Deadlock detection in networks of automata communicating via flags

Andrei Karatkevich

Abstract: A range of digital systems can be represented as the state machine networks in which FSMs communicate with the help of flip-flops. The article presents a methodology of detecting possible deadlocks in such networks. The methodology is illustrated by applying it to a project of a pipeline processor.

Keywords: State machines, system design, verification, graph-schemes, pipeline.

1. INTRODUCTION

Finite state machine (FSM), or finite automaton, is a model widely used in digital system design. Modeling of complex systems, however, often requires using more than one state machine. Several automata can be composed into a network. For such networks, different ways of communication between the automata are used.

Automata can be connected directly (that means that the input of one automaton can be connected to the output of another one). But it is inconvenient to consider inputs and outputs of the automata in such networks as single symbols, as it is done in classical automata theory. More flexible and popular approach, allowing to represent easily Mealy and Moore (and mixed) automata with multiple outputs, uses the concept of broadcast events. It is applied in various models of concurrent finite state machines [2,5]. This fits perfectly well for the systems of logical control, but may be not enough for more complex cases, as modeling the structure of processors. Then the flags (variables) may be used as one of the ways of communication between automata [6]. Difference between flags and events is that an event has a short life (occurs only during an instant of time, unless it is not generated again and again by an active state, as "do" events in Statecharts [5]), and a flag keeps a value which is assigned to it by an action until another action sets another value.

In many models more complex dataflow domains are used [1,6]. But even for complex cases analysis of simplified models can provide some important information for verification. Detection of deadlocks (the situations in which two or more automata are waiting for some action of each other, so all of them are blocked) is one of the most important verification tasks. Deadlock detection in the FSM networks with communication via events is considered in [3]. Here we consider the same task for the FSM networks with communication via binary flags. A project of a pipeline processor is used for an example.

2. AN EXAMPLE: A PIPELINE PROCESSOR

As an example we use a project of a basic pipeline RISC processor with high performance, designed as a diploma project by Ilan Kutzman and Alex Raitzin at Bar Ilan University, Israel, supervised by Prof. Samary Baranov [4]. Figs 1-4 show the Algorithmic State Machines (ASMs) [1] with generalized operators (not expanded here), specifying behavior of four pipeline stages. The fifth stage (Write stage) is not shown here, as less important for deadlock detection. This example has more than 10^9 states.

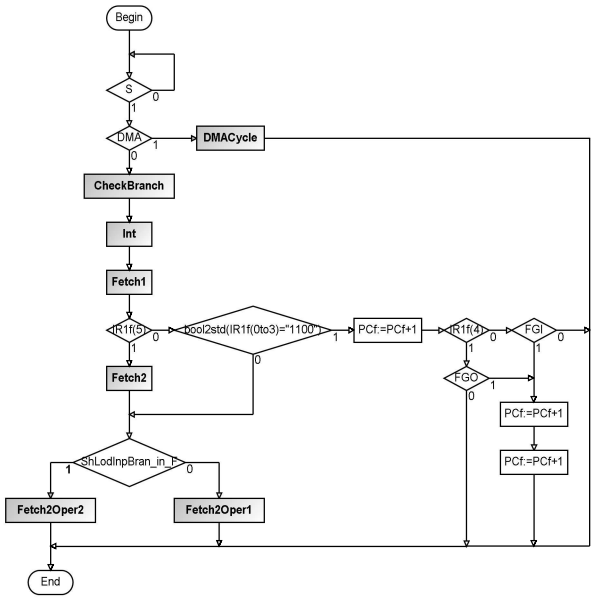

Fig. 1. Fetch stage

3. ANALYSIS PROCESS

3.1. CONSTRUCTING OF THE MODEL

To create a model with reduced complexity (consisting of FSMs, not ASMs) an abstraction can be used. At this step we are going to abandon the computation of variables and to consider only the single Boolean variables to which concrete values are assigned by concrete operators (processing steps). In the given system there are following such variables (flags): *Fop1B*, *Fop2B*, *FImB*, *BranChk*, *BIP*, *IEN*, *FGO* and *FGI*.

