ADAM: Causality-Based Synthesis of Distributed Systems*

Bernd Finkbeiner[1], Manuel Gieseking[2], and Ernst-Rüdiger Olderog[2]

[1] Universität des Saarlandes
[2] Carl von Ossietzky Universität Oldenburg

Abstract. We present ADAM, a tool for the automatic synthesis of distributed systems with multiple concurrent processes. For each process, an individual controller is synthesized that acts on locally available information obtained through synchronization with the environment and with other system processes. ADAM is based on Petri games, an extension of Petri nets where each token is a player in a multiplayer game. ADAM implements the first symbolic game solving algorithm for Petri games. We report on experience from several case studies with up to 38 system processes.

1 Introduction

Research on the *reactive synthesis* problem, i.e., the challenge of constructing a reactive system automatically from a formal specification, dates back to the early years of computer science [6, 5, 16]. Over the past decade, this research has led to tools like Acacia+ [4], Ratsy [3], and Unbeast [7], which translate a formal specification automatically into an implementation that is correct in the sense that the system reacts to every possible input from the system's environment in a way that ensures that the specification is satisfied. The tools have been used in nontrivial applications, such as the synthesis of bus arbiter circuits [2] and robotic control [12]. The key limitation of the current state of the art is that the underlying system model consists of a single process. If the system under construction consists of several distributed parts, such as several robots, then the implementation is always based on a central controller with whom the entire system must constantly synchronize. This is unfortunate, because in practice, it is specifically the design of the *distributed* implementation with multiple concurrent processes that is most error-prone and would, therefore, benefit most from a synthesis tool.

In this paper, we present ADAM, a synthesis tool designed for distributed systems with multiple concurrent processes. Unlike previous tools based on automata, ADAM uses concurrent processes in the form of Petri nets as its underlying system model. (ADAM is named in honor of Carl *Adam* Petri.) Our aim

* This research was partially supported by the German Research Council (DFG) in the Transregional Collaborative Research Center SFB/TR 14 AVACS.

is to automate the construction of complex distributed systems, such as production plants with multiple independent robots. Rather than creating a central controller, with whom all robots must constantly synchronize, ADAM creates an individual controller for each robot, which acts on locally available information such as the information obtained through observations and through synchronization with nearby robots.

The most well-studied model for the synthesis of distributed systems is due to Pnueli and Rosner [14]. This model captures the *partial information* available to the processes by specifying for each process the subset of events that are visible to the process. The decisions of the process are based only on the history of its observations, not on the full state history. Unfortunately, the Pnueli/Rosner model has never been translated into practical tools; the synthesis problem under the Pnueli/Rosner model is undecidable in general, and very expensive (nonelementary) in the special cases where it can be decided [15, 10].

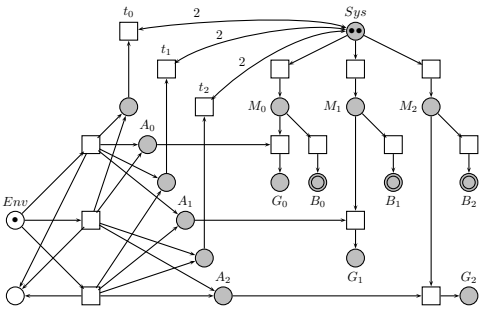

Fig. 1. Example Petri game from the synthesis of two independent robots in a manufacturing situation with k orders on n machines (here: $k = 2, n = 3$), where one machine is chosen by the environment to be defective.

