PranCS: A Protocol and Discrete Controller Synthesis Tool

Idress Husien, Sven Schewe, and Nicolas Berthier(✉)

Department of Computer Science, University of Liverpool, Liverpool, UK
nicolas.berthier@liverpool.ac.uk

Abstract. PranCS is a tool for synthesizing protocol adapters and discrete controllers. It exploits general search techniques such as simulated annealing and genetic programming for homing in on correct solutions, and evaluates the fitness of candidates by using model-checking results. Our **Pr**octocol **an**d **C**ontroller **S**ynthesis (PranCS) tool uses NuSMV as a back-end for the individual model-checking tasks and a simple candidate mutator to drive the search.

PranCS is also designed to explore the parameter space of the search techniques it implements. In this paper, we use PranCS to study the influence of turning various parameters in the synthesis process.

1 Introduction

Discrete Controller Synthesis (DCS) and *Program Synthesis* have similar goals: they are automated techniques to infer a control strategy and an implementation, respectively, that is correct by construction.

There are mild differences between these two classes of problems. DCS typically operates on the model of a plant. It seeks the automated construction of a strategy to control the plant, such that its runs satisfy a set of given objectives [2,22]. Similarly, program synthesis seeks to infer an implementation, often of a reactive system, such that the runs of this system satisfy a given specification [21]. Program synthesis is particularly attractive for the construction of protocols that govern the intricate interplay between different threads; we use mutual exclusion and leader election as examples.

Apart from their numerous applications to manufacturing systems [19,22,24], DCS algorithms have been used to enforce fault-tolerance [11], deadlock avoidance in multi-threaded programs [23], and correct resource management in embedded systems [1,3].

Foundations of DCS and program synthesis are similar to principles of model-checking [5,8]. Model-checking refers to automated techniques that determines whether or not a system satisfies a number of specifications. Traditional DCS algorithms are inspired by this approach. Given a model of the plant, they first *exhaustively* compute an unsafe portion of the state-space to avoid for the desired

This work was supported by the Ministry of Higher Education in Iraq through the University of Kirkuk and by the EPSRC through grant EP/M027287/1.

K.G. Larsen et al. (Eds.): SETTA 2017, LNCS 10606, pp. 337–349, 2017.
https://doi.org/10.1007/978-3-319-69483-2_20

objectives to be satisfied, and then derive a strategy that avoids entering the unsafe region. Finally, a controller is built that restricts the behaviour of the plant according to this strategy, so that it is guaranteed to always comply with its specification. Just as for model-checking, *symbolic* approaches for solving DCS problems have been successfully investigated [2,4,10,20].

Techniques based on genetic programming [7,12–17], as well as on simulated annealing [13,14], have been tried for program synthesis. Instead of performing an exhaustive search, these techniques proceed by using a measure of the fitness—reflecting the question "*How close am I to satisfying the specification?*"—to find a short path towards a solution. Among the generic search techniques that look promising for this approach, we focus on *genetic programming* [18] and *simulated annealing* [7,12]. When applied to program synthesis, both search techniques work by successively mutating candidate programs that are deemed "good" by using some measure of their fitness. We obtain their fitness for meeting the desired objectives by using a model-checker to measure the share of objectives that are satisfied by the candidate program, *cf.* [13,14,16,17].

Simulated annealing keeps one candidate solution, and a "cooling schedule" describes the evolution of a "temperature". In a sequence of iterations, the algorithm mutates the current candidate and compares the fitness of the old and new candidate. If the fitness increases, the new candidate is always maintained. If it decreases, a random process decides if the new candidate replaces the old one in the next iteration. The chances of the new candidate to replace the old one then decrease with the gap in the fitness and increase with the temperature; thus, a lower temperature makes the system "stiffer".

Genetic programming maintains a population of candidate programs over a number of iterations. In each iteration, new candidate programs are generated by mutation or by mixing randomly selected candidates ("crossover"). At the end of each iteration, the number of candidates under consideration is shrunken back to the original number. A higher fitness makes it more likely for a candidate to survive this step.

In Sect. 2, we describe the tool PranCS, which implements the simulated annealing based approach proposed in [13,14] as well as approaches based on similar genetic programming from [16,17]. PranCS uses quantitative measures for partial compliance with a specification, which serve as a measure for the fitness (or: quality) of a candidate solution. Furthering on the comparison of simulated annealing with genetic programming [13,14], we extend the quest for the best general search technique in Sect. 3 by:

1. looking for good cooling schedules for simulated annealing; and
2. investigating the impact of the population size and crossover ratio for genetic programming.

