Chapter 7

Experimental Results

In this chapter we present some empirical results in applying the UV system to some practical problems. Each example has been chosen with the intention to emphasize a particular aspect of the UV system and the model checking procedures it is based on.

In section 7.1 we present a two-process mutual exclusion protocol that has been one of the first applications of the UV system, which detected errors in a manual correctness proof. A resource allocation protocol based on a dining philosopher algorithm demonstrates the increased efficiency of local safety checking in section 7.2. The counter example of chapter 4 has been included in section 7.3 to document the savings obtained by generalized progress checking. The remaining two examples, Milner's cycler in section 7.4 and the elevator control program of chapter 5 in section 7.5, are more elaborate examples that illustrate the advantages of using generalized progress properties.

Each example is presented with the program and relevant properties written using the UV input language. However, for the sake of conciseness of the listings some language features not implemented in the current revision 2.3.3 of the UV system are used. In particular, quantified assignments and formulae are used extensively. We refer to the language including these additional syntactic features as the

extended UV input language.

All examples were run on a SPARC-20 workstation with about 20 MB of main memory allocated to the model checker.

7.1 Two-Process Mutual Exclusion

The following two-process mutual exclusion algorithm is taken from [Mis90a], where it is derived by refinement from a set of properties. Important properties of the algorithm, like mutual exclusion and absence of starvation, are established along with the manual. Although these properties can be established, some of the intermediate invariant proofs in the paper were indeed not correct. Some of these errors had been discovered independently [DF90], but they were all discovered automatically by the UV system.

The mutual exclusion algorithm for two processes U and V is based on an encoding of a *shared queue* by three boolean variables `u`, `v`, and `p`. Processes enter their IDs at the end of the queue if they are requesting to enter their critical sections; the ID at the head of the queue belongs to the process permitted to enter its critical section; upon leaving its critical section a process removes its ID from the head of the queue. The variables `u` and `v` indicate whether the IDs of U or V are in the queue, respectively; variable `p` indicates which process ID is at the head of the queue: if it is the ID of V then `p` is true, if it is the ID of U then it is false (its value is immaterial if the queue is empty).

The encoding of the mutual exclusion algorithm as a UNITY program is given in the following listing. We have added two boolean variables `hu` and `hv` to model the possibility that either process can remain in its non-critical section forever (without these variables each process would request to enter its critical section eventually due to the unconditional fairness constraint).

```
program Mutex
  declare
    type PC = enum(noncritical, requesting,
                   trying, critical, exiting);
    var m, n: PC;
    var u, v, p: boolean;
    var hu, hv: boolean;

  always
  initially
    !u;
    !v;
    m = noncritical;
    n = noncritical;
  assign

    // first process (u)
    [u0]    hu   := !hu
    [u1]    u, m := true, requesting   if hu /\ m = noncritical
    [u2]    p, m := v, trying          if m = requesting
    [u3]    m    := critical           if !p /\ m = trying
    [u4]    u, m := false, exiting     if m = critical
    [u5]    p, m := true, noncritical  if m = exiting

    // second process (v)
    [v0]    hv   := !hv
    [v1]    v, n := true, requesting   if hv /\ n = noncritical
    [v2]    p, n := !u, trying         if n = requesting
```

```
    [v3]    n    := critical           if p /\ n = trying
    [v4]    v, n := false, exiting     if n = critical
    [v5]    p, n := false, noncritical if n = exiting

end;

// invariants
in Mutex: invariant u == (m >= requesting /\ m <= critical);
in Mutex: invariant v == (n >= requesting /\ n <= critical);

// invalid invariants
in Mutex: invariant m = critical \/ m = exiting ==> !p;
in Mutex: invariant n = critical \/ n = exiting ==> p;
in Mutex: invariant (u == (m >= requesting /\ m <= critical))
                    /\ (m = critical \/ m = exiting ==> !p);
in Mutex: invariant (v == (n >= requesting /\ n <= critical))
                    /\ (n = critical \/ n = exiting ==> p);

// corrected invariants
in Mutex: invariant (u == (m >= requesting /\ m <= critical))
                    /\ (m = critical ==> !p);
in Mutex: invariant (v == (n >= requesting /\ n <= critical))
                    /\ (n = critical ==> p);

// misc. properties
in Mutex: m = trying unless m = critical;
in Mutex: m = requesting --> (p == v) /\ m = trying;
in Mutex: m = critical --> p;
```

```
// mutual exclusion
in Mutex: invariant !(m = critical /\ n = critical);

// absence of starvation
in Mutex: m = requesting --> m = critical;
```

Due to the small size of program *Mutex* the strongest invariant and all valid properties can be checked almost instantaneously. Two facts, however, are worth noting: first, the four invariants in the sections labeled *invariants* and *corrected invariants* are sufficient to establish all the other valid properties. Second, the number of inner iterations necessary for checking the absence of starvation property can be reduced from 116 to 22 by using the regular expression hint $[u2]([v2][v3][v4][v5])[u3]$, which captures the transition of process u from `requesting` to `critical` with the possibly necessary pass of process v through its critical section.

7.2 Resource Allocation: Dining Philosophers

The dining philosophers program presented here is based on the distributed dining philosophers algorithm found in [CM88]. It implements a ring topology in which two neighboring processes share a fork. Each process is in one of three states, *thinking*, *hungry*, or *eating*. A process can transit from *thinking* to *hungry* at any time, can move from *hungry* to *eating* only if neither of its neighboring processes is in its *eating* state, and moves from *eating* to *thinking* after finite time. While the transitions from *thinking* to *hungry* and from *eating* to *thinking* are under control of the processes, a scheduler has to determine when processes can transit from *hungry* to *eating*. An important property that need to be maintained by the scheduler is the *mutual exclusion* property, stating that two neighboring processes are not eating at

the same time.

In order to resolve conflicts arising when two neighboring processes are ready to transit from *hungry* to *eating*, the algorithm for the scheduler maintains a partial order among the processes in which processes with higher priority are chosen over processes with lower one. To guarantee that no process is permanently discriminated against, the partial order needs to be dynamic and fair over time. This is accomplished my maintaining a directed acyclic graph over the topology of the processes, in which an edge from process u to process v indicates that u has higher priority than v. A process chosen to enter its *eating* state decreases its priority by changing all incident edges to point away from it.

The algorithm implements the directed acyclic graph in a distributed fashion in which every pair of neighboring processes communicates via three variables, `fork`, `clean`, and `rf`. These variables encode both the ordering between the processes and their respective states. For a detailed description of this encoding and the derivation of the algorithm, the reader is referred to [CM88].

In the following we show the program and several relevant properties written in the extended UV input language. The syntactic parameter `N` is instantiated to yield ring topologies of varying sizes. Syntactic features used in the presentation of this program that are not present in the current implementation of the UV input language include the quantification of expressions and assignments, the use of array literals (such as the one in the definition of `other`), and the if-then-else operator `<| |>`.

```
program dining

  declare
    type State = enum(thinking, hungry, eating);
    type Neighbors = enum(left, right);
    type Index = cyclic(N);
```

```
  var ready : boolean;    // used for modeling the nondeterministic
                          // behavior of a thinking process

  var dine : Index -> State;
  var clean : Index -> boolean;
  var fork : Index -> Neighbors;
  var rf : Index -> Neighbors;

always
  other : Neighbor -> Neighbor =
      (-> n: Neighbor |: left <| n = right |> right);
  mayEat : Index -> boolean =
  (-> i: Index |: (fork.i = left /\ (clean.i \/ rf.i = right)) /\
                      (fork.(i-1) = right /\ (clean.(i-1) \/
                                              rf.(i-1) = left)));
  sendReq : Index -> Neighbors -> boolean =
      (-> i: Index |:
      (-> n: Neighbors |: fork.i = n /\ rf.i = other.n /\
                              dine.i = hungry));
  sendFork : Index -> Neighbors -> boolean =
  (-> i: Index |:
      (-> n |: Neighbors: fork.i = n /\ !clean.i /\
                              rf.0 = n /\ !(dine.i = eating)));

initially
  (/\ i: Index |: dine.i = thinking);
  (/\ i: Index |: !clean.i);
```

```
    (/\ i: Index |: fork.i != rf.i);
    (/\ i: Index |: fork.i = left <| i != N-1 |> right);

  assign
    [toggle]    ready := !ready

    ([] i: Index |:
    [th]    dine.i := hungry if dine.i = thinking /\ ready)

    ([] i: Index |:
    [et]    dine.i := thinking if dine.i = eating)

    ([] i: Index |:
        [he]    dine.i, clean.i, clean.(i-1) := eating, false, false
            if (dine.i = hungry) /\ mayEat.i)
    ([] i: Index |:
    ([] n: Neighbor |:
        [r] rf.i := n if sendReq.i.(other.n)))

    ([] i: Index |:
    ([] n: Neighbor |:
        [f] fork.i, clean.i := n, true if sendFork.i.(other.n)))

end;

// auxiliary invariants
in dining: (/\ i: Index |:
                  invariant (dine.i = eating) ==>
```

```
                          (fork.i = left) /\ !clean.i);
in dining: (/\ i: Index |:
                  invariant (dine.i = eating) ==>
                          (fork.i-1 = right) /\ !clean.i-1);

// mutual exclusion property
in dining: (/\ i, j: Index | i != j:
                invariant !(dine.i = eating /\ dine.j = eating));
```

