Journal of Computer Engineering
ISSN 10003428

Original Research Article
Vol 8, Issue 3, 2019 | ISSN: 1000-3428

A Comparative Analysis of Hardwired and Microprogrammed Statechart Implementations

Maria Rodriguez-Palacios and Carlos Diaz-Gonzalez

Maria Rodriguez-Palacios, Department of Electrical Engineering, KTH Royal Institute of Technology, 10044 Stockholm, Sweden

Abstract: In scientific facilities such as particle accelerators, fast and jitter-free synchronization is required in order to trigger a large number of actuators at the right time in a variety of situations. The behaviour of the control systems and subsystems may be specified by using statechart diagrams, which expand the capabilities of finite state machines allowing concurrency, a hierarchy of states, and history. Hence, there is a need of tools for synthesizing those diagrams so that a new control configuration may be deployed in a short time and an error-free manner in the required environments. In this work, we present a tool that analyses the specification of a variant of the State Chart XML (SCXML) standard tailored to hardware systems and produces a hardware description language (HDL) code suited to implement the required control systems using FPGAs. A number of solutions are provided to deal with the specific features of statecharts, such as multiple triggering events and concurrent super-states. We also present a microprogrammed architecture able to implement statecharts defined as firmware. Finally, we compare the advantages of each strategy in terms of usability, resource usage, and performance, and their applicability to a specific facility is evaluated. **Keywords:** statecharts; finite state machines; industrial control; FPGA

1. Introduction

Hardware-based control systems are required in certain environments, when perfect synchronization is compulsory. In this work we present and compare the implementation of jitter-free synchronization in research facilities using two methods: a tool that allows the automated synthesis of hardware control systems based on graphical statechart descriptions and a microprogrammed architecture [1] for the same statechart. The statechart descriptions follow a constrained version of SCXML [2] tailored to hardware systems, and are also compared to the implementations developed by a skilled engineer not only in terms of the generated code, but also in relation to the ease and speed of support, maintainability and upgradability of those implementations in deployments with certain requirements such as research facilities. We will focus on the specific case that motivated this work, which is the generation of triggering signals in a particle accelerator at the European Spallation Source (ESS) [3].

Statecharts were introduced by Harel in [4] as a tool to implement complex control systems, either in software or hardware. They may be seen as an extension of Finite State Machines (FSMs) that allow a clear specification of hierarchy and concurrency. These diagrams allow to group basic states into super-states and specify conditions and transitions at a super-state level, reducing the complexity of the specification and improving the readability. Some states may be active concurrently, making them suitable to describe

How to cite: Maria Rodriguez-Palacios and Carlos Diaz-Gonzalez (2019). A Comparative Analysis of Hardwired and Microprogrammed Statechart Implementations. Journal of Computer Engineering, 8(3), 157–177. DOI: https://doi.org/10.5281/zenodo.19340024

complex real-world systems, and the conditions to enable or disable them can be specified in an unambiguous manner. Therefore, statecharts largely improve FSMs, and they become part of the Unified Modeling Language (UML) [5].

Whereas automated software synthesis of statecharts is supported by several tools [6–8], only a few efforts have been made towards hardware synthesis of statecharts. A number of tools, most of them discontinued, provided partial solutions, but in general not all the functionalities in statecharts are implemented, specially history. A thorough literature review of hardware synthesis of statecharts is described in Section 2 together with an introduction to statecharts. Section 3 presents how an engineer would tackle the HDL specification of a statechart, and Section 4 a description of the hardware synthesis tool, listing the techniques that allow to synthesize statecharts. In Section 5 the description of the microprogrammed architecture and the procedure to write the microcode that implements the statechart is also presented. Finally, in Section 6 the code obtained from all methods is compared, focusing on the application of these methods to the ESS case, and in Section 7 the conclusions of this work are presented.

2. Statecharts

Statecharts were introduced in 1987 as a tool to overcome the limitations of FSMs [9,10] in describing the behaviour of complex systems. FSMs are state-based models where only one state is active at any given time, which can be changed by external inputs or internal conditions. The change between states is called transition.

