversed where tr.Source = s
    let t = tr.Target
    Traversed := Traversed + {tr}
    if t in States then
      Transitions := Transitions + {tr}
    elseif exists i in {1..k} where Count(i,t) < Bound(i,t) then
      Transitions := Transitions + {tr}
      States := States + {t}
      Frontier := Frontier + {t}
    else
      skip
  ifnone
    Frontier := Frontier - {s}
ifnone
  skip
```

The main purpose of a bound function for a grouping G is to specify the desired number of representatives in G-groups. Notice that exclusion of a state s and its incoming transition from the FSM happens only if *all* grouping bounds have been breached for s. Typically the bound function is constant, however, it is sometimes desirable to temporarily raise (or lower) the bar for particular states or groups so that transitions to important (or unimporatant) states are included in (or excluded from) the generated FSM.

The algorithm is described at a very high level to illustrate the main idea, an implementation of it will look different. Most importantly, an implementation would work directly with P, rather than M_P that is typically infinite. P is only partially unwound into M_P and this happens on a need-to-know basis. For example, the choice of transition in the second `choose`-statement would correspond roughly to the following steps.

1. An action method m is chosen from P so that m has not been fully explored from s.
2. Assume m has not been explored at all from s.
   (a) A state-based parameter generator is used to generate a finite collection of possible input argument sequences I for m.
   (b) The method m is fully explored for all input arguments $\boldsymbol{v}_{\text{in}} \in I$ such that $\textit{require}_m[\boldsymbol{v}_{\text{in}}]$ holds in s. The resulting transitions are stored as untraversed m-transitions from s.
   (c) `tr` is selected randomly among the untraversed m-transitions from s.
3. Assume m has been partially explored but not fully explored from s. In this case there is at least one untraversed m-transition from s; `tr` is selected randomly among such transitions.

The use of a multiple state grouping extends the original approach using a single grouping [7] is a way that is analogous to partial order reduction or pairwise parameter combination. For example, a common technique for stateless, combinatorial testing is the use of pairwise selection of parameters for actions. The use of multiple state groupings can be used to apply a similar approach to state-based testing as follows. Assume that the state variables in the model program are $v_1, \ldots, v_n$. For any two distinct variables v_i and v_j define the grouping function G_{ij} by the pair (v_i, v_j) with the associated bound function $B_{ij} = 1$. Thus, two states s an t are G_{ij}-equivalent if $v_i(s) = v_i(t)$ and $v_j(s) = v_j(t)$. Running the exploration algorithm with these groupings yields a state space where all reachable pairwise combinations of variable values are present but not necessarily all states. Notice that a single grouping G defined by the tuple $(v_1, \ldots, v_n)$ would not reduce the state space at all since all states would have distinct G-labels.

4 Example: Counting Problem

In this section we study an example to illustrate the use of multiple state groupings. The example shows the exploration of the abstract state space of a 1-bit counting protocol, expressed in terms of a multi-agent game. The example is also available under the same name in the Spec Explorer distribution [1]. The example was suggested by Yuri Gurevich.

4.1 The problem

There are n prisoners and a prison warden. The warden chooses one prisoner per day to be interviewed privately by drawing from a hat containing the names of all prisoners. During the interview the prisoner is asked "Have all of the prisoners been interviewed yet?" The prisoner may answer or remain silent. If a prisoner does not answer, he is sent back to his cell. If a prisoner answers correctly, then all prisoners go free; however, an incorrect answer results in the execution of all n prisoners. Before the first interview occurs, the prisoners are allowed to get together as a group to devise a strategy. Then they are isolated from each other and the interviews begin. The prisoners know of a light switch by the warden's door. They can observe the state of this switch ("on" or "off") when they are interviewed and can change it on their way back to their cell. The prisoners know that the light switch is initially "off" prior to the first interview, and that the warden never touches the switch. Is there a strategy the prisoners can to use to guarantee their freedom?

4.2 The solution

The prisoners divide themselves into one observer and $n - 1$ signalers. The signalers perform the following action whenever they are interviewed: if the light switch is off, they turn it on one time (in other words, if they have not already done so in a previous interview). Otherwise they do not alter the state of the switch. The observer, whenever he is called in for an interview, checks the state of the switch. If it is on he adds one to a total he keeps in his head and turns the switch off. When the count that the observer keeps track of reaches $n - 1$, he knows that all of his fellow prisoners have been interviewed. At this point, he answers yes to the question, and the prisoners are freed.