In order to establish the mutual exclusion of the two neighboring processes 0 and 1, we have to check the following property:

```
in dining: invariant !(dine.0 = eating /\ dine.1 = eating);
```

The attempt to compute the strongest invariant is not successful for all but small values of `N`. For instance, the program for a ring with 10 processes has 71 assignment statements, requires 51 state bits for a syntactic state space of $1.27 \cdot 10^{14}$ states which has a diameter of 104 (i.e., the number of iterations needed in the computation of the strongest invariant). Computing the strongest invariant takes about 45 minutes and produces a OBDD representation consisting of 844 nodes. The subsequent check of the invariant property takes only a few milliseconds.

We contrast this with the interactive approach in which the user supplies design knowledge in the form of invariants. From the design of the algorithm, it is immediately clear that a process can only be in its eating state if it holds both forks and both the forks are dirty. In particular, the following weaker invariants can be asserted for processes 0 and 1:

```
in dining: invariant (dine.0 = eating) ==>
                     (fork.0 = left) /\ !clean.0;
in dining: invariant (dine.1 = eating) ==>
                     (fork.0 = right) /\ !clean.0;
```

The above invariants can be established directly with the type invariant, the mutual exclusion property can then be checked successfully with respect to the current invariant. For 10 processes all three checks together take only a few milliseconds. Checking mutual exclusion for all 10 processes requires 20 auxiliary invariants, all of which can be checked in about 0.5 seconds. For 20 processes with 101 state bits and $8 \cdot 10^{24}$ states, checking of the 20 mutual exclusion properties and the 40 auxiliary invariants establishes mutual exclusion in about 2 seconds. An important point is that the auxiliary design invariants express a straightforward fact about the design of the algorithm and are therefore readily available in a situation in which the verification is not entirely separated from the program design.

7.3 Progress by Regular Expressions: A Counter

In chapter 4 the counter program *UpDown* was used as an illustrative example to introduce the concept of generalized progress properties and was discussed there in detail. Here, we present some performance measurements that demonstrate the improved efficiency of using generalized progress properties as compared to ordinary leads-to properties.

In the following listing of the program and the two progress properties the syntactic parameter `N` needs to be instantiated for different counter sizes:

```
program UpDown
  declare
    var b: boolean;
    var x: int(0..N-1);
  always
  initially
  assign
    [set]      b := true
```

```
    [up]        x := x+1 if !b /\ x < N-1
    [down]      x := x-1 if x > 0
end;

in UpDown: true --> x=0;
in UpDown: true --> x=0 by [set][down]*;
```

In the following table we summarize performance measurements for different counter sizes. The two properties, for which model checking is compared, are the ordinary leads-to property $\text{true} \mapsto x = 0$ (indicated by $\mapsto$ in the table), and the generalized leads-to property $\text{true} \overset{[set][down]*}{\longmapsto} x = 0$ (indicated by r- $\mapsto$). Three measurements are listed: *iterations* states the number of inner fixpoint iterations needed to complete the check, *ops* states the number of OBDD node lookup requests in thousands, and *time* shows the execution time in seconds. All model checker invocation establish the respective property as correct and use only the automatically generated type invariant.