The aspect that limits the usability of FSMs is that they can greatly grow in complexity when adding states. Statecharts deal with this issue by extending the conventional state-transition diagrams allowing for hierarchy and nested states, concurrency of parallel states and better communication among the states. This allows for more compact, expressive and modular diagrams, that can describe complex behaviour with smaller diagrams when compared to FSMs. As such, statecharts are a visual formalism for describing states and transitions. At the same time statecharts maintain all of the characteristics of FSMs, such as conditions, outputs, etc. Their main contributions are:

- Orthogonality: as opposed to classical FSMs, where only one state can be active at a time, statecharts can have more than one state active concurrently. These are called AND-states, while the traditional approach are called OR-states. Orthogonality is very useful for describing subsystems.
- Depth: there is a hierarchy in the state structure, allowing for states or even complete FSMs or sub-statecharts to live inside other states, connected with inter-level transitions. In the nested structure the state containing other states is called super-state. Depth allows for great modularity, clustering, and ease of movement between levels of abstraction by zooming in or out. It is also possible to define initial and default initial states, and have history in the states, as explained in Section 4.2.3.
- Broadcast mechanism: the different parts of a statechart are not independent of each other. An action taking place in one part of the statechart can cause many different actions in a completely different part of the statechart taking into account the orthogonality and depth. To allow this it may be necessary that many components communicate with each other even if it is not evident in the specification of the statechart.

In Section 4.2 the features of statecharts are explained in more detail.

In Figure 1, a statechart is shown. At the higher level, top is an OR super-state, because either active OR wait super-states must be running, but not both at the same time. The active

super-state is made up of two regions, send and receive. This illustrates the concept of hierarchy, as one super-state may be made up of several ones. In this case, both super-states are running in parallel, allowing to describe concurrent processing. This is called an AND super-state (denoted by the divider line). Inside each of them, only one state is active (e.g., update and save working concurrently). Contrarily, when wait is active, either idle or background are running, but not both at the same time. A black dot and an arrow point at the initial node for each super-state. Moreover, two super-states are denoted to have history (an H within the dot). Therefore when processing returns to wait, it remembers whether it was running in idle or background and, in the latter case, in which of the three nodes. History is a crucial characteristic that is seldom supported in previous works.

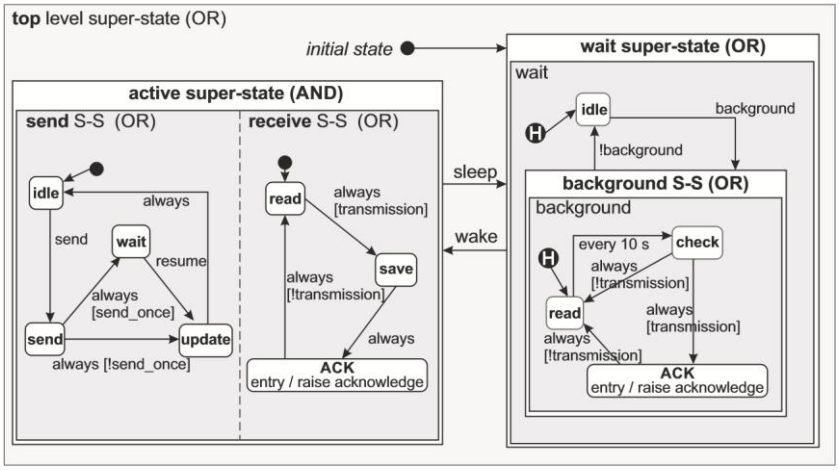


Figure 1. Simple statechart with superstates, actions and conditions.

Review on Hardware Synthesis

Soon after statecharts were proposed, a number of tools for hardware synthesis were developed.

However, most of them were limited in scope, or were based on now obsolete technologies. In 1989, Drusinsky and Harel [11] studied the challenges related to defining history and activation and deactivation of super-states. The work gives implementation hints focused on Programmable Logic Arrays (PLAs). Drusinksy and Yoresh [12] analysed the limitations of the previous work, and focused on reducing the complexity of the transitions between super-states and efficient state-coding. Again, the solutions are mainly oriented to PLA implementations. The first reference to an automated tool is found in [13], where a tool called I-Logix Express VHDL (Very High Speed Integrated Circuit Hardware Description Language) is used. However, it does not cover most of the statechart features, and the paper is mainly a description of some use cases. Another paper [14] using the same tool focusses mainly on validation, and overlooks implementation aspects, such as history. A later product by I-Logix (Statemate) supported hardware synthesis with some limitations. Although I-Logix has produced more statechart software synthesis tools, such as Rhapsody (now owned by IBM), hardware synthesis was not included in them.