ADAM is based on the more recently developed model of *Petri games* [9]. The synthesis problem is modeled as a game between a team of system players, representing the processes, and an environment (player), representing the user (and other external influences) of the system. Both the system players and the environment are represented as tokens of a Petri net. As Petri nets, the games capture the complex *causal dependencies* (and independence) between the processes (cf. [17]). Figure 1 shows a typical application scenario, taken here from the synthesis of robot controllers in a production plant, addressing concurrency, usage constraints, and uncertain availability of machines. The robots are expected to process k orders on n machines (here: $k = 2, n = 3$), despite the actions of a hostile environment, which is allowed to declare one machine to be defective. The environment, initially at place *Env*, chooses which machine is defective and activates the remaining machines by putting tokens on two of the places A_i, $i \in \{0, 1, 2\}$. The two system players in place *Sys* represent the two robots. Different robots can take their orders concurrently to different machines. If a robot chooses a machine M_i right away, it does not know whether M_i is defective, i.e., without a token on A_i. Then from M_i only the bad place B_i is reachable. If a robot chooses an active machine M_i (with a token on A_i) then from M_i the good place G_i is reachable by consuming the token from A_i. If a robot chooses M_i again, the token on A_i is missing, and only the bad place B_i is reachable. A winning strategy for the robots must avoid any transitions to a

bad place. To this end, the robots first inform themselves, via the synchronizing transitions t_0, t_1 and t_2, which machines are broken (this is done simultaneously by the two robots due to the arc multiplicity 2) and then use two different active machines.

In recent work [9], we showed that solving Petri games with safety objectives, a single environment player and an arbitrary (but fixed) number of system players is EXPTIME-complete, and thus dramatically cheaper than comparable synthesis problems in the Pnueli/Rosner setting. ADAM represents the first practical implementation of this theoretical result.

2 The Synthesis Game

We model the synthesis problem as a game between a team of system players on one side and a hostile environment player on the other side. The system players have a joint objective, to defeat the environment, but are independent of each other in the sense that they have no information of each other's state unless they explicitly communicate. A *Petri game* [9] is a refinement of a Petri net. The players are the tokens in the underlying Petri net. They are organized into two teams, the system players and the environment players, where the system players wish to avoid a certain "bad" place (i.e., they follow a safety objective), while the environment players wish to reach just such a marking. To partition the tokens into the teams, we distinguish each place p as belonging to either the system ($p \in \mathcal{P}_S$) or the environment ($p \in \mathcal{P}_E$). A token belongs to one of these teams whenever it is on a place that belongs to that team. Formally, a Petri game is a structure $\mathcal{G} = (\mathcal{P}_S, \mathcal{P}_E, \mathcal{T}, \mathcal{F}, In, \mathcal{B})$, where the underlying Petri net of $\mathcal{G}$ is $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, In)$ with set of places $\mathcal{P} = \mathcal{P}_S \cup \mathcal{P}_E$, set of transitions $\mathcal{T}$, flow relation $\mathcal{F}$, initial marking In, and set of bad places $\mathcal{B}$. We depict places of $\mathcal{P}_S$ in gray and of $\mathcal{P}_E$ in white. In the following, we assume that there is a single environment player and an arbitrary (but bounded) number of system players. We further assume that the Petri net is safe, i.e., every place is, at all times, occupied by at most one token. Petri games that are bounded, but not necessarily safe, like the example from the introduction, can be translated into an equivalent game with a safe net using the standard transformation.

A player (token) is always informed about its causal past. As long as different players move in concurrent places of the net, they do not know of each other. Only when they communicate, i.e., synchronize at a joint transition, they exchange their knowledge about the past. Formally, this is modelled by the net unfolding.

A *strategy* σ for the system players will eliminate at each place of the net unfolding some of the available branches. A strategy is *winning* for the system players if all branches that lead to a bad place are eliminated. For each player we can obtain a *local controller* by isolating the part of σ that is relevant for this player. These local controllers can proceed independently unless they have to synchronize at a joint transition with other local controllers as described by σ. Since the winning condition of a game is a *safety objective*, the system players can satisfy it by doing nothing. To avoid such trivial solutions, we look for

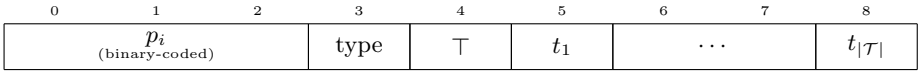

Fig. 2. Bitvector representation of a cut. The subvector encodes the ith system token.

strategies that are *deadlock-avoiding* in the sense that in every reachable marking, whenever there is a transition enabled in the unfolding then there is some transition enabled in the strategy. A marking where there is no enabled transition in the unfolding either is not a deadlock. Then we say that the game has *terminated*. A *play* π (conforming to a strategy σ) is obtained from σ by eliminating all remaining choices such that at each place there is only one transition left (determinism). The system players win the play π if it does not contain a bad place. Otherwise, the environment wins.