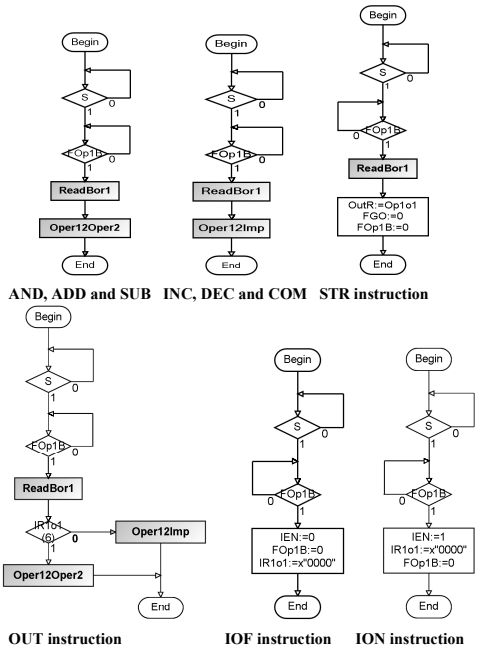

Fig. 2. Operands1 stage

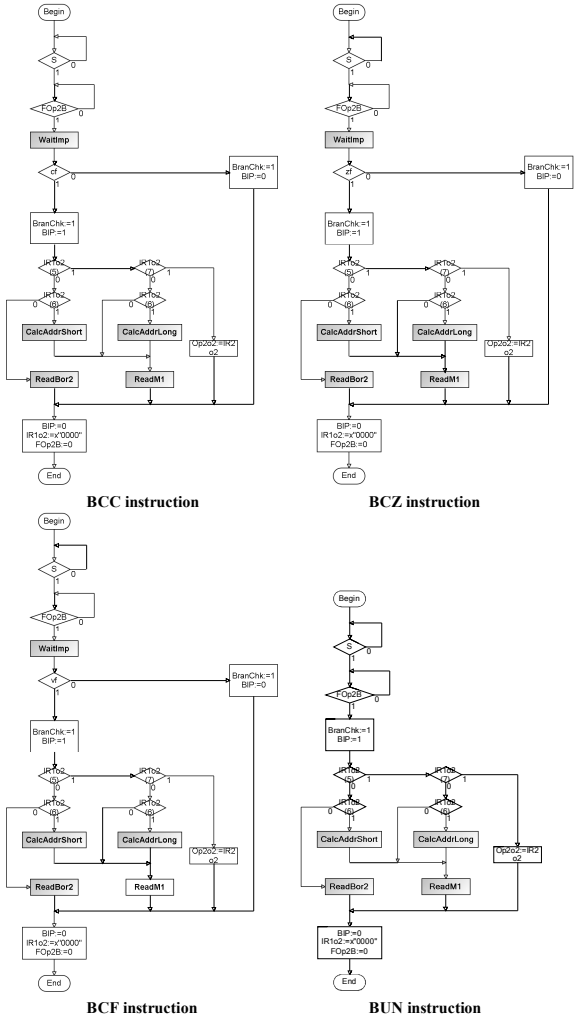

Fig. 3. Operands2 stage (only ASMs for the branch instructions are shown)

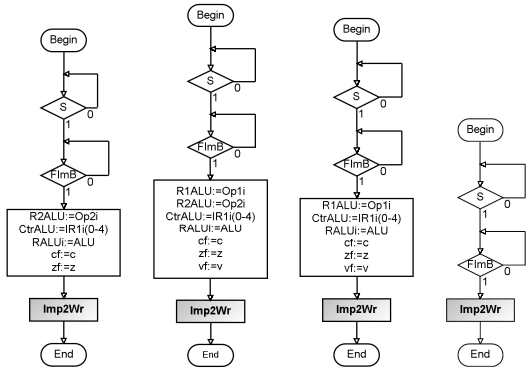

Fig. 4. Implement stage

3.2. BUILDING AND ANALYSIS OF WFG

Now we are going to construct a Wait-for graph (WFG) [8] – a kind of dependency graph, in which nodes represent processes, and arcs represent blockages. In our graph nodes will correspond to the automata. An arc from automaton A_i to A_j means that A_j has a state in which it is waiting for certain value of flag x, and A_i has a state in which it assigns this value to x. The arc is labeled by x ($\overline{x}$), if A_j waits for value 1 (0). The difference between this graph and the WFG used for deadlock detection in operating systems is that it is constructed not for a current situation but for all possible dependencies in given system.