In [15], a comparison is made on the complexity of implementing the system directly in VHDL code versus modelling and synthesizing the statecharts with the help of SPeeDCHART. Despite the advantages of using a graphical environment, the authors highlight that strong VHDL knowledge is still required. The same authors published a similar comparison based on implementing fuzzy control systems. Nevertheless, the provided examples are actually implemented as FSMs, as they lack of the sophistication of real statecharts. In [16], an Application-Specific Instruction Set Processor (ASIP) is proposed for mapping statecharts. Most of the paper focuses on the architecture of the ASIP. Statecharts are analysed using a tool called ROOM. History and other features are not supported. In a

later work [17], this limitation is not solved. A methodology for co-design is shown where statecharts are implemented in hardware as flattened FSMs. Furthermore, in [18] the implementation of datapaths ruled by a FSM or statechart is studied. The latter is implemented using SPeeDCHART (which was also used in [15] and [19]), and the analysis reveals that the tool does not implement concurrency in a satisfactory manner. Moreover, hierarchical superstates are flattened, rendering a single superstate
with a large number of states. The authors do not mention how they deal with history.

More recently, Qin, Chin et al. have published a number of papers [20,21] that analyse the theoretical basis of automatic conversion from statecharts to Verilog HDL. Moreover, in [22], they introduce a statechart editor and a hardware mapping tool for which some implementation examples are provided. The applicability of these works is limited by the fact that they do not support history nor transitions between states at different levels.

In [23], a methodology is presented for generating SystemC and VHDL code from a statechart specification. Much of the paper is devoted to explain how to guarantee consistency between SystemC and VHDL code behaviour when triggering events during simulation. The paper is mostly focused on SystemC, so little details are given on the generation of VHDL code. Furthermore, as in many other works, history is not supported.

Finally, Mathworks includes Stateflow [6], a commercial toolbox for Matlab that supports the statecharts formalism, including history in nested regions. An add-on to the same product called HDL Coder produces usable code for over 200 hardware platforms.

In summary, there is still a need for synthesis tools as the other existing tools, with the exception of a commercial one, cover only a partial set of features and most of them are now discontinued.

3. Hardwired Strategy

Following there is an explanation of how an engineer would implement a statechart by hand. Using an HDL such as Verilog or VHDL, the starting point consists of defining a process for each leaf superstate. That is, those that do not contain other superstates. Each of those processes encode a FSM with as many states as there are in each superstate. Moreover, a registered activation bit is defined for each of them. The code for every process checks the activation bit. If active, a switch-case construct will be used to write code for each basic state, which may include: generating outputs, updating the internal state, producing events (signals) for other processes and switching off the current superstate and sending an enabling event to a different one.

Inactive superstates evaluate if the conditions to be enabled are fulfilled. In such a case, the activation bit will be switched on. Hierarchy is not implemented by nesting processes, but by the way in which events are produced and consumed by the different processes. Looking at Figure 1, event wake will activate both send and receive. There is not a process for superstate active. Furthermore, sleep will be processed by all the superstates. First, send and receive will become inactive.

Moreover, the internal state will be recovered if history is implemented or, otherwise, reset. In some cases, a superstate may become active or inactive under different events. Thus, some processes must register additional information that enables the code to decide when and how to activate the superstate. This is sometimes called deep history. As an example, wait/idle will activate itself only if it was disabled by wake, but not by background. Similarly, background will decide whether it should resume or not.

Most of the communications between processes are carried out using events (signals). In some cases, an event or an output may be produced by different processes. In such a case, all the sources are or-ed to produce an unified event or output. Hence, inactive processes must produce zero-valued outputs and events. Beside the superstates, additional processes

could be defined in order to produce and/or consume specific events. This includes all kind of counters or alarm triggers.

In summary, the well-known methodology for synthesizing FSMs must be extended with the addition of an on/off bit. Furthermore, a simple mechanism for implementing history is needed; and hierarchy is modelled by raising and consuming events. It must be remarked that all processing is specified with a clock cycle granularity. Therefore, actions and transitions must be completed in the duration of a clock cycle. Those actions that may span several cycles (such as iterations) could require splitting into different states.

4. Automatic Synthesis of Statecharts

4.1. Statechart Parsing and Analysis from a Graphical Tool

There are several graