(Self-)reconfigurable Finite State Machines: Theory and Implementation

Markus Köster, Jürgen Teich
University of Paderborn
Computer Engineering Laboratory
Paderborn, Germany
email: {koester, teich}@date.upb.de

Abstract

In this paper, we introduce the concept of (self-)reconfigurable finite state machines *as a formal model to describe state-machines implemented in hardware that may be reconfigured during operation. By the advent of reconfigurable logic devices such as FPGAs, this model may become important to characterize and implement (self-)reconfigurable hardware. An FSM is called* (self-)reconfigurable *if reconfiguration of either output function or transition function is initiated by the FSM itself and not based on external reconfiguration events. We propose an efficient hardware realisation and give algorithmic solutions and bounds for the reconfiguration overhead of migrating a given FSM specification into a new target FSM.*

1 Introduction

With the advent of reconfigurable and partially reconfigurable hardware such as *field-programmable gate arrays* (FPGAs), it is possible to change the functionality and wiring of hardware over time.

Most of these circuits are SRAM-based, and reconfiguration takes place by reconfiguring the actually configuration bit-stream for logic function and switches over rows or columns of an array of reconfigurable logic blocks (CLBs).

Using today's submicron technology, chips with more than 10 million gates are on the market, some with, some without partial reconfigurability.

With reconfiguration times in the order of milliseconds, the time to switch a reconfiguration is thus much smaller than compile-times for mapping algorithms or state machines to reconfiguration bit-streams.

Recent approaches such as [1, 2, 3, 6, 10, 9, 12, 14] describe applications and algorithms exploiting reconfiguration capabilities of dynamically reconfigurable FPGAs such as [15]. In order not to take too much time, presynthesized bit-streams are generated at compile-time and only these configuration streams are overwritten or simply swapped [4] at run-time using techniques such as multi-context FPGAs [8, 13].

The idea of reconfigurable hardware may, however, be spun further: In the software world, the first generations of microprocessors made use of the ingenious idea of self-reconfiguring program code stepwise in order to execute larger or more complex programs that would otherwise not fit into the existing memory.

Here, we intend to make use of this principle in the context of hardware: *Circuits that reconfigure themselves gradually!*

The pure idea alone is obvious and not very new, however, so far there exist 1) no high-level models such as *(self-)reconfigurable state machines* presented here for characterizing (self-)reconfigurable hardware, and 2) there is no obvious way how to implement enhanced (self-)reconfigurable principles that not just swap complete reconfiguration contents. If, for example a circuit configured in a FPGA has to compute new bit-level configurations which subsequently replace the configuration itself, it will be very tedious to describe the hardware functionality. Indeed, the bit streams to be generated would need to know where on the chip the logic would reside, needing all the information of the placement and routing tools.

Here, we therefore introduce the high-level model of

- *(self-)reconfigurable finite state machines* as a formal model of finite state machines with the ability to change output and transition functions or both gradually during operation, and derive
- *an FPGA-based hardware implementation* of this model as well as
- *algorithms* and *bounds* for the overhead of automatic reconfiguration of a given FSM.

Real application domains that may profit from the con-

cept of (self-)reconfigurable FSMs are areas of time-varying control, e.g., network protocol applications that require packet-dependent processing.

1.1 Overview

In Section 2, we provide basic definitions introducing *(self-)reconfigurable FSMs*. An implementation of this model on FPGA-based architectures is explained in Section 3. Finally, in Section 4, we show under which condition a given state machine M may be reconfigured into a state machine M' and we present algorithms and bounds for the overhead of reconfiguration of a given FSM M into a FSM M'.

2 Definitions

2.1 Finite State Machines

Definition 2.1 (Incompletely specified non-deterministic Mealy-FSM) *An* incompletely specified non-deterministic Mealy Finite State Machine *(FSM) is a 6-tuple* (I, O, S, S_0, F, G)*:*

- S *is a finite set of* internal states *(often simply called states) of the FSM.*
- I *is a finite set of* input states *of the FSM, which either are symbolic or are represented as a binary vector of values of its* input signals.
- O *is a finite set of* output states *of the FSM, which are either symbolic or are represented as a binary vector of values of its* output signals.
- $F(i, s)$ *is a relation from the (input state, present state) pairs, also called* total states*, to the next states (i.e.,* $F \subseteq I \times S \times S$*).*
- $G(i, s)$ *is a relation from the (input state, present state) pairs to the output states (i.e.,* $G \subseteq I \times S \times O$*).*
- $S_0 \subseteq S$ *is a set of* initial *(or reset)* states.

An FSM is *deterministic* if S_0 is a singleton set and both F and G are functions. We will call an incompletely specified deterministic FSM simply *an FSM*.