2 Overview of PranCS

PranCS implements several generic search algorithms that can be used for solving DCS problems as well as for synthesising programs.

2.1 Representing Candidates

The representation of candidates depends on the kind of problems to solve. Candidate programs are represented as abstract syntax trees according to the grammar of the sought implementation. They feature conditional and iteration statements, assignments to one variable taken among a given set, and expressions involving such variables. Candidates for DCS only involve a series of assignments to a given subset of Boolean variables involved in the system (called "*controllables*").

2.2 Structure of PranCS

The structure of PranCS is shown in Fig. 1. Via the user interface, the user can select a search technique, and enter the problem to solve along with values for relevant parameters of the selected algorithm. For program synthesis, the user enters the number, size, and type of variables that candidate implementations may use, and whether thay may involve complex conditional statements ("if" and "while" statements). DCS problems are manually entered as a series of assignments to state variables involving expressions expressed on state and input variables; the user also lists the subset of input variables that are "controllable". In both cases, the user also provides the specification as a list of objectives.

Generator. The Generator uses the parameters provided to either generate new candidates or to update them when required during the search.

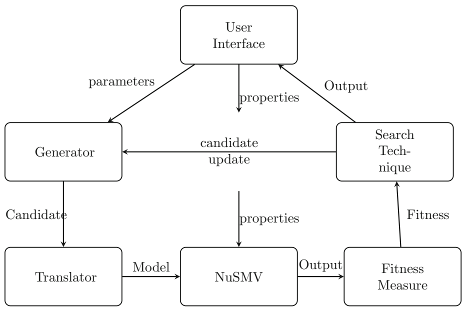

Fig. 1. Overview of PranCS.

Translator & NuSMV. We use NuSMV [6] as a model-checker. Every candidate is translated into the modelling language of NuSMV using a method suggested by Clark and Jacob [7]. (We detail this translation for programs and plants in [14] and [13] respectively, and give an example program translation in Appendix A.) The resulting model is then model-checked against the desired properties. The result forms the basis of a fitness function for the selected search technique.

Fitness Measure. To design a fitness measure for candidates, we make the hypothesis that the share of objectives that are satisfied so far by a candidate is a good indication of its suitability *w.r.t.* the desired specification. We additionally observe that weaker properties that can be mechanically derived are useful to identify good candidates worth selecting for the generation of further potential solutions. For example, if a property shall hold on all paths, it is better if it holds on some path, and even better if it holds almost surely.

Search Technique. The fitness measure obtained for a candidate is used as a fitness function for the selected search technique. If a candidate is evaluated as correct, we return (and display) it to the user. Otherwise, depending on the search technique selected and the old and new fitness measure/s, the current candidate or population is updated, and one or more candidates are sent for change to the Generator. The process is re-started if no solution has been found in a predefined number of steps (genetic programming) or when the cooling schedule expires (simulated annealing).

2.3 Selecting and Tuning Search Techniques

In terms of search techniques, PranCS implements the following methods: *genetic programming*, and *simulated annealing*. Katz and Peled [17] extend genetic programming by considering the fitness as a pair of "safety-fitness" and "liveness-fitness", where the latter is only used for equal values of "safety-fitness". Building upon this idea, we define two flavours for both simulated annealing and genetic programming: *rigid* (where the classic fitness function is used) and *safety-first*, which uses the two-step fitness approach as above. Further, genetic programming can be used with or without crossovers between candidates [13,14].

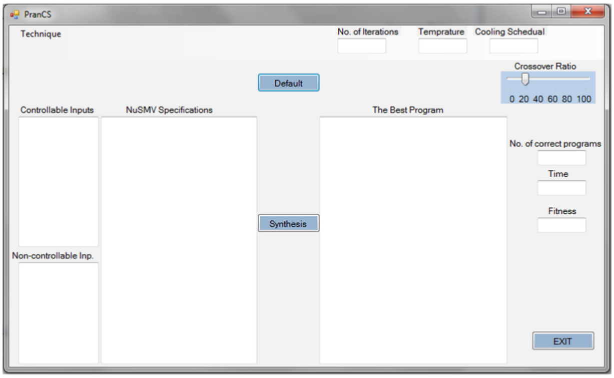

Fig. 2. Graphical User Interface. PranCS allows the user to fine-tune each search technique by means of dedicated parameters.