N		10	20	50	100	200	500	1000	10000
itera-	$\mapsto$	107	320	1551	5608	21222	128000	506006	n/a
tions	r- $\mapsto$	41	81	201	401	801	2001	4001	40001
ops	$\mapsto$	2.5	11	100	548	2810	22283	88933	n/a
in 10^3	r- $\mapsto$	0.8	2.1	8.7	18	39	119	243	3844
time	$\mapsto$	0.3	0.3	1.1	4.8	27.3	227.9	1028.4	n/a
in s	r- $\mapsto$	0.2	0.3	0.3	0.4	0.5	1.1	2.2	38.0

As can be seen from this table, the number of iterations for the ordinary leads-to check is quadratic in `N`, whereas it is only linear for the generalized leads-to check.

7.4 Scheduling: Milner's Cycler

The scheduling problem known as *Milner's Cycler* ([Mil89]) has been described in section 2.2.1; it also has been used to illustrate the UNITY model checking procedure in section 3.4. In the following, we present some performance measurements for checking different properties.

Milner's Cycler has been used as a benchmark in the literature in order to compare different verification methods and systems. Preliminary results in [Kal95a] demonstrated the advantages of using design invariants for checking safety properties. The following listing shows the program *Cycler* together with several relevant properties written in the extended UV input language, where the syntactic parameter `N` needs to be instantiated to the actual ring size.

```
program Cycler

declare
  type Index = cyclic(N);
  type PC = enum(start, sync, choose, bc, cb);

  var a: Index;          // a holds the index of the last process
                         // that performed its a-action
  var i: Index;          // specification variable
  var cyc: Index -> PC;  // process states

initially
  a = N - 1;
  cyc.0 = start;
  (/\ i: I | i != 0: cyc.i = bc);
```

```
assign

  ([] i: Index |:
    [st] a, cyc.i := i, sync    if cyc.i = start
  )

  ([] i: Index |:
    [ch] cyc.i := bc            if cyc.i = choose
  )

  ([] i: Index |:
    [cb] cyc.i := start         if cyc.i = cb
  )

  ([] i: Index |:
    [sb] cyc.i, cyc.(i+1) := choose, start
                    if cyc.i = sync /\ cyc.(i+1) = bc
  )

  ([] i: Index |:
    [sc] cyc.i, cyc.(i+1) := choose, cb
                    if cyc.i = sync /\ cyc.(i+1) = choose
  )

end;

// design invariants
in Cycler: invariant a = i ==>
```

```
        cyc.i = sync \/ cyc.(i+1) = start \/ cyc.(i+1) = cb;
in Cycler:
invariant (/\ i: Index |: ((cyc.i = cb \/ cyc.i = start \/
                            cyc.i = sync) ==>
            (/\ j: Index | i != j: (cyc.j = choose \/ cyc.j = bc))));

// safety property characterizing cyclic behavior
in Cycler: a = i co a = i \/ a = i+1;

// progress properties asserting absence of deadlock
in Cycler: a = i --> a = i+1;
in Cycler: a = i --> a = i+1 by
            ([sc.i][cb.(i+1)][sb.i])[st.(i+1)];

// specific versions of the above progress properties for i = 1
in Cycler: a = 1 --> a = 2;
in Cycler: a = 1 --> a = 2 by ([sc.1][cb.2][sb.1])[st.2];
```

The design invariants are obtained from the design of the processes, or, alternatively, from inspection of the state transition diagram as follows. If the a-action of process i has been taken most recently (modeled by the predicate $a = i$), then either process i is in state `sync`, or it has synchronized with process $i+1$ which has not yet performed its a-action, i.e., process $i+1$ is in state `cb` or in state `start`. This is formulated as the first design invariant above.