4.3 Model state

We can encode the solution as an abstract state machine with the following state:

```
type Prisoner = Integer // prisoners are identifed by number
enum Mode
  Initializing          // prisoners are counting themselves
  Interviewing          // interviews are happening
  Answered              // an answer has been given
  Decided               // the outcome of the game has been decided

var mode as Mode = Initializing;               // phase of the protocol
var prisoners as Set of Prisoner   = {}        // who are the prisoners?
var signal as Boolean              = False     // is the switch on?
var interviewed as Set of Prisoner = {}        // who has been interviewed?
var signalled as Set of Prisoner   = {}        // who has signalled?
var nCounted as Integer            = 0         // how many signals seen?
```

We introduce several derived functions for convenience:

```
NPrisoners() as Integer return prisoners.Size
NInterviewed() as Integer return interviewed.Size
NSignalled() as Integer return signalled.Size
IsObserver(p as Prisoner) as Boolean return p = 1
IsSignaler(p as Prisoner) as Boolean return p <> 1
HasSignaled(p as Prisoner) as Boolean return p in signalled
```

These methods answer the respective questions: How many prisoners are there? How many have been interviewed? How many have sent signals so far? Is prisoner p the chosen observer? Is prisoner p one of the signalers? Has prisoner p previously sent a signal?

4.4 Actions

There are three actions methods that move the game forward: `Start`, `Interview` and `Finish`. Any sequence $(a_1, a_2, \ldots, a_k)$ of actions is a *valid run* if a_1 is enabled in the initial state s_0 and, for $1 \leq i \leq k$, the transition (s_{i-1}, a_i, s_i) is enabled in state s_{i-1}. The game ends when there are no enabled actions.

Start action

```
Start(n as Integer)
  requires n > 1 and mode = Initializing
  prisoners := {1..n}
  mode := Interviewing
```

The action $\langle$`Start`$, n\rangle$ corresponds to the prisoners' strategy meeting, where they choose an observer (by convention, this will be prisoner number 1) and make a count of how many prisoners there are. Note that n is externally given.

Interview action

```
Interview(p as Prisoner)
  requires p in prisoners and mode = Interviewing
  // observer behavior
  if IsObserver(p) and signal then
    let newCount = nCounted + 1       // 1) compute the new count
    nCounted := newCount              // 2) remember the new count
    signal := false                   // 3) turn the switch off
    if newCount = NPrisoners() - 1 then
      mode := Answered                // 4) give answer if possible

  // signaler behavior
  if IsSignaler(p) and not HasSignaled(p) and not signal then
    signal := true                    // 1) give the signal
    signalled += {p}                  // 2) remember that signal was given

  // warden behavior
  interviewed += {p}                  // remember the interviewed prisoner
```

The $\langle$`Interview`$, p\rangle$ action represents each occurrence of an interview. If the prisoner p being interviewed is the observer and the switch has been set, then the observer increments his count and resets the switch. If the observer determines that all of the signalers have given a signal (there will $n - 1$ of these), then he gives the answer to the warden (`mode` becomes `Answered`). If p is a signaler who has not previously given his signal *and* there is no previous signal pending the observers' attention, then the prisoner sets the signal. Finally, the warden observes that p has been interviewed. This knowledge will be used later to decide the outcome of the game.

Finish action

```
Finish()
  requires mode = Answered
  if NInterviewed() == NPrisoners() then
    WriteLine("All prisoners are freed.")
  else
    WriteLine("All prisoners are executed.")
  mode := Decided
```

The `Finish` action evaluates the answer given by the observer and determines the outcome.

4.5 State groupings

The example shows how using the technique of multiple state groupings can reduce the number of test cases needed to cover the desired configurations of state variables. In this case the state groupings help to reduce the size of the explored state space to be linear in the size of prisoners. Full exploration would yield a state space that grows exponentially with the number of prisoners. Exploration with a single combined grouping (G_1, G_2, G_3) yields a number of groups that is quadratic in the number of prisoners.

The appropriate state groupings for exploring this model are

- Warden state: $G_1 = \texttt{NInterviewed()}$
- Prisoner state: $G_2 = (\texttt{signal}, \texttt{nCounted})$
- Control state: $G_3 = \texttt{mode}$

Suppose that the three groupings reflect the desired testing coverage. We also assume that the bound for each grouping is the constant function 1. In other words, we wish to see at least one state for each count of prisoners that have been interviewed, but we do not care about which prisoners have been interviewed. We also care about the state that the observer keeps about prisoners. Finally, we want to see all the modes. It is immaterial for the testing purpose if all the combinations of the groupings have been reached.