3 Solving Petri Games

Petri games can be solved via a reduction to two-player games over finite graphs. In this section, we give an informal sketch of the reduction, focusing on the symbolic representation and the fixed point iteration of the game solving algorithm. For a more formal presentation of the reduction from Petri games to two-player games over finite graphs, the reader is referred to [9].

The two-player game simulates the Petri game through a sequence of *cuts*, i.e., maximal sets of concurrent places. We annotate the system places in a cut with a *decision set*, i.e., a set of transitions currently selected by the player represented by the token on the system place. In each cut, we designate a subset of the system places as *type-2*, which means that its strategy will no longer synchronize (directly or indirectly through other system tokens) with the environment. Additionally, we designate a subset of the system places as *type-1*. These are places that still require a synchronization with the environment but are, in the current cut, not able to move (following their decision sets) before the environment makes its next move.

Cuts where all system places are either type-1 or type-2 are called *mcuts*. Mcuts correspond to situations in which the system players have progressed maximally in the sense that all non-type-2 places are blocked until the environment moves. The key idea of the reduction is to delay all environment decisions until an mcut is reached. This ensures that all system decisions that should be made independently of the environment choice have actually been made *before* the environment decision is made, and are, hence, guaranteed to be independent of this decision. A winning strategy for the system players must thus legally move from mcut to mcut, in response to the environment decisions at the mcuts, without encountering bad situations (such as bad places), either until the Petri net terminates or forever, if the play never terminates.

The symbolic representation of cuts. Our representation of a cut is organized by the tokens, rather than places: this is motivated by the fact that the number of tokens in a Petri net is usually much smaller than the number of places; it is

therefore cheaper to assign to each token the currently occupied place instead of simply representing an (arbitrary) subset of the places. A cut is represented as a bitvector, which is composed of several subvectors, one for each token. Figure 2 depicts such a subvector for a system token i. The first part of the bitvector encodes the place p_i and its type (type-1 vs. type-2). The second part encodes the decision of the strategy. The bit t_j is set iff the player represented by token i chooses to allow the jth transition of the Petri net. The $\top$-bit is set right after a transition is executed. It indicates that the player is allowed to choose a new set of transitions. For the special case of the environment token, we only need to encode the place, without the type, $\top$-, and transition flags. We use BDDs to represent sets of cuts and relations on cuts.

The game solving algorithm. The game solving algorithm consists of three phases. *Phase 1* is a preprocessing step that identifies the type-2 places in the cuts. The strategy from type-2 places must guarantee that the tokens on the type-2 places have no further interaction with the environment. The set of all cuts with correct type-2 annotation is computed as a largest fixed point. *Phase 2* identifies the winning mcuts, i.e., mcuts where the strategy from type-1 places guarantees that the game continues with an infinite sequence of mcuts or reaches an mcut with only type-2 tokens. The set of mcuts is computed as a largest fixed point. Nested inside the largest fixed point computation is a least fixed point iteration that finds the predecessor mcuts, by first identifying all cuts from which the system players can force the game without further interaction with the environment into some mcut of the current approximation of the largest fixed point. Phase 2 also computes, as a least fixed point, the set of all cuts from which the system players can enforce a visit of such an mcut. The game is won by the system players iff the initial cut is in this set. *Phase 3* constructs a winning strategy if the game is won by the system players. The strategy first enforces the visit of a winning mcut according to the computation in Phase 2. From there, the strategy from type-1 places forces the game into new winning mcuts, and the strategy from type-2 places ensures, according to the computation in Phase 1, the safe continuation without any further interaction with the environment.