Depending on the selected search technique, the tool allows the user to input parameters that control the dynamics of the synthesis process. These parameters determine the likelihood of finding a correct program in each iteration and the expected running time for each iteration, and thus heavily influence the overall search speed. For the genetic programming approach, the parameters include the population size, the number of selected candidates, the number of iterations, and the crossover ratio. For simulated annealing, the user chooses the initial temperature and the cooling schedule. Figure 2 shows the graphical user interface of PranCS.

Parameters for Simulated Annealing. In simulated annealing (SA), the intuition is that, at the beginning of the search phase, the temperature is high, and it cools down as time goes by. The higher the temperature, the higher is the likelihood that a new candidate solution with inferior fitness replaces the previous solution. While this allows for escaping local minima, it can also happen that the candidates develop into an undesirable direction. For this reason, simulated annealing does not continue for ever, but is re-started at the end of the cooling schedule. Consequently, there is a sweet-spot in just how long a cooling schedule should be and when it becomes preferable to re-start, but this sweet-spot is difficult to find. We report our experiments with PranCS for tuning the cooling schedule in Sect. 3.1.

Parameters for Genetic Programming. For Genetic Programming (GP), the parameters are the initial population size, the crossover vs mutation ratio, and the fitness measure used to select the individuals. The population size affects the algorithm in two ways: a larger population size could provide better diversity and reduce the number of iterations required or, for a fixed number of iterations, increase the likelihood of finding a solution. However, it also increases the time spent for each individual iteration. The crossover ratio describes the amount of new candidates that are generated by mating. Crossovers allow for the appearance of solutions that synthesise the best traits of good candidates, and a high crossover ratio promises to make this more likely. This requires, however, a high degree of diversity in the population, where these traits need to draw from different parts of the program tree, and it comes to the cost of creating diversity through a reduction of the number of mutations applied in each iteration.

We investigate how the population size and crossover ratio affect the performance of these algorithms in Sects. 3.2 and 3.3.

3 Exploration of the Parameter Space

Besides serving as a synthesis tool, PranCS provides the user with the ability to compare various search techniques. In [13,14], we have carried out experiments by applying our algorithms to generate correct solutions on benchmarks comprising mutual exclusion, leader election, and DCS problems of growing size and complexity. With parameter values borrowed from [16,17], we could already accelerate synthesis significantly using simulated annealing compared to genetic programming (by 1.5 to 2 orders of magnitude).

In this paper, our aim is to further explore the performance impact of the parameters for each search technique. We thus reuse the same scalable benchmarks as in [13,14]: program synthesis problems consist of mutual exclusion ("2 or 3 shared bits") and leader election ("3 or 4 nodes"); DCS problems compute controllers enforcing mutual exclusions and progress between 1 to 6 tasks modelled as automata ("1 through 6-Tasks").

In all Tables, execution times are in seconds; $\bar{t}$ is the mean execution time of single executions (succeeding or failing), and columns T extrapolate $\bar{t}$ based on the success rate obtained in 100 single executions (columns "%").

3.1 Exploring Cooling Schedules for Simulated Annealing

In order to test if the hypothesis from [9] that simulated annealing does most of its work during the middle stages—while being in a good temperature range—

holds for our application, we have developed the tool to allow for "cooling schedules" that do not cool at all, but use a constant temperature. In order to be comparable to the default strategy, we use up to 25,001 iterations in each attempt.

We have run 100 attempts to create a correct candidate using various constant temperatures, and inferred expected overall running times T based on the success rates and average execution time of single executions $\bar{t}$. We first report the results for program synthesis and DCS problems in Tables 1 and 2 respectively.

Table 1. Impact of search temperature (θ) for program synthesis with safety-first simulated annealing