Flags *FGO* and *FGI* cannot cause a deadlock, because no value of them can block any automaton. Let us construct the WFG representing dependencies via the rest of flags. It is shown in fig. 5.

If there is a deadlock, there is a corresponding cycle in the WFG. But existing of a cycle is not a sufficient condition of a deadlock. For example, there is a cycle between Write and Impl nodes, but it does not indicate any deadlock, because it would imply that flag *FwrB* has value 0 and 1 at the same time. A cycle in the WFG means cyclic waiting, if *for any literal labeling an arc in it, its complement does not appear in the cycle*. Then a deadlock is possible, if the flags have values turning the literals appearing in the cycle to 1, and the automata are in the corresponding states. We call such cycle a *deadlock cycle*.

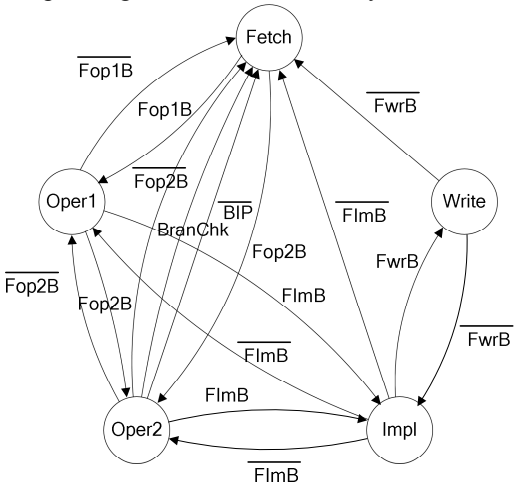

Fig. 5. Wait-for graph for the pipeline processor

All cycles in an oriented graph can be generated by the algorithm described in [7]. As far as number of cycles in a graph depends exponentially on its size, it is reasonable to simplify the graph by removing the nodes which cannot belong to the cycles which are interesting for us. Such simplification can be performed step by step (removing of some nodes can allow removing the following ones). Of course any node which indegree or outdegree is 0 can be removed, because it cannot belong to any cycle. Also, if every incoming arc of a node is labeled by a literal and every outgoing arc is labeled by its complement, this node cannot belong to a deadlock cycle. That is the case for node Write: its incoming arc is labeled by *FwrB*, and both outgoing arcs are labeled by $\overline{FwrB}$. But after removing node Write from the graph, node Impl turns to be in similar situation: now every its incoming arc is labeled by *FimB*, and every outgoing – by $\overline{FimB}$. That means that Impl also cannot be deadlocked.

Further simplification can be performed by removing some arcs. If every incoming arc of a node is labeled by the same literal, then no outgoing arc labeled by its complement can belong to a deadlock cycle. Every such outgoing arc can be removed. In our example, both incoming arcs of Oper2 have labels *Fop2B*. Then two outgoing arcs labeled by $\overline{Fop2B}$ can be removed. It implies the situation in which Oper1 has only one incoming arc labeled with *Fop1B*, which means that the arc from Oper1 to Fetch can be removed. The simplified graph is shown in fig. 6.

This graph, as it is easy to see now, has 4 deadlock cycles. But one may note that some of them in a sense cover others. For any cycle Fetch→Oper1→Oper2→Fetch there is a cycle Fetch→Oper2→Fetch, labeled with the same literals except *Fop1B*. That means that stages Fetch, Operands1 and Operands2 can be mutually blocked, only if Fetch and Operands2 are mutually blocked. So it is enough to check possibility of two deadlocks: corresponding to cycles $\text{Fetch} \xrightarrow{Fop2B} \text{Oper2} \xrightarrow{BranChk} \text{Fetch}$ and $\text{Fetch} \xrightarrow{Fop2B} \text{Oper2} \xrightarrow{\overline{BIP}} \text{Fetch}$. Other deadlocks are not possible in this system.