4 Experience with ADAM

ADAM is a Java-based implementation of the fixed point construction described in Section 3. For the BDD operations, ADAM uses BuDDy [13], a BDD library written in C. The size of the BDDs is reduced with various optimizations, including a compact representation of the place encodings, based on net invariants (which reduce the set of potential places for each token). We use the DOT [1] format as output for Graphviz for the visualization of the Petri games and the strategy graph of the 2-player game, as well as the Petri game strategies.

We have applied ADAM in several case studies from robotic control, workflow management, and other distributed applications. Table 1 shows representative results from several synthesis problems. The tool, more examples and their benchmarks are available online [8].

For each benchmark, the table shows the number $\#Tok$ of tokens, the number $\#Var$ of BDD variables used, and the numbers $\#P$ and $\#T$ of places and transitions, respectively, of the Petri game. We give the elapsed CPU *time* in s, and the used *memory* in GB for solving the problem. For the resulting solution, $\#P_s$ and $\#T_s$ are the number of places and transitions of the strategy, respectively. *Par* states the parameter size(s) of the example. The time and memory values are an average of 10 runs. The results were obtained on an Intel i7-2700K CPU with 3.50GHz and 32 GB RAM. The experiments refer to the following scalable benchmarks:

Table 1. Experimental results.

Ben.	*Par.*	$\#Tok$	$\#Var$	$\#P$	$\#T$	*time*	*memory*	$\#P_s$	$\#T_s$
CM	2/1	6	66	13	10	1.4	0.31	14	8
	2/2	7	96	18	16	1.3	0.29	-	-
			...			...		...	
	2/6	11	216	38	40	206.5	5.43	-	-
	3/1	8	92	18	15	1.3	0.29	26	12
	3/2	9	132	25	24	2.1	0.3	36	18
	3/3	10	172	32	33	3.3	0.38	-	-
	3/4	11	212	39	42	11.6	0.8	-	-
	3/5	12	252	46	51	180.9	5.43	-	-
	4/1	10	120	23	20	1.6	0.29	42	16
	4/2	11	172	32	32	3.9	0.38	55	24
	4/3	12	224	41	44	14.4	0.8	68	32
	4/4	13	276	50	56	155.3	4.27	-	-
	5/1	12	146	28	25	4.0	0.38	62	20
	5/2	13	208	39	40	24.3	0.8	78	30
	5/3	14	270	50	55	468.3	3.5	94	40
	6/1	14	172	33	30	19.6	0.8	86	24
	6/2	15	244	46	48	1042.2	2.51	105	36
SR	2/1	5	86	18	17	1.3	0.29	32	16
	2/2	6	116	24	26	1.6	0.29	-	-
	2/3	7	144	30	35	4.4	0.39	-	-
	2/4	8	174	36	44	42.7	0.8	-	-
	3/1	6	204	34	49	1155.6	10.05	79.7	45
JP	2	3	46	12	13	1.1	0.31	16	13
	3	4	76	18	23	1.8	0.31	34	28
			...			...		...	
	10	11	612	88	149	146.9	5.43	552	385
	11	12	762	102	175	434.8	16.62	706	484
DW	1	3	46	12	10	0.9	0.25	10	6
	2	4	72	19	16	1.8	0.30	22	15
			...			...		...	
	19	21	492	138	118	1411.8	15.93	1144	780
	20	22	516	145	124	1734.7	15.85	1264	861
DWs	1	3	36	11	6	0.8	0.31	8	3
	2	5	70	21	12	1.6	0.31	23	10
			...			...		...	
	18	37	588	181	108	1027.3	11.94	1351	666
	19	39	620	191	114	1451.9	15.99	1502	741

'-' means no winning strategy exists.