A deterministic FSM is *completely specified* if both F and G are total functions (i.e., they are defined for all elements of their domains). This is the class of FSMs we will consider throughout this work. Finally, a deterministic FSM is a *Moore-FSM* if the edges directed into a state s have a single output label. Hence, for Moore-FSMs the output state is associated with the internal state rather than with the total state.

Fig. 1 shows a schematic diagram of a Mealy-FSM.

An FSM can also be represented by a directed graph, called the *state transition graph*, where

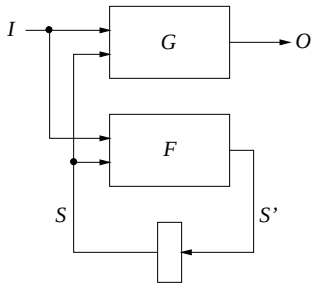

Figure 1. Schematic diagram of a Mealy-FSM

- each vertex is associated with an internal state, and
- each edge (also called a *transition*) is labeled with an input state/output state pair, and is directed from the *present state* vertex to the *next state* vertex.

Finally, a *stable total state* of an FSM is a pair (i, s) such that $F(i, s) = \{s\}$, i.e., a self-loop in the graph representation.

2.2 (Self-)reconfigurable Finite State Machines

We extend the definition of finite state machines in Definition 2.1 to the concept of *(self-)reconfigurable finite state machines*.

Obviously, it is possible to allow an FSM to change its output function G and its transition function F over time. We call this extension a *reconfigurable FSM*. In case the reconfiguration is initiated by the system itself (autonomously), then we speak of *self-reconfiguration*.

In the following, we introduce a formal definition of reconfigurable finite state machines. Interestingly, it might be possible to change G and F not only autonomously, but possibly also in dependence of external events such as *reconfiguration events*. In order to handle both possibilities accordingly, we define *reconfigurable FSMs* as follows.

2.2.1 Reconfigurable FSMs

Definition 2.2 (Reconfigurable FSM) *A* reconfigurable FSM *is a 10-tuple* $(I, O, S, S_0, F, G, H_f, H_g, H_i, R)$ *in which*

- (I, O, S, S_0, F, G) *describes an FSM according to Definition 2.1, and*
- R *is a finite set of* reconfiguration states *of the FSM which either are symbolic or are represented as a binary vector of values of its* reconfiguration input signals.
- *The* transition reconfigurator function $H_f(r)$*,* $r \in R$*, is a mapping from the reconfiguration state to the total state* $F(i', s)$*,* $i' \in I$*,* $s \in S$*, i.e.,* $F(i', s)$ *may*

be reconfigured by an update as follows: $F(i', s) := H_f(r)$.

- *The* output reconfigurator function $H_g(r)$, $r \in R$, *is a mapping from the reconfiguration state to the output state* $G(i', s)$, $i' \in I$, $s \in S$, *i.e.,* $G(i', s)$ *may be reconfigured by an update as follows:* $G(i', s) := H_g(r)$.
- $H_i(i, r)$, $i' \in I$, $r \in R$, *is a mapping from the (input state, reconfiguration state) tuple to the internal input state* $i' \in I$.

The concept is shown in the schematic of Figure 2. In our model, the transition function F may be updated by H_f and the output function G may be updated by H_g. H_f and H_g depend on the reconfiguration state, whereas H_i depends on the input state I and the reconfiguration state R. The function H_i is defined such that during normal operation of the FSM, $I' = I$, and during the reconfiguration process, I' is depending on r only. Physical aspects of when and how the update of the output function and the transition function may take place will be explained in Section 3.