The second design invariant is derived from the following observation: process states can be divided into two classes, one containing `cb`, `start`, and `sync`, the other containing `bc` and `choose`. The two program statements effecting a process synchronization, namely `[sb]` and `[sc]`, have the property that the two participating processes have states in different classes, and each participating process changes

its class as a result of the synchronization. Together with the fact, that initially exactly one process, namely process 0, has a state in the first class, it follows, that there is always at most one process with in a state from that first class. This is formulated in the second design invariant.

In the following table we compare the checking of the safety property $a = i$ **co** $a = i \vee a = i + 1$ by computing the strongest invariant (indicated by *si*) and by using the design invariants (indicated by *inv*). The following table lists the number of OBDD node lookup requests in thousands under *ops*, the checking time in seconds under *time*, as well as the size of the syntactic state space under *states* and the maximum distance of any state from the start state under *diameter* for various sizes of `N`.

N		4	8	12	16	20
states		$1.00 \cdot 10^4$	$2.50 \cdot 10^7$	$3.52 \cdot 10^{10}$	$3.91 \cdot 10^{13}$	$3.81 \cdot 10^{16}$
diameter		20	44	68	92	116
ops	si	24	313	2479	9980	32990
in 10^3	inv	5	21	54	86	170
time	si	0.4	2.2	16.6	74.0	296.5
in s	inv	0.3	0.5	0.7	1.1	1.9

Next, we check the progress property, $a = i \mapsto a = i + 1$. Clearly, progress from $a = i$ to $a = i+1$ is achieved in two phases: first, process $i+1$ has to reach its `start` state, then $a = i + 1$ is established by virtue of the `[st.`$(i+1)$`]` action. In the first phase synchronization between processes i and $i+1$ needs to take place, achieved by either action `[sb.`i`]` or action `[sc.`i`]`. Action `[sb.`i`]` moves process $i+1$ to its `start` state, whereas after action `[sc.`i`]` an execution of action `[cb.`$(i+1)$`]` is necessary in order to complete the first phase. Since the choice between synchronization via `[sb.`i`]` or `[sc.`i`]` is not determined by the current state (for instance, if process i is in its `sync` state and process $i + 1$ is in its `choose` state, then executions with

either synchronization action are possible), we choose as regular expression for the first phase the sequential composition $\texttt{[sc.}i\texttt{][cb.}(i+1)\texttt{][sb.}i\texttt{]}$, which results in the regular expression for the progress property as shown above.

The following table compares the ordinary progress check (indicated by $\mapsto$) for the property $a = 1 \mapsto a = 2$ with the generalized one (indicated by r-$\mapsto$) using the regular expression hint derived above. As in the previous examples, the lines labeled with *iterations* show the total number of inner fixpoint iterations, the ones labeled with *ops* show the number of OBDD node lookup requests in thousands, and the ones labeled with *time* show the checking time in seconds:

N		4	8	12	16	20
itera-	$\mapsto$	86	174	268	370	480
tions	r-$\mapsto$	12	12	12	12	12
ops	$\mapsto$	22	287	2030	8917	29334
in 10^3	r-$\mapsto$	7	24	52	87	145
time	$\mapsto$	0.4	2.3	16.5	87.7	369.8
in s	r-$\mapsto$	0.3	0.5	0.8	1.3	1.8

In this example the effect of using a generalized progress property is particularly impressive. Since the regular expression hint does not contain a *-operator, the verification condition evaluation is reduced to a few simple fixpoint computations of depth 1. We are thereby able to take advantage of the local nature of the achievement of progress, something we could not have done with ordinary leads-to properties. We could have attempted to establish the progress property by a series of **ensures** properties; in that case, however, we would have been required to come up with suitable intermediate predicates, a task which the verification check using generalized progress properties performs implicitly.

7.5 An Elevator Control Program

The elevator control program has been discussed in detail in section 5.3. Here we present the program source in the extended UV language and report some of the performance results for checking progress properties.