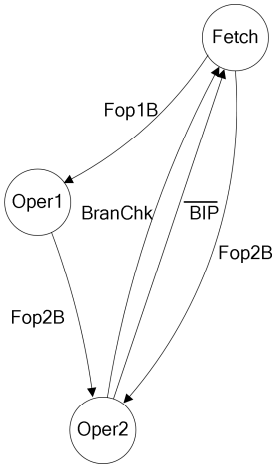

Fig. 6. Reduced Wait-for graph

3.3. CHECKING REACHABILITY OF THE DEADLOCKS

Existing of a deadlock cycle in the WFG means, that there exists a combination of states of automata, in which they are deadlocked, but it does not always mean that such combination is reachable from the initial state of the system. Now, similarly as it was proposed for deadlock analysis in Statecharts [3], after detection of possibility of deadlocks by means of structural analysis, it is necessary to check reachability of possible deadlocks.

To simplify this check, let us use the reduced flowcharts representing only the behavior related to reading or writing the flags appearing in the reduced WFG (*Fop1B*, *Fop2B*, *BranChk*, *BIP*). The fragments of automata which have nothing to do with the flags can be reduced. Identical fragments can be fused. Such reduced representation of Fetch stage is presented in fig. 7.

As one can see, at this level we represent the system as a nondeterministic state machine. Here big rounds mean fragments of the automaton graph which neither read nor write the flags we are interested in. They can end up in more than one state, depending on the inputs we abstract here. That's why they may have multiple outputs. Small rounds mean the states in which the automaton waits for a value of a flag. Only in such states it can be deadlocked. As follows from the WFG, the only states in which the Fetch stage can be deadlocked are *a* and *d*.

Reduced representations of Operands1 and Operands2 are shown in fig. 8. There is only one possibility for stage Operands2 to be deadlocked: if it is blocked in state *a*, i.e. at the very beginning. Analysis of the state space of the reduced flowcharts demonstrates that, assuming that initially all flags have value 0, a deadlock corresponding to the cycle $\text{Fetch} \xrightarrow{Fop2B} \text{Oper2} \xrightarrow{\overline{BIP}} \text{Fetch}$ is not possible. Indeed, Fetch can be blocked in state *d* (waiting for *BIP*=0) only when *BIP*=1, *BIP* can be set to 1 only by Operands2, and Operands2 always sets *BIP* to 0 afterwards and cannot be blocked between these two operators.

The last potential deadlock we have to consider is when Fetch is in state *a* (waiting for *BranChk*=1) and Operands2 is in its state *a* (waiting for *Fop2B*=1). At the level of the reduced flowcharts it seems to be possible: for example, if Fetch immediately enters state *a*, both Operands stages cannot leave their initial states. Such analysis is not enough in our case, because transitions between states in the stages of pipeline processor are not independent. So we have to turn to the detailed description of the pipeline processor and to check whether this deadlock is indeed possible.

Analysis at the level of detailed descriptions shows that, according to the generalized operator CheckBranch (which details are not shown here), Fetch stage can enter the state of waiting for *BranChk*=1 only when there is branch instruction in Operands2 stage. It means that it cannot happen from the very beginning of system functioning, because each stage maintains its own copy of instruction, and the copy for Operands2 can be written only by operator Fetch2Oper2 (Fetch) or Oper12Oper2 (Operands1). First two stages can function avoiding executing these operators (Operands2 stage is skipped for

some instructions – see figs 1 and 2). However, as far as Operands2 clears IR1O2 register, which contains this stage's copy of instruction (see fig. 3), and flag *Fop2B* is set to 1 always *together* with writing the instruction to IR1O2, the situation in which both Fetch and Operands2 stages are waiting for each other is not possible. So, we have shown that the system is deadlock-free.