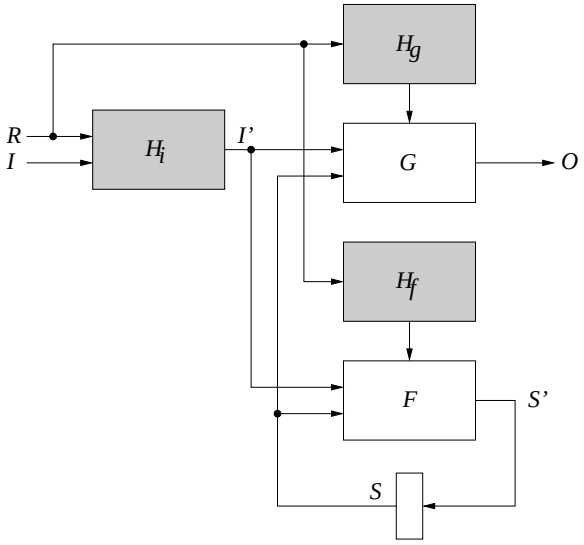

Figure 2. Schematic of a reconfigurable FSM

Example 2.1 *The following VHDL-specification [11] describes the behavior of a finite state machine (Mealy-FSM)* $M = (I, O, S, S_0, F, G)$ *with* $I = \{i\}, i \in \{0, 1\}, O = \{o\}, o \in \{0, 1\}, S = \{S_0, S_1\}$, *that reads an endless bitstream and outputs the value* $o = 1$ *in case two or more successive ones have been detected in the bitstream until the first time, a zero occurs again at its input.*

```
ENTITY rec IS
  PORT (in, clk: IN BIT; out: OUT BIT);
END rec;
ARCHITECTURE behavior OF rec IS
  TYPE state_type IS (S0, S1);
  SIGNAL state: state_type := S0;
  PROCESS
  BEGIN
    WAIT UNTIL clk'EVENT AND clk = '1';
    IF (in = '1') THEN
      CASE state IS
        WHEN => S0
          state <= S1;
          out <= '0';
        WHEN => S1
          state <= S1;
          out <= '1';
      END CASE;
    ELSE
      state <= S0;
      out <= '0';
    END IF;
  END PROCESS;
END behavior;
```

In Fig. 3, the according state-transition diagram is shown as well as a hardware implementation. Now, assume we want to reconfigure the state-machine such that the machine now should count the number of zeros in the bitstream instead of the ones. Therefore, let us assume the machine is in state S_0 *and* $R = \{r_0, r_1, r_2, r_3, r_4\}$ *is the set of reconfiguration states. Then a reconfiguration sequence taking four clock cycles is shown in Fig. 4 where 1) shows the given state machine and 4) shows the reconfigured state machine, see also Tab. 1.*

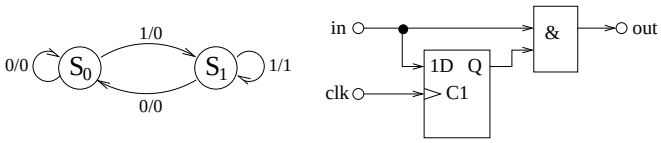

Figure 3. Example of a simple state-transition graph and its implementation

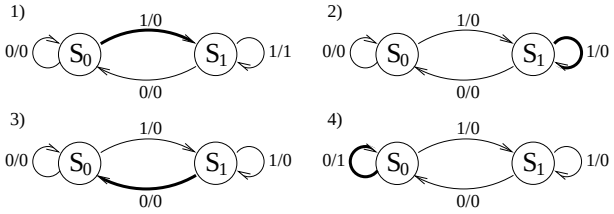

Figure 4. Transitions taken during reconfiguration of the FSM in Fig. 3

r	$i' = H_i(i, r)$	$H_f(r)$	$H_g(r)$
r_1	1	S_1	0
r_2	1	S_1	0
r_3	0	S_0	0
r_4	0	S_0	1
r_0	i	–	–

Table 1. Reconfiguration sequence for the example introduced in Fig. 3

In state S_0, we have $r = r_1$ and define $i' = H_i(i, r_1) = 1$, so that a transition to state S_1 under input $i' = 1$ is forced. This represents the first reconfiguration cycle. Simultaneously, we reconfigure the output $G(1, S_0)$ such that $G(1, S_0) := H_g(r_1) = 0$. In S_1, with $r = r_2$, we also let $i' = H_i(i, r_2) = 1$, stay in S_1 ($H_f(r_2) = S_1$) and change the output of this transition to $G(1, S_1) := H_g(r_2) = 0$. After two more cycles, the machine is reconfigured.

From that cycle on, we let $r = r_0$ (normal operation) which finishes the reconfiguration, so F and G will keep their values.

What can be seen from the above example is that the principle of reconfigurable state machines is realized by taking a sequence of state transitions where during each transition, the output and transition function are updated until the machine is reconfigured.

In Def. 2.2, we assumed that the reconfiguration state $r \in R$ is input from the external world. An FSM may therefore be called self-reconfigurable, if the reconfiguration sequences are generated as part of the system, e.g. in dependence of a reached state or other conditions.

3 Implementation

In the previous section, we introduced the definition of (self-)reconfigurable state machines. Here, we will present concepts for implementing these principles in hardware. Figure 5 shows the schematic of our hardware implementation of a (self-)reconfigurable FSM. In this implementation the block **Reconfigurator** is realizing the functions H_i, H_f, H_g according to Def. 2.2, and generating two additional signals *rst* and *rst-state* that are explained later. **F-RAM** and **G-RAM** are memory devices which implement the reconfigurable transition and output functions $F(i', s), G(i', s), i' \in I, s \in S$ according to Def. 2.2.

The state register **ST-REG** outputs the current state s. It is updated on every rising clock edge by the next state s'.

The reset multiplexer **RST-MUX** is used to force the next state to the reset state ($s' = S_0$) in dependence of the reset signal *rst*. Hence, no matter what current state s the machine is in, it is always possible to enter the reset state S_0 on the next rising clock edge.