θ	3 nodes			4 nodes			2 shared bits			3 shared bits		
	$\bar{t}$	%	T	$\bar{t}$	%	T	$\bar{t}$	%	T	$\bar{t}$	%	T
0.7	316	0	∞	521	0	∞	147	0	∞	155	0	∞
400	285	0	∞	493	0	∞	143	0	∞	148	0	∞
4,000	196	11	1,781	368	10	3,680	129	3	4,300	121	4	3,025
7,000	97	14	692	314	13	2,415	77	12	641	81	11	252
10,000	73	21	347	138	18	766	15	22	68	17	24	70
13,000	78	22	354	146	19	768	16	23	69	18	24	75
16,000	83	20	415	150	17	882	17	21	80	19	22	86
20,000	87	19	457	153	15	1,020	21	20	105	23	22	104
25,000	94	17	494	167	13	1,284	23	19	121	25	21	191
30,000	108	15	720	184	11	1,672	28	18	155	30	19	157
40,000	117	15	780	193	11	1,754	31	16	193	34	17	200
50,000	129	13	992	201	10	2,010	37	15	246	41	16	256
100,000	193	12	1,608	287	9	3,188	52	11	472	58	13	446

The findings support the hypothesis that some temperatures are much better suited than others: low temperatures provide a very small chance of succeeding, and the chances also go down at the high temperature end.

While the values for low temperatures are broadly what we had expected, the high end performed better than we had thought. This might be because some small guidance is maintained even for infinite temperature, as a change that is decreasing the fitness is taken with an (almost) 50% chance in this case, while increases are always selected. However, the figures for high temperatures are much worse than the figures for the good temperature range of 10,000 to 16,000.

In the majority of cases, the best results have been obtained at a temperature of 10,000. Notably, these results are better than the running time for the cooling schedule that uses a linear decline in the temperature as used and reported in [13, 14]. They indicate that it seems likely that the last third of the improvement cycles in this cooling schedule had little avail, especially for smaller problems.

Table 2. Impact of search temperature (θ) for DCS with Safety-first simulated annealing

	1-Task			2-Tasks			3-Tasks			4-Tasks			5-Tasks			6-Tasks		
θ	$\bar{t}$	%	T	$\bar{t}$	%	T	$\bar{t}$	%	T	$\bar{t}$	%	T	$\bar{t}$	%	T	$\bar{t}$	%	T
0.7	163	0	∞	177	0	∞	192	0	∞	332	0	∞	298	0	∞	613	0	∞
400	93	0	∞	99	0	∞	163	0	∞	167	0	∞	153	0	∞	598	0	∞
4,000	54	7	771	58	6	966	88	6	1,466	98	3	3,266	98	4	2,450	278	3	9,266
7,000	39	12	325	47	9	522	45	9	500	65	6	1,083	79	6	1,316	125	5	2,500
10,000	18	19	94	29	14	207	26	11	236	39	9	433	61	9	677	99	8	1,237
13,000	22	20	110	33	15	220	31	11	281	43	11	390	67	10	670	115	9	1,277
16,000	29	19	152	39	13	300	37	10	370	58	9	644	73	8	912	127	9	1,411
20,000	37	17	217	47	11	427	42	10	420	67	9	744	81	6	1,350	134	7	1,914
25,000	43	15	286	56	10	560	47	9	522	81	7	1,157	89	6	1,483	152	6	2,533
30,000	49	15	326	67	10	670	56	8	700	89	6	1,483	102	4	2,550	159	6	2,650
40,000	53	13	407	75	9	833	63	9	700	95	6	1,583	116	3	3,866	168	6	2,800
50,000	59	12	491	82	7	1,171	79	7	1,128	103	5	2,060	128	4	3,200	192	5	3,840
100,000	72	11	654	94	7	1,342	98	7	1,400	118	4	2,950	178	3	5,933	253	4	6,325

A robust temperature sweet-spot clearly exists for our scalable benchmarks, suggesting that the quest for robust and generic good cooling schedules is worth pursuing.

3.2 Impact of Population Size for Genetic Programming

One of the important parameters of genetic programming is the initial population size; another parameter worth tuning is the number of candidates η selected for mating at each iteration of the algorithm. In order to investigate their effects on our synthesis approach and evaluate the actual cost of large population sizes, we defined several setups with various values for the population size $|P|$ and amount of mating candidates η. We then performed 100 executions of our GP-based algorithms with each of these setups for the 2 shared bits mutual exclusion and 2-Tasks problems.

We show the results in Tables 3 and 4. As expected, increasing the size of the initial population also dramatically increases the cost of finding a good solution. Broadly speaking, increasing the population size reduces the number of iterations and increases the success rate, but it also increases the computation time required at each individual iteration. Smaller population sizes appear to benefit individual running times more than they harm success rates.

The impact of η on performance appears very limited on the range we have investigated.