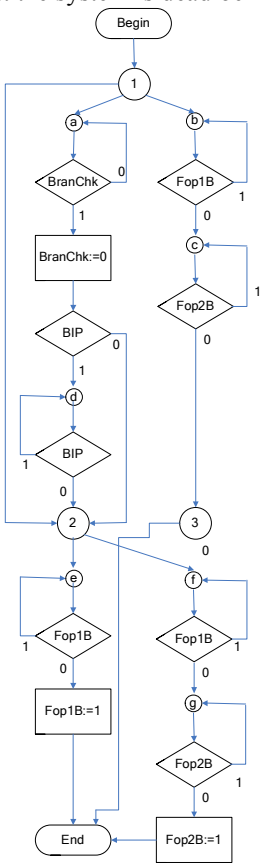

Fig. 7. Reduced flowchart of the Fetch stage

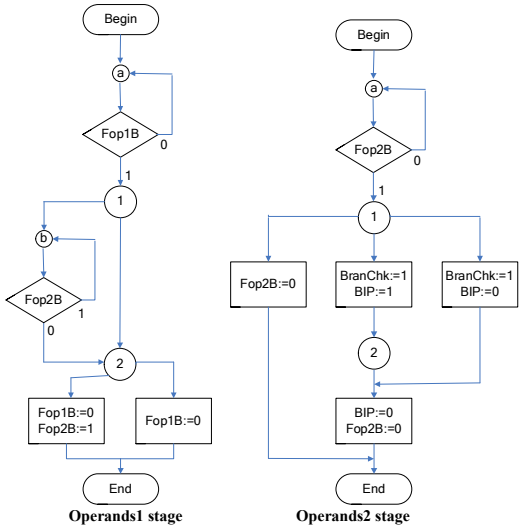

Fig. 8. Reduced flowchart of the Operands1 and Operands2 stages

Note that in the initial design described in [4] there was no zeroing of IR1O2 after every branch, and the deadlock corresponding to the cycle Fetch $\xrightarrow{Fop2B}$ Oper2 $\xrightarrow{BranChk}$ Fetch was possible. It was discovered during simulation of the design. But simulation never ensures detecting all possible bugs (in this case deadlocks), and formal analysis like proposed in this paper does ensure.

4. Summary

The proposed method consists of the following steps.

1. Constructing an FSM network model of the analyzed system.
2. Constructing a Wait-for graph describing dependencies between the automata.
3. Reducing the WFG by removing nodes and arcs which cannot be visited by any deadlock cycle.
4. Generating all deadlock cycles in the reduced WFG. If there are no such cycles, the system is deadlock-free.
5. Reducing the fragments of the automata in the network which do not deal with the selected flags.
6. Checking whether the potential deadlocks detected in step 4 are reachable in the reduced network. If not, the system is deadlock-free.
7. Checking whether the potential deadlocks, not excluded in step 6, are reachable in the analyzed system.

The method can be used for formal verification of a wide range of systems with parallel logical control units. It allows detecting, without much computational effort and without exploration of most of the state space, deadlocks possible in a system or ensuring that it is deadlock-free.

Acknowledgement

I am grateful to Samary Baranov for inspiration, fruitful discussions and for the example described in [4].

References

[1] Baranov S., *Logic and System Design of Digital Systems*. Tallinn, TGU, 2008.

[2] Harel D., "Statecharts: a visual formalism for complex systems", *Science of Computer Programming*, nr 8, pp. 231-274, 1987.

[3] Karatkevich A., „Deadlock analysis in Statecharts", *Proceedings of the Forum on Specification and Design Languages – FDL'03*, Frankfurt, Germany, 2003, pp. 414-424.

[4] Kutzman I., Raitzin A., *Design Of Fast Pipelined Processor With Complex Addressing Mode*. B.Sc. diploma project, Bar Ilan University, 2007.

[5] Łabiak G., „From UML Statecharts to FPGA - the HiCoS approach", *Proceedings of the Forum on Specification and Design Languages – FDL'03*, Frankfurt, Germany, 2003, pp. 354-363.

[6] Lee E.A., "Finite State Machines and Modal Models in Ptolemy II", technical report, EECS Department, Univ. of California, Berkeley, USA, 2009.

[7] Reingold E.M., Nievergelt J., Deo N., *Combinatorial Algorithms. Theory and Practice*. Prentice-Hall, Inc., 1977.

[8] Silberschatz A., Galvin P., Gagne G., *Operating System Concepts*, John Wiley & Sons, Inc., 2003.