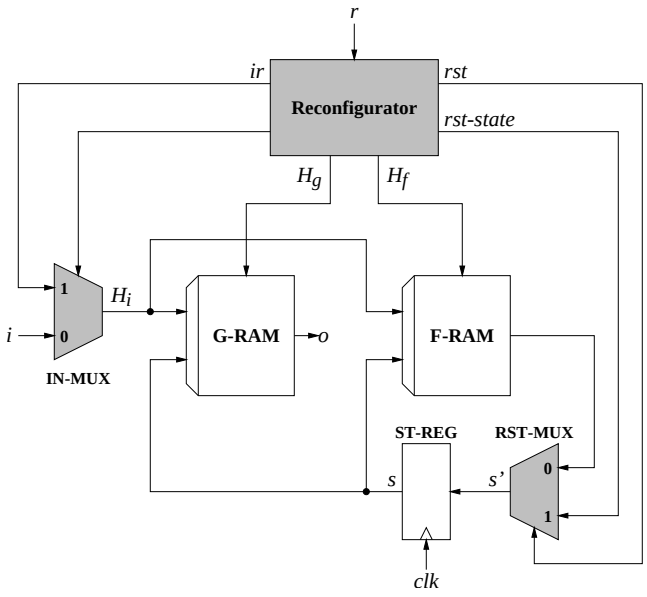

Figure 5. Implementation of a reconfigurable FSM

In case the FSM works in *normal mode*, the multiplexer **IN-MUX** selects the external input i ($H_i(i, r) = i$). So, the address of the memory blocks **F-RAM** and **G-RAM** depend on the external input i and the current state s. **G-RAM** generates the output o, whereas **F-RAM** generates the next state s' if the reset state is not forced.

In case the FSM is in *reconfiguration mode*, the multiplexer **IN-MUX** selects the signal ir generated by the block **Reconfigurator** ($H_i(i, r) = ir$). Now, the address of the memory blocks **F-RAM** and **G-RAM** depend on ir and the current state s, so the external input i does not affect the machine. Depending on the reconfiguration state r, the **Reconfigurator** generates new values for ir, H_f and H_g to update the corresponding memory locations in **F-RAM** and **G-RAM**. Thereby, the FSM is changed.

This implementation has been realized in FPGA-technology[1], where the block **Reconfigurator** is realized using logic blocks whereas **F-RAM** and **G-RAM** have been realized in embedded memory blocks.

Note the advantage of our definition and implementation of (self-)reconfigurable state machines: The reconfiguration function is independent on the placement and routing of the hardware on the FPGA. This is in contrast to approaches where the placed circuit has to generate reconfiguration sequences in the form of technology-dependent bitstreams.

[1] XILINX Virtex XCV300.

4 Algorithms for Reconfiguration

The above reconfiguration mechanism gradually changes a state machine. In each reconfiguration cycle, the machine may change its structure stepwise. However, the change of a state machine to the next may only affect one transition/output pair at most. Only after a certain number of state transitions will the output function, respectively transition function be such that the complete reconfiguration is finished.

Arising with the above principle therefore comes the general question of migrating a given FSM M into a new FSM M' by reconfiguration:

- Given a state machine M and a state machine M' . Under what conditions may M mutate into M' by (self-)reconfiguration at run-time?
- If M may mutate into M', then what is the minimal reconfiguration effort in terms of the number of state transitions necessary?

We will answer these questions next.

4.1 Feasibility of (Self-)reconfiguration

Without loss of generality, we assume a fully specified, deterministic FSM $M = (I, O, S, S_0, F, G)$ of type Mealy is given in the following[2].

4.1.1 Reconfiguration of transition function and output function

In order to be able to solve the general questions of feasibility and optimality of reconfiguration of output and transition functions, a combination thereof, and including also the case of partial reconfiguration, we state the following mostly general problem formulation:

Definition 4.1 (Feasible/Optimal (Self-)reconfiguration) *Given a finite state machine $M = (I, O, S, S_0, F, G)$ and a finite state machine $M' = (I', O', S', S'_0, F', G')$ and let S_{super} denote a super set including S and S', i.e., $S \subseteq S_{\text{super}}$ and $S' \subseteq S_{\text{super}}$. Similarly, let I_{super} denote a super set of input states including I and I', i.e., $I \subseteq I_{\text{super}}$ and $I' \subseteq I_{\text{super}}$, and finally, let O_{super} denote a super set of output states including O and O', i.e., $O \subseteq O_{\text{super}}$ and $O' \subseteq O_{\text{super}}$.*

The problem of feasible (self-)reconfiguration *denotes the decision problem whether M may be transformed into M' by a finite sequence of (self-)reconfiguration steps starting in $S_0 \in S$ and ending in $S'_0 \in S'$. The problem of* optimal (self-)reconfiguration *is to determine the minimum number of transitions in order to (self-)reconfigure M to M', initially starting in S_0 and ending in S'_0.*

Theorem 4.1 (Feasible (Self-)reconfiguration) *Given an FSM M and an FSM M' according to Definition 4.1. It is always possible to specify a finite sequence of reconfiguration transitions in order to transform M into M'.*

A constructive proof of this theorem is provided later in Section 4.4.