4.6 Analysis

We consider runs of the system where the number of prisoners is fixed at 3. We see in Figure 1 that full exploration of the model program produces an FSM with 47 transitions among 17 distinct states. The FSM encodes all possible traces of the system as paths from the initial state S0 to the end state S10. The number of possible traces is infinite and also the maximum number of steps in a given trace has no limit (due to looping `Interview`-actions).

We can use the FSM in Figure 1 to generate test sequences that cover all transitions. For example, it takes 8 test sequences that begin in S0 and end in S10, with a total of 79 steps, to cover every transition given in the state machine.

We can project the FSM in Figure 1 on a chosen state grouping; see Figure 2. For example, if we consider control state, we project the 17 distinct states of the FSM into four groups (and the 47 transitions into 6), as shown in Figure 2(a). Clearly, by generat-

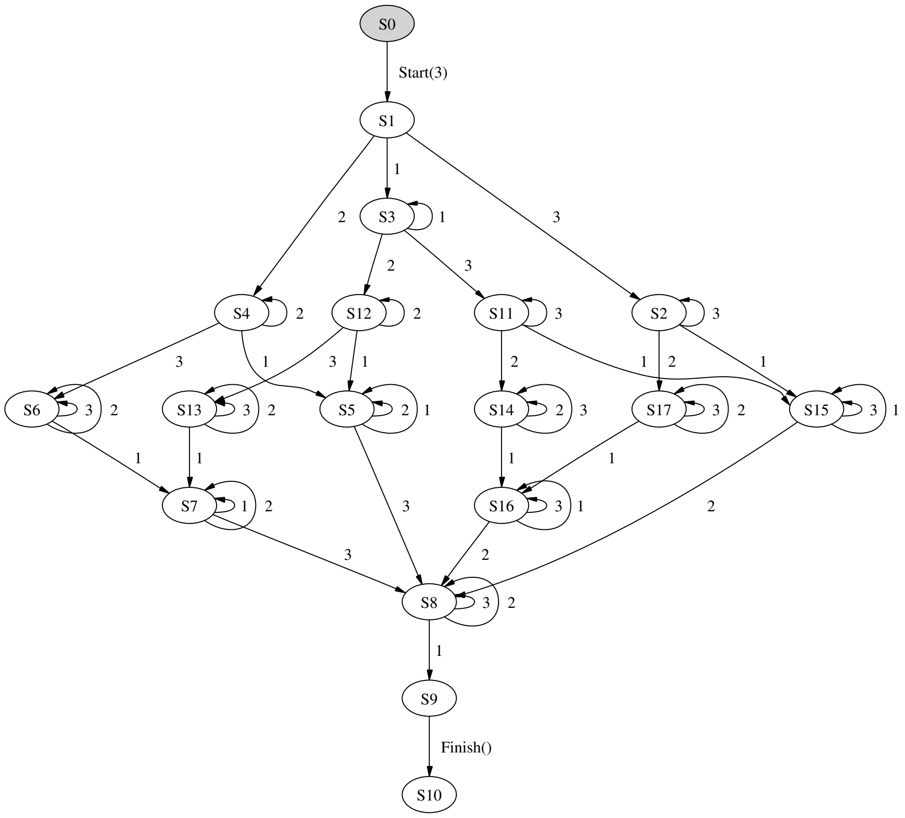

Fig. 1. Full FSM generated from the model with 3 prisoners. Transition labels corresponding to `Interview`-actions are abbreviated by the prisoner being interviewed.

ing tests for the full FSM in Figure 1 we would achieve this goal. However, if we would start for example with ten prisoners, this solution would already be infeasible. Let us apply the state exploration algorithm using multiple state groupings with the groupings G_1, G_2 and G_3 and the constant bound function 1 for all groupings. Figure 3 shows an example result: an FSM with seven states and eleven transitions.

The FSM shown in Figure 3 is advantageous because it contains many fewer states and transitions than the FSM of Figure 1, but it still produces projections for G_1, G_2 and G_3 with the same group labels as those generated from the FSM of Figure 1. As mentioned above, if we increase the number of players ("prisoners"), the full FSM grows exponentially while the FSM given by the method described here grows linearly.