• CM: *Concurrent Machines* (see Fig. 1). The environment decides which of n machines is functioning. On these machines, k orders should be processed, each order by one machine. Different orders can be processed concurrently on different machines. No machine should be used twice for processing orders.
Parameters: n machines / k orders

• SR: *Self-reconfiguring Robots* [11]. Each piece of material needs to be processed by n different tools. There are n robots having all n tools to their disposal, of which only one tool is currently used. The environment may (repeatedly) destroy a tool on a robot R. Then the robots reconfigure themselves so that R uses another tool and the other robots adapt their usage of tools accordingly.
Parameters: n robots with n tools / k tools will be successively destroyed

• JP: *Job Processing*. The environment chooses a subset of n different processors and a job that requires handling by each processor in this subset in ascending order. *Parameter*: n processors

• DW: *Document Workflow*. The environment hands over a document to one of n clerks. The document then circulates among the clerks. Each clerk should endorse it or not, but wants to make the decision dependent on who has endorsed it already. Altogether they should reach a unanimous decision. In a simpler variant DWs, all clerks should endorse it. *Parameter*: n clerks

The benchmarks represent essential building blocks for modeling various manufacturing and workflow scenarios that can be analyzed automatically by synthesizing winning strategies with Adam.

References

1. AT&T, Bell-Labs: DOT file format for Graphviz – Graph visualization software. http://www.graphviz.org/
2. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Automatic hardware synthesis from specifications: A case study. In: Proc. of the Conf. on Design, Automation and Test in Europe (DATE). pp. 1188–1193 (2007)
3. Bloem, R.P., Gamauf, H.J., Hofferek, G., Könighofer, B., Könighofer, R.: Synthesizing robust systems with RATSY. In: Association, O.P. (ed.) SYNT 2012. vol. 84, pp. 47 – 53. Electronic Proceedings in Theoretical Computer Science (2012)
4. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.F.: Acacia+, a tool for LTL synthesis. In: Madhusudan, P., Seshia, S.A. (eds.) CAV. LNCS, vol. 7358, pp. 652–657. Springer (2012)
5. Büchi, J.R., Landweber, L.H.: Solving sequential conditions by finite-state strategies. Transactions of the American Mathematical Society 138 (1969)
6. Church, A.: Logic, arithmetic and automata. In: Proc. 1962 Intl. Congr. Math. pp. 23–25. Uppsala (1963)
7. Ehlers, R.: Unbeast: Symbolic bounded synthesis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS. LNCS, vol. 6605, pp. 272–275. Springer (2011)
8. Finkbeiner, B., Gieseking, M., Olderog, E.: ADAM – Analyzer of distributed asynchronous models. University of Oldenburg and Saarland University. http://www.uni-oldenburg.de/csd/adam (2014)
9. Finkbeiner, B., Olderog, E.: Petri games: Synthesis of distributed systems with causal memory. In: Peron, A., Piazza, C. (eds.) Proc. Fifth Intern. Symp. on Games, Automata, Logics and Formal Verification (GandALF). EPTCS, vol. 161, pp. 217–230 (2014), http://dx.doi.org/10.4204/EPTCS.161.19
10. Finkbeiner, B., Schewe, S.: Coordination logic. In: CSL. LNCS, vol. 6247, pp. 305–319. Springer (2010)
11. Güdemann, M., Ortmeier, F., Reif, W.: Formal modeling and verification of systems with self-x properties. In: Yang, L., Jin, H., Ma, J., Ungerer, T. (eds.) Autonomic and Trusted Computing. LNCS, vol. 4158, pp. 38–47. Springer (2006)
12. Kress-Gazit, H., Fainekos, G., Pappas, G.: Temporal-logic-based reactive mission and motion planning. Robotics, IEEE Transactions on 25(6), 1370–1381 (Dec 2009)
13. Lind-Nielsen, J.: BuDDy – Binary decision diagram package. IT-University of Copenhagen. http://sourceforge.net/projects/buddy/
14. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proc. POPL'89. pp. 179–190. ACM Press, New York, NY, USA (1989)
15. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: Proc. FOCS'90. pp. 746–757 (1990)
16. Rabin, M.O.: Automata on Infinite Objects and Church's Problem, Regional Conference Series in Mathematics, vol. 13. Amer. Math. Soc. (1972)
17. Reisig, W.: Elements of Distributed Algorithms – Modeling and Analysis with Petri Nets. Springer (1998)