4.2 Reconfiguration Program

Consider a given state machine M we want to reconfigure into a target state machine M'. First we need to know what transitions have to be reconfigured when migrating M to M'.

Definition 4.2 (Delta Transition) *Given a given finite state machine $M = (I, O, S, S_0, F, G)$ and a target finite state machine $M' = (I', O', S', S'_0, F', G')$. Let $T' = \{(i, s_x, s_y, o) : i \in I',\ s_x \in S',\ s_y = F'(i, s_x),\ o = G'(i, s_x)\}$ denote the total set of transitions of M'. Then a transition $t_d = (i, s_x, s_y, o) \in T'$ is called* delta transition *and needs to be reconfigured in order to mutate M into M', if at least one of the following conditions hold:*

- *$i \notin I$, or $s_x \notin S$, or $s_y \notin S$, or $o \notin O$, or*
- *$s_y \neq F(i, s_x) \wedge i \in (I \cap I') \wedge s_x \in (S \cap S')$, or*
- *$o \neq G(i, s_x) \wedge i \in (I \cap I') \wedge s_x \in (S \cap S')$.*

Let $T_d \subseteq T'$ be the set of all delta transitions.

Example 4.1 *Consider the finite state machine $M = (I, O, S, S_0, F, G)$ with $I = \{i\}, i \in \{0, 1\}, O = \{o\}, o \in \{0, 1\}, S = \{S_0, S_1, S_2\}$ and the finite state machine $M' = (I', O', S', S'_0, F', G')$ with $I' = I, O' = O, S' = \{S_0, S_1, S_2, S_3\}$ shown in Fig. 6. If we want*

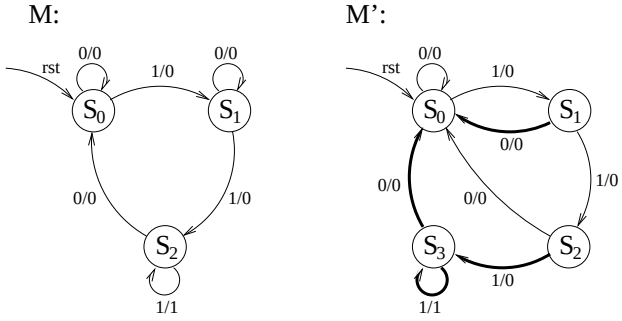

Figure 6. Given state machine M and reconfigured state machine M'

to reconfigure M into M', the delta transitions are $T_d = \{(0, S_1, S_0, 0), (1, S_2, S_3, 0), (1, S_3, S_3, 1), (0, S_3, S_0, 0)\}$ (highlighted bold in Fig. 6).

[2]Note that a Moore-FSM is just a special case where the output function is dependent on the state only.

Since we are not able to reconfigure more than one transition at a time, we need to generate a sequence of successive transitions to reach and traverse every delta transition of M'. In the following, we call this sequence of transitions a *reconfiguration program* $Z = (z_0, z_1, \ldots, z_n), z_k \in T', 0 \leq k \leq n$. Every reconfiguration program has an *initial state* which represents the last state the machine M is in before it is reconfigured, and a *terminal state* which represents the state of M' after finishing reconfiguration. From a reconfiguration program, a corresponding reconfiguration sequence according to Table 1 can be easily derived as follows: The input condition of each transition describes the value of the function H_i. The new target state of a transition describes the value of the function H_f, and the new output state describes the value of the function H_g.

4.3 Temporary Transitions

During a reconfiguration process, it is sometimes necessary to traverse several transitions that either are no delta transitions or have been reconfigured already in order to reach a state with an outgoing delta transition. To shorten the path to this delta transition, we may obviously reconfigure any transition leaving the actual state to reach directly the state from which the delta transition emanates. We call such a transition a *temporary transition*[3]. Reconfiguring an already configured transition causes the original transition to change into a delta transition. One example of the usefulness of temporary transitions during reconfiguration is given below.

Example 4.2 *Assume we want to reconfigure the state machine* $M = (I, O, S, S_0, F, G)$ *with* $I = \{i\}, i \in \{0,1\}, O = \{o\}, o \in \{0,1\}, S = \{S_0, S_1, S_2, S_3\}$ *into state machine* $M' = (I', O', S', S'_0, F', G')$ *with* $I' = I, O' = O, S' = S$ *shown in Fig. 7. Let us assume* M *is in state* S_0 *and the delta transition* $(0, S_3, S_0, 0)$ *should be reconfigured next. Using the given transitions of state machine* M*, the shortest reconfiguration program to configure the delta transition is:* Z=$((1, S_0, S_1, 0), (1, S_1, S_2, 0), (1, S_2, S_3, 0), (0, S_3, S_0, 0))$. *We need four cycles ending in state* S_0.