Table 3. Impact of population size ($|P|$) for Program Synthesis (2 shared bits mutual exclusion only)

		Rigid GP						Safety-first GP					
		w/o crossover			with crossover			w/o crossover			with crossover		
$\|P\|$	η	$\bar{t}$	%	T	$\bar{t}$	%	T	$\bar{t}$	%	T	$\bar{t}$	%	T
150	5	583	7	8,328	589	9	6,544	113	31	364	115	33	348
	7	583	7	8,328	589	9	6,544	113	31	364	115	33	348
	9	584	7	8,342	588	9	6,533	113	31	364	114	33	345
250	5	1,024	12	8,533	1,057	15	7,046	230	46	500	245	49	500
	7	1,024	12	8,533	1,057	15	7,046	230	46	500	245	49	500
	9	1,024	12	8,533	1,057	15	7,046	231	46	502	245	49	500
350	5	1,435	15	9,566	1,451	18	8,061	325	63	515	367	67	547
	7	1,435	15	9,566	1,451	18	8,061	325	63	515	366	67	546
	9	1,435	15	9,566	1,451	19	7,636	325	64	507	367	67	547

Table 4. Impact of population size ($|P|$) for DCS (2-Tasks only)

		Rigid GP						Safety-first GP					
		w/o crossover			with crossover			w/o crossover			with crossover		
$\|P\|$	η	$\bar{t}$	%	T	$\bar{t}$	%	T	$\bar{t}$	%	T	$\bar{t}$	%	T
150	5	463	3	15,433	484	4	12,100	132	13	1,015	138	15	920
	7	463	3	15,433	485	4	12,125	132	13	1,015	139	15	926
	9	464	3	15,466	485	4	12,125	131	13	1,007	139	14	992
250	5	943	5	18,860	969	7	13,842	241	18	1,338	218	19	1,147
	7	943	5	18,860	969	7	13,842	241	18	1,338	218	19	1,147
	9	943	5	18,860	969	7	13,842	242	18	1,344	218	19	1,147
350	5	1,517	9	16,855	1,557	10	15,570	403	24	1,679	340	24	1,416
	7	1,517	9	16,855	1,557	10	15,570	403	24	1,679	340	24	1,416
	9	1,518	9	16,866	1,557	10	15,570	403	24	1,679	340	24	1,416

3.3 Impact of Crossover Ratio for Genetic Programming

Finally, we have also studied the effect of changing the share between crossover and mutation in genetic programming.

We report our results in Tables 5 and 6. Interestingly, the running time per instance increased with the share of crossovers, which might point to a production of more complex candidate solutions. Regarding expected running times, the results also indicate the existence of a sweet-spot for the crossover ratio at around 20% for both Rigid and Safety-first variants of the algorithm.

Table 5. Impact of crossover ratio (ρ, in percent) for Program Synthesis with Rigid and Safety-first GP

	Rigid GP				Safety-first GP		
	ρ	$\bar{t}$	%	T	$\bar{t}$	%	T
2 shared bits	0	583	7	8,328	113	31	364
	20	589	9	6,544	115	33	348
	40	602	9	6,688	123	33	372
	60	614	8	7,657	134	33	406
	80	613	8	7,662	142	21	676
	100	652	2	32,600	151	5	3,020
3 shared bits	0	615	7	8,785	171	17	1,005
	20	620	9	6,888	175	19	921
	40	637	9	7,077	187	19	984
	60	658	8	8,225	196	19	1,031
	80	669	4	16,725	207	11	1,881
	100	682	2	34,100	223	3	7,433
3 nodes	0	1,120	3	37,333	418	15	2,786
	20	1,123	6	18,716	421	16	2,631
	40	1,137	5	22,740	427	16	2,668
	60	1,149	5	22,980	453	13	3,484
	80	1,154	3	38,466	469	9	5,211
	100	1,167	2	58,350	487	4	12,175
4 nodes	0	1,311	3	43,700	536	11	4,872
	20	1,314	5	26,280	541	14	3,864
	40	1,325	4	33,125	557	13	4,284
	60	1,336	3	44,533	569	13	4,376
	80	1,345	3	44,833	581	9	6,455
	100	1,353	2	67,650	593	3	17,966

Table 6. Impact of crossover ratio (ρ, in percent) for DCS with Rigid and Safety-first GP

	Rigid GP				Safety-first GP		
	ρ	$\bar{t}$	%	T	$\bar{t}$	%	T
1-Task	0	378	4	9,450	89	17	523
	20	385	5	7,700	94	20	470
	40	403	5	8,060	101	19	531
	60	418	4	10,450	109	19	573
	80	425	3	14,166	116	12	966
	100	438	1	43,800	124	5	2,480