The benefit of this can be seen in the test cases: the FSM of Figure 3 requires only a single test sequence from S0 to S6 (with 11 total steps) to achieve the same coverage of groups under projections G_1, G_2 and G_3 as the FSM of Figure 1 which required 8 sequences with a total length of 79 steps. If we increase the number of prisoners to 10 say, the difference becomes more visible. The full FSM has then 40650 states and 406472

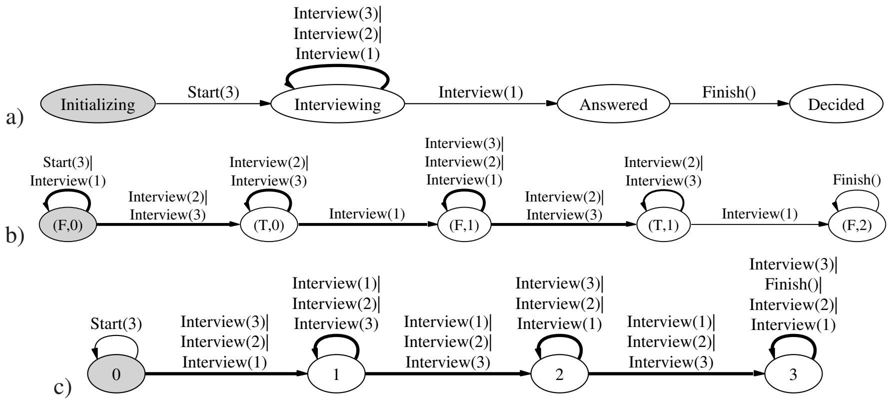

Fig. 2. Projections of FSM in Figure 1 with respect to: a) *control state* grouping G_3; b) *prisoners state* grouping G_2; c) *warden state* grouping G_1. All groups are displayed with their respective labels. Bold arrows indicate that there are more than one underlying state transition.

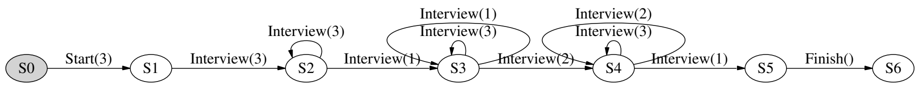

Fig. 3. FSM generated from the model with three prisoners, using the exploration algorithm with multiple state groupings G_1, G_2 and G_3.

transitions, whereas the algorithm using the grouping functions generates an FSM like the one in Figure 4, and test generation from it produces 4 test sequences with a total length of 173 steps, that provide the desired covergage under all of the three groupings. If we combined all the three groupings into a single grouping (`NInterviewed()`, `signal`, `nCounted`, `mode`) we would obtain in this case 32 test cases with the total length of 882 steps.

From our experience, this technique is mostly useful for model-based testing of loosely coupled componentized systems, where the aim is to reach all the interesting states of all the components, but not necessarily all the possible combinations of those. When the components are essentially independent, combining the groupings may lead to state space explosion. Such patterns occur frequently for example in system-level testing of platforms for distributed applications or testing services of the operating system that interact with the environment.

5 Related work

The multiple state grouping algorithm described in this paper is one aspect of the FSM generation algorithm implemented in the Spec Explorer tool, the overall goal of which is to get a finite state space of the size that allows one to explore the state space for testing.

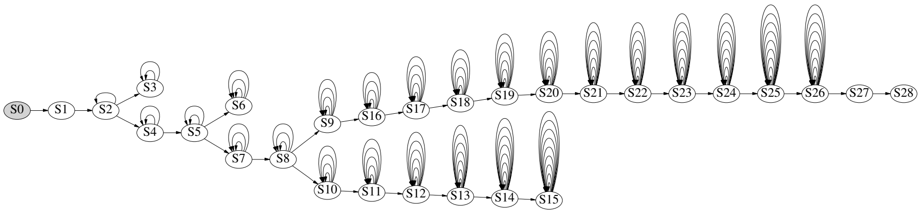

Fig. 4. FSM generated from the model with 10 prisoners, using the exploration algorithm with the state groupings G_1, G_2 and G_3.

To this end, Spec Explorer enables the tester also to generate a finite but representative set of parameters for the action methods. FSM generation from ASMs, based on a single state grouping, was introduced in [7]. This algorithm was initially implemented in the AsmLT tool that is the predecessor of Spec Explorer; the algorithm is also described in [4]. Similar techniques involving the use of state groupings for state space reduction have been used in other model-based testing tools [11,13], although, as far as we know, not with multiple simultaneous groupings as described in this paper. The notion of state grouping is analogous to predicate abstraction in the context of model-checking when the grouping expressions are tuples of Boolean expressions, in which case a grouping label of a state is a tuple of Boolean values. The use of multiple state groupings would correspond to the use of multiple simultaneous predicate abstractions, which to the best of our knowledge has not been considered in this context. The effect of using multiple state groupings on a large state space is sometimes reminiscent to partial order reduction, although we believe there is no direct connection between the two.