Now we generate a different reconfiguration program. We configure the transition $(0, S_0, S_0, 0)$ *to a temporary transition* $(0, S_0, S_3, 0)$ *(see Fig. 8) and shorten the path to the delta transition* $(0, S_3, S_0, 0)$. *Thereby, the transition* $(0, S_0, S_0, 0)$ *is changed to a new delta transition as shown in Figure 8. Apparently, we have introduced another delta transition to be reconfigured. However, including the step needed to configure it back to the original transition* $(0, S_0, S_0, 0)$*, we now only need three cycles for the reconfiguration ending in state* S_0. *The corresponding reconfiguration program is* Z=$((0, S_0, S_3, 0), (0, S_3, S_0, 0), (0, S_0, S_0, 0))$.

[3]We call these transitions temporary as they only exist temporarily during the reconfiguration process.

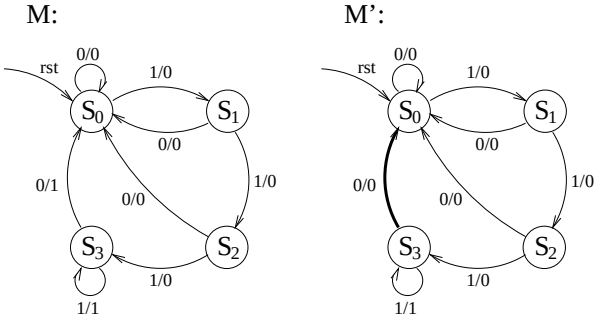

Figure 7. Given and reconfigured state machine

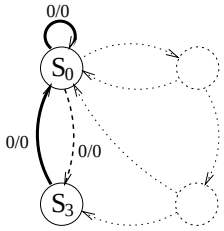

Figure 8. Reconfiguration using temporary transitions

In the following, we describe an algorithm that computes a reconfiguration program using the principle of temporary transitions. This algorithm also proves Theorem 4.1.

4.4 JSR-heuristic (Jump,Set,Return)

Let $M = (I, O, S, S_0, F, G)$ denote a state machine in its actual state $s \in S$ and $M' = (I', O', S', S'_0, F, G)$ denote a target state machine [4]. Furthermore let T_d be its set of delta transitions. Let $t_d \in T_d, t_d = (i, s_x, s_y, o)$, $H_{in} : T_d \longrightarrow S'$, $H_{out} : T_d \longrightarrow S'$, $H_{in}(t_d) = s_y$, $H_{out}(t_d) = s_x$. $H_{in}(t_d)$ returns the destination state of delta transition t_d, whereas $H_{out}(t_d)$ returns the source state of delta transition t_d. Let $S'_0 \in S'$ be the terminal state of the reconfiguration program.

Input: a FSM $M' = (I', O', S', S'_0, F', G')$ with delta transitions T_d

Output: a reconfiguration program Z

(1) $n :=$ 1;
(2) $i_0 :=$ any input state $i \in I'$ of M';
(3) $z_0:=$ reset transition; */* take reset to S'_0 */*
(4) FOR ALL $t_d \in T_d$ {

[4]No matter what state the given machine M' is in, assume it is possible to enter the state S'_0 via a reset transition on a special reset event (rst).

(5) $z_n :=$ temporary transition $(i_0, S'_0, H_{out}(t_d), -)$;
(6) $z_{n+1} :=$ delta transition t_d; /* *reconfigure* t_d */
(7) $z_{n+2} :=$ reset transition; /* *take reset to* S'_0 */
(8) $n := n + 3$;
(9) }
(10) $z_n := (i_0, S'_0, F'(i_0, S'_0), G'(i_0, S'_0))$;
(11) $z_{n+1} :=$reset transition;

First we set the index n to 1 (1). The variable i_0 represents a constant input condition used for all temporary transitions which we use later in the loop (4). i_0 is set to any input state $i \in I'$ of M' (2). No matter what state the given machine M is in, we step into the reset state S'_0 of M' first by taking the reset transition (3). Within the loop (4), we choose a temporary transition under input condition i_0 to directly jump to a state with an outgoing delta transition (5). Thereby, a new delta transition is created. We configure the delta transition we have jumped to (6) and return to S'_0 by taking the reset transition (7). Now, the last steps (5-8) inside the loop (4) are repeated for every remaining delta transition except the one caused by the temporary transition. Since in every iteration of loop (4) the same input condition i_0 is used for the temporary transition, no further delta transition is created. Finally, the delta transition caused by the temporary transition is reconfigured (10), and with the reset transition (11), we step into S'_0 and finish the reconfiguration.