(*continued*)

Table 6. (*Continued*)

	Rigid GP				Safety-first GP		
	ρ	$\bar{t}$	%	T	$\bar{t}$	%	T
2-Tasks	0	475	3	15,833	127	13	976
	20	484	4	12,100	138	15	920
	40	491	4	12,275	146	15	973
	60	501	3	16,700	158	13	1,215
	80	509	2	25,450	169	11	1,536
	100	521	1	52,100	181	4	4,525
3-Tasks	0	571	3	19,033	189	9	2,100
	20	589	4	14,725	201	11	1,827
	40	597	3	19,900	209	11	1,900
	60	606	3	20,200	217	8	2,712
	80	613	1	61,300	225	7	3,214
	100	627	1	62,700	239	3	7,966
4-Tasks	0	658	3	21,933	288	9	3,200
	20	664	4	16,600	296	12	2,466
	40	679	4	16,975	303	11	2,754
	60	687	3	22,900	313	10	3,130
	80	693	2	34,650	321	8	4,012
	100	711	1	71,100	333	4	8,325
5-Tasks	0	776	1	77,600	438	7	6,257
	20	787	3	26,233	445	11	4,045
	40	792	3	26,400	451	8	5,637
	60	799	2	39,950	459	7	6,557
	80	804	2	40,200	467	5	9,340
	100	815	1	81,500	479	2	23,950
6-Tasks	0	961	2	48,050	659	6	10,983
	20	972	3	32,400	673	10	6,730
	40	981	2	49,050	679	10	6,790
	60	989	2	49,450	695	7	9,928
	80	997	2	49,850	703	4	17,575
	100	1,011	1	101,100	718	2	35,900

4 Conclusion

Together with our extensive exploration of the parameter space, the evaluation of PranCS indicates that simulated annealing is faster than genetic programming (we report some synthesis times with the best parameters observed using simulated annealing in Table 7), and that some temperature ranges are more useful than others. Additional information about the tool can be found at: https://cgi.csc.liv.ac.uk/~idresshu/index2.html.

Table 7. Synthesis times with the best parameters observed for Simulated Annealing with linearly decreasing cooling schedule applied to our DCS benchmarks; results for row "2-Tasks" should be compared with best results reported in Table 4 for solving the same DCS benchmark problem using GP-based algorithms.

	Rigid SA			Safety-first SA		
	$\bar{t}$	%	T	$\bar{t}$	%	T
1-Task	20	13	153	19	16	**118**
2-Tasks	25	10	250	24	13	**184**
3-Tasks	33	9	366	29	10	**290**
4-Tasks	47	9	522	43	9	**477**
5-Tasks	76	8	950	70	9	**777**
6-Tasks	119	7	1,700	106	7	**1,514**

In order to integrate this result into the cooling schedule we plan to use an adaptive cooling schedule, in which the decrements of the temperature depends on the improvement of the fitness.

Appendix A Pseud-Code to NuSMV Translation Example

To evaluate the fitness of the produced program, it is first translated into the language of the model checker NuSMV [6]. We have used the translation method suggested by Clark and Jacob [7].

In this translation, the program is converted into very simple statements, similar to assembly language. To simplify the translation, the program lines

```
process me
while (true) do
   noncritical section
   while (turn==me) do
      skip
   end while
   critical section
   turn=other
end while
```

'me' and 'other' denote (different) variable valuations, in this example implemented as boolean variables. In other instances, they might be have a different (finite) datatype.

```
MODULE p(turn)
VAR
  pc: {11, 12, 14,15};
ASSIGN
  init(pc) := 11;
  next(pc) := case
    (pc=11) : {11, 12};
    (pc=12)&(turn=me) : 14;
    (pc=14) : 15;
    (pc=15) : 11;
    TRUE: pc;
  esac;
  next(turn):= case
    (pc=15): other;
    TRUE :turn;
  esac;
```

Fig. 3. Translation example – source pseudo-code (left) and target NuSMV (right)

are first labeled, and this label is then used as a pointer that represents the program counter *(PC)*. From this intermediate language, the NuSMV model is built by creating *(case)* and *(next)* statements that use the *PC*. Figure 3 shows the translation of a mutual exclusion algorithm.