The following listing shows the program *Elevator*, four design invariants, and the leads-to properties we are interested in. The syntactic parameter `N` needs to be instantiated to obtain programs for various number of floors:

```
program Elevator

declare
  type Range = int(1..N);
  var state: enum(STOP, UP, DOWN); // state of elevator
  var dir  : int(-1..1);           // direction of movement
  var pos  : Range;                // current floor
  var req  : Range -> boolean;     // array of requested floors
  var user : boolean;              // user enable flag

always
  upReq    : boolean = (\/ i: Range | pos < i: req.i);
                                   // indicates request above
  downReq  : boolean = (\/ i: Range | pos > i: req.i);
                                   // indicates request below

initially
  pos = 1 /\ dir = 0 /\ state = STOP;
  (/\ i: Range |: !req.i);
```

```
assign
  [service] req.pos, state := false, STOP
            if req.pos

  [move]    pos := pos + dir
            if state != STOP /\ !req.pos

  [goOn]    state := UP
            if upReq /\ dir = 1 /\ state = STOP /\ !req.pos
         ~        DOWN
            if downReq /\ dir = -1 /\ state = STOP /\ !req.pos

  [turnUp]  state, dir := UP, 1
            if upReq /\ (dir = 0 \/ !downReq)
                 /\ state = STOP /\ !req.pos

  [turnDown]    state, dir := DOWN, -1
            if downReq /\ (dir = 0 \/ !upReq)
                   /\ state = STOP /\ !req.pos

  [halt]    dir := 0
            if !upReq /\ !downReq /\ state = STOP /\ !req.pos

  ([] i: Range |:
    [request]   req.i := true
                if pos != i /\ user)

  [toggle]  user := !user
```

```
end;

// auxiliary invariants

in Elevator: invariant state = UP ==> dir = 1;
in Elevator: state = DOWN ==> dir = -1;
in Elevator: invariant state = UP ==> upReq \/ req.pos;
in Elevator: invariant state = DOWN ==> downReq \/ req.pos;

// quantified leads-to properties

in Elevator: (/\ i: Range |:
                    req.i --> pos = i /\ state = STOP);
in Elevator: (/\ i: Range |:
                    req.i --> pos = i /\ state = STOP by
                     ([service][turnUp][turnDown][goOn][move])*);

// specific leads-to properties for i = 3

in Elevator: req.3 --> pos = 3 /\ state = STOP;
in Elevator: req.3 --> pos = 3 /\ state = STOP by
                ([service][turnUp][turnDown][goOn][move])*;
```

The strategy for the generalized leads-to properties has been derived and motivated in section 5.3. We compare the verification of the specific leads-to properties, first using the ordinary leads-to, then using the regular expression strategy. The results are summarized in the following table, where *iterations* states the number of inner fixpoint iterations needed to complete the check, *ops* states the number of OBDD

node lookup requests in millions, and *time* shows the execution time in seconds. The properties have been checked with respect to the current invariant, being the conjunction of the four auxiliary design invariants. All checks establish the respective properties to hold for program *Elevator*. The ordinary leads-to property is indicated by $\mapsto$, the generalized by r- $\mapsto$:

N		20	50	100
states		$3.77 \cdot 10^8$	$1.01 \cdot 10^{18}$	$2.28 \cdot 10^{33}$
itera-	$\mapsto$	2264	13248	51576
tions	r- $\mapsto$	658	1798	3698
ops	$\mapsto$	2.25	18.3	659
in 10^6	r- $\mapsto$	1.17	7.6	95
time	$\mapsto$	10.8	280	>10000
in s	r- $\mapsto$	5.3	83	1150

Using the regular expression hint reduces the number of required fixpoint iterations significantly. The actual checking time does not decrease by the same factor, because the fewer inner iterations are expected to correspond on average to more complicated formulae, since they are more likely to contribute to progress (some iterations in the ordinary progress computation are more likely to terminate very quickly since they do not contribute to the desired progress). Also, some implementation specific issue may play a role; for instance, the hit ratio of the *and-exists* cache was significantly worse for the generalized progress checking than for the ordinary one. A different chaching strategy or cache organization could lead to even better results.

It should be pointed out again that the observed results confirm that the increase in performance observed when using regular expression hints is closely related to the ratio of the number of actions that contribute to progress versus the total number of actions. Even in a program like the above that does not have a behavior that can be described in sequential phases, the possibility of restricting

one's attention to the actions that are helpful for the property under consideration produces significant performance improvements.