Example 4.3 *Consider a given finite state machine* $M = (I, O, S, S_0, F, G)$ *with* $I = \{i\}, i \in \{0, 1\}, O = \{o\}, o \in \{0, 1\}, S = \{S_0, S_1, S_2\}$ *and the finite state machine* $M' = (I', O', S', S'_0, F', G')$ *with* $I' = I, O' = O, S' = \{S_0, S_1, S_2, S_3\}$ *shown in Fig. 6.* $T_d = \{(0, S_1, S_0, 0), (1, S_2, S_3, 0), (1, S_3, S_3, 1), (0, S_3, S_0, 0)\}$ *is the set of delta transitions and the terminal state is* S'_0.

Let us assume we are in state S_1 *of machine* M. *First, we step into the reset state* S'_0 *of machine* M' *by taking the reset transition. Then, we jump to any state with an outgoing delta transition, for instance state* S_2, *by taking a temporary transition* $(1, S_0, S_2, 0)$, *changing the existing transition* $(1, S_0, S_1, 0)$ *to a delta transition (Fig. 9-1). Now we reconfigure the transition* $(1, S_2, S_3, 0)$ *(Fig. 9-2) and by taking the reset transition, we return to the reset state* S_0 *(Fig. 9-3) again.*

Now that transition $(1, S_2, S_3, 0)$ *is reconfigured, we continue in the same manner as before with any remaining delta transition except the one caused by the temporary transition. After all other delta transitions are configured, we finally reconfigure the delta transition* $(1, S'_0, S_1, 0)$. *With the reset transition, we step into* S'_0 *and finish the reconfiguration. So, in the above example, the reconfiguration program by applying the JSR-heuristic is: Z=(*rst-transition, $(1, S_0, S_2, 0)$, $(1, S_2, S_3, 0)$, rst-transition, $(1, S_0, S_3, 0)$, $(1, S_3, S_3, 1)$, rst-transition, $(1, S_0, S_1, 0)$, $(0, S_1, S_0, 0)$, rst-transition, $(1, S_0, S_3, 0)$, $(0, S_3, S_0, 0)$, rst-transition, $(1, S_0, S_1, 0)$, rst-transition*)*.

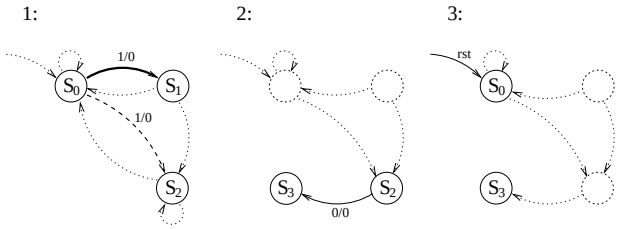

Figure 9. Reconfiguration using JSR

Though proving the feasibility of reconfiguring a machine M into a machine M', this heuristic may sometimes produce a lot of overhead. But the conclusiveness of this heuristic is the calculable length of the reconfiguration program, since it only depends on the number of delta transitions as we will see in the following.

4.5 Upper and Lower Bound

Theorem 4.2 (Upper Bound) *According to Definition 4.2, let* $|T_d|$ *be the number of delta transitions and let the FSM* M' *be resetable to* S'_0 *in every state. Then the upper bound of the length of the reconfiguration program* Z *is* $3 * (|T_d| + 1)$ *according to the jump-set-return-heuristic.*

Proof. The length of a reconfiguration program with the JSR-heuristic is independent from the transition function F' of the state machine M'. It only depends on the number of delta transitions $|T_d|$. For each delta transition we need three transitions in the reconfiguration program ($(i_0, S'_0, H_{out}(t_d), -)$,(*delta transition* t_d),(*rst-transition*)). Additionally, we need a transition at the beginning of the reconfiguration program to step into the reset state (*rst-transition*) and two more transitions to reconfigure the temporary transition at the end of the sequence ($(i_0, S'_0, F'(i_0, S'_0), G'(i_0, S'_0))$,(*rst-transition*)). So the length of a reconfiguration program is $1 + 3 * |T_d| + 2 = 3 * (|T_d| + 1)$.

Theorem 4.3 (Lower bound) *A strict lower bound for the number of transitions in a reconfiguration program is the number of delta transitions* $|T_d|$ *according to Definition 4.2.*

Proof. Since we cannot simultaneously reconfigure more than one transition at a time, the shortest reconfiguration program is a sequence of successive delta transitions according to Definition 4.2.

4.6 Evolutionary-Algorithm-Based Heuristic

Without the use of temporary transitions, the length of a reconfiguration program Z depends on the order in which

the delta transitions are reconfigured. If we treat each delta transition as a city, and the shortest path from each target state of a delta transition to each source state of another delta transition as a road, then finding the shortest path to traverse every delta transition is comparable to a traveling salesman problem. Hence, there is no algorithm that finds the optimal solution in polynomial time [5]. Therefore, in addition to the above mentioned JSR-heuristic, we have implemented a heuristic based on a evolutionary algorithm. Each individual is coded as a permutation of the order in which the delta transitions are reconfigured. The corresponding reconfiguration program of each individual is generated by connecting each delta transition with its following. This is done either by taking the shortest path to the source state of the next delta transition if the length of the path is smaller or equal than one transition, or by a reset transition followed by a temporary transition similar to the JSR-heuristic if the length of the path is greater than one transition.

The fitness of each individual is represented by the length of its corresponding reconfiguration program. The evolutionary algorithm searches for the individual with the best fitness representing the reconfiguration program with the shortest length. Table 2 shows a comparison of reconfiguration program lengths calculated with the evolutionary algorithm and the JSR-heuristic. It can be seen that even for

$\|T_d\|$	$\|Z\|$ (EA)	$\|Z\|$ (JSR)
4	10	15
17	31	54
40	55	123
72	96	219

Table 2. Experimental results of reconfiguration program lengths

finite state machines with a small set of delta transitions, the evolutionary algorithm generates reconfiguration programs which are considerably smaller (sometimes more than 50%) than those generated by the JSR-heuristic.

5 Conclusions

In this paper, we introduced a high-level, technology-independent concept of (self-)reconfigurable finite state machines. Contrary to context-swapping, a FSM implementation may be reconfigured stepwise by taking a sequence of state-transitions in which the transition and output function are gradually reconfigured.

We also have presented an FPGA-based implementation as well as bounds and algorithms for computing reconfigurations. Details for the automated mapping from the FSM specification into hardware is presented in [7].

References

[1] K. Bondalapati and V. K. Prasanna. Mapping loops onto reconfigurable architectures. In *Int. Workshop on Field-Programmable Logic and Applications (FPL)*, Sept. 1998.

[2] G. Brebner. The swappable logic unit: a paradigm for virtual hardware. In *IEEE Symp. on FPGAs and Custom Computing Machines (FCCM)*, 1997.

[3] S. P. Fekete, E. Köhler, and J. Teich. Optimale FPGA module placement with temporal precedence constraints. In *Proc. DATE 2001, Design, Automation and Test in Europe*, pages 658–665, Munich, Germany, March 13-16 2001. IEEE Computer Society Press.

[4] P. C. French and R. W. Taylor. A self-reconfiguring processor. In *IEEE Symp. on FPGAs and Custom Computing Machines (FCCM)*, pages 50–59, Apr. 1993.

[5] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1976.

[6] R. Hartenstein and H. Grünbacher(Editors). Field-programmable logic and applications : The roadmap to reconfigurable computing. In *Proc. 10th Int. Workshop FPL2000*, Aug. 2000.

[7] M. Köster. Konzept und Implementierung selbstrekonfigurierender Zustandsmaschinen. Master's thesis, Universität-GH Paderborn, Fachbereich Elektrotechnik und Informationstechnik, Fachgebiet Datentechnik, 2001.

[8] M. Motomura, Y. Aimoto, A. Shibayama, Y. Yabe, and M. Yamashina. An embedded DRAM-FPGA chip with instantaneous logic reconfiguration. In *Symp. on VLSI Circuits Digest of Technical Papers*, pages 55–56, June 1997.

[9] R. Sidhu, A. Mei, and V. K. Prasanna. Genetic programming using self-reconfigurable FPGAs. In *Int. Workshop on Field-Programmable Logic and Applications (FPL)*, Aug. 1999.

[10] R. Sidhu, A. Mei, and V. K. Prasanna. String matching on multicontext FPGAs using self-reconfiguration. In *Proc. 7th Int. Symposium on Field Programmable Gate Arrays (FPGA)*, 1999.

[11] J. Teich. *Digitale Hardware/Software-Systeme: Synthese und Optimierung*. Springer-Lehrbuch, Heidelberg, New York, Tokio, 1997.

[12] J. Teich, S. Fekete, and J. Schepers. Optimization of dynamic hardware reconfigurations. *The J. of Supercomputing*, 19(1):57–75, May 2000.

[13] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed FPGA. In *IEEE Symp. on FPGAs and Custom Computing Machines (FCCM)*, pages 22–28, Apr. 1997.

[14] S. Wadhwa and A. Dandalis. Efficient self-reconfigurable implementations using on-chip memory. In *Int. Workshop on Field-Programmable Logic and Applications (FPL)*, Aug. 2000.

[15] Xilinx. XC6200 field programmable gate arrays. Technical report, Xilinx, Inc., October 1996.