References

1. Altisen, K., Clodic, A., Maraninchi, F., Rutten, E.: Using controller-synthesis techniques to build property-enforcing layers. In: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 174–188. Springer, Heidelberg (2003). doi:10.1007/3-540-36575-3_13
2. Asarin, E., Maler, O., Pnueli, A.: Symbolic controller synthesis for discrete and timed systems. In: Antsaklis, P., Kohn, W., Nerode, A., Sastry, S. (eds.) HS 1994. LNCS, vol. 999, pp. 1–20. Springer, Heidelberg (1995). doi:10.1007/3-540-60472-3_1
3. Berthier, N., Maraninchi, F., Mounier, L.: Synchronous Programming of Device Drivers for Global Resource Control in Embedded Operating Systems. ACM Trans. Embed. Comput. Syst. 12(1s), 39: 1–39: 26., March 2013
4. Berthier, N., Marchand, H.: Discrete controller synthesis for infinite state systems with ReaX. In: 12th Internation Workshop on Discrete Event Systems. WODES 20114, IFAC, pp. 46–53, May 2014
5. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. **98**(2), 142–170 (1992)
6. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). doi:10.1007/3-540-45657-0_29
7. Clark, J.A., Jacob, J.L.: Protocols are programs too: the meta-heuristic search for security protocols. Inf. Softw. Technol. **43**, 891–904 (2001)
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
9. Connolly, D.: An improved annealing scheme for the qap. Eur. J. Oper. Res. **46**, 93–100 (1990)
10. Cury, J.E., Krogh, B.H., Niinomi, T.: Synthesis of supervisory controllers for hybrid systems based on approximating automata. IEEE Trans. Autom. Control **43**(4), 564–568 (1998)
11. Girault, A., Rutten, É.: Automating the addition of fault tolerance with discrete controller synthesis. Formal Methods Syst. Des. **35**(2), 190 (2009)
12. Henderson, D., Jacobson, S.H., Johnson, A.W.: The theory and practice of simulated annealing. In: Glover, F., Kochenberger, G.A. (eds.) Handbook of Metaheuristics, International Series in Operations Research & Management Science, vol. 57, pp. 287–319. Springer, Boston (2003). doi:10.1007/0-306-48056-5_10
13. Husien, I., Berthier, N., Schewe, S.: A hot method for synthesising cool controllers. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software. SPIN 2017, pp. 122–131. ACM, New York (2017)
14. Husien, I., Schewe, S.: Program generation using simulated annealing and model checking. In: De Nicola, R., Kühn, E. (eds.) SEFM 2016. LNCS, vol. 9763, pp. 155–171. Springer, Cham (2016). doi:10.1007/978-3-319-41591-8_11

15. Johnson, C.G.: Genetic programming with fitness based on model checking. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 114–124. Springer, Heidelberg (2007). doi:10.1007/978-3-540-71605-1_11
16. Katz, G., Peled, D.: Model checking-based genetic programming with an application to mutual exclusion. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 141–156. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_11
17. Katz, G., Peled, D.: Model checking driven heuristic search for correct programs. In: Peled, D.A., Wooldridge, M.J. (eds.) MoChArt 2008. LNCS (LNAI), vol. 5348, pp. 122–131. Springer, Heidelberg (2009). doi:10.1007/978-3-642-00431-5_8
18. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
19. Krogh, B.H., Holloway, L.E.: Synthesis of feedback control logic for discrete manufacturing systems. Automatica **27**(4), 641–651 (1991)
20. Marchand, H., Bournai, P., Le Borgne, M., Le Guernic, P.: Synthesis of discrete-event controllers based on the signal environment. Discrete Event Dynamic Syst. Theory Appl. **10**(4), 325–346 (2000)
21. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL 1989. pp. 179–190. ACM, New York (1989)
22. Ramadge, P., Wonham, W.: The control of discrete event systems. Proc. IEEE Spec. Issue Dyn. Discr. Event Syst. **77**(1), 81–98 (1989)
23. Wang, Y., Lafortune, S., Kelly, T., Kudlur, M., Mahlke, S.: The theory of deadlock avoidance via discrete control. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 252–263. POPL 2009. ACM, New York (2009)
24. Zhou, M., DiCesare, F.: Petri Net Synthesis for Discrete Event Control of Manufacturing Systems, vol. 204. Springer Science & Business Media, Heidelberg (2012). doi:10.1007/978-1-4615-3126-5