For testing large reactive multithreaded or distributed systems, it is sometimes not feasible to first generate an FSM and then generate tests from it. On-the-fly testing is a technique in which test derivation from a model program and test execution are combined into a single algorithm. It can also be called behavioral stress testing or model-based *online* testing, to distinguish it from *offline* test generation as a separate process. On-the-fly testing is supported by Spec Explorer and other model-based testing tools like TorX [15] and TGV [12]. The on-the-fly testing algorithm that is currently implemented in Spec Explorer uses state-dependent action weights to select controllable actions during testing [16]. One of our current plans is to use the multiple state groupings algorithm to dynamically record the coverage of the state space and to use this information in the on-the-fly action selection strategies, e.g. to avoid reentering groups that have already been visited. This idea has not been implemented yet.

For modeling *reactive* systems, Spec Explorer allows the action methods to be split into *controllable* and *observable* ones. Testing of a reactive system is viewed as a game, where the two players are the test tool and the implementation under test [3,14]. Only one of the two players, namely the tester, has a goal. The other player is disinterested and makes random choices. A variation of the multiple state grouping algorithm is used in this context as a pruning technique to generate what we call a finite *test graph* from the model program. Test case generation algorithms from test graphs use techniques from Markov decision process theory for expected cost optimization [3], and specialized algorithms for other optimization criteria [14]. The definition of conformance of the implementation under test to the model is in this case alternating simulation of interface automata [5], rather than trace inclusion or simulation of transition systems. Games for testing are also discussed in [17].

Spec Explorer is briefly described in [8]; a more comprehensive description of it is in preparation. The tool can be downloaded from [1].

References

1. Spec Explorer. URL: http://research.microsoft.com/specexplorer.
2. M. Barnett, R. Leino, and W. Schulte. The Spec# programming system. In M. Huisman, editor, *Cassis International Workshop, Marseille*, LNCS. Springer, 2004.
3. A. Blass, Y. Gurevich, L. Nachmanson, and M. Veanes. Play to test. Technical report MSR-TR-2005-04, Microsoft Research, January 2005.
4. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.
5. L. de Alfaro. Game models for open systems. In N. Dershowitz, editor, *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *LNCS*, pages 269 – 289. Springer, 2004.
6. U. Glässer, Y. Gurevich, and M. Veanes. Abstract communication model for distributed systems. *IEEE Transactions on Software Engineering*, 30(7):458–472, July 2004.
7. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA'02*, volume 27 of *Software Engineering Notes*, pages 112–122. ACM, 2002.
8. W. Grieskamp, N. Tillmann, and M. Veanes. Instrumenting scenarios in a model-driven development environment. *Information and Software Technology*, 46(15):1027–1036, December 2004.
9. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
10. Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of AsmL. *Theoretical Computer Science*, 2005. To appear in special issue dedicated to FMCO 2003, preliminary version available as Microsoft Research Technical Report MSR-TR-2004-27.
11. A. Hartman and K. Nagin. Model driven testing - AGEDIS architecture interfaces and tools. In *1st European Conference on Model Driven Software Engineering*, pages 1–11, Nuremberg, Germany, December 2003.
12. C. Jard and T. Jéron. TGV: theory, principles and algorithms. In *The Sixth World Conference on Integrated Design and Process Technology, IDPT'02*, Pasadena, California, June 2002.
13. V. V. Kuliamin, A. K. Petrenko, A. S. Kossatchev, and I. B. Bourdonov. UniTesK: Model based testing in industrial practice. In *1st European Conference on Model Driven Software Engineering*, pages 55–63, Nuremberg, Germany, December 2003.
14. L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. Optimal strategies for testing nondeterministic systems. In *ISSTA'04*, volume 29 of *Software Engineering Notes*, pages 55–64. ACM, July 2004.
15. J. Tretmans and E. Brinksma. TorX: Automated model based testing. In *1st European Conference on Model Driven Software Engineering*, pages 31–43, Nuremberg, Germany, December 2003.
16. M. Veanes, C. Campbell, W. Schulte, and P. Kohli. On-the-fly testing of reactive systems. Technical report MSR-TR-2005-05, Microsoft Research, January 2005.
17. M. Yannakakis. Testing, optimization, and games. In *Proceedings of the Nineteenth Annual IEEE Symposium on Logic In Computer Science, LICS 2004*, pages 78–88. IEEE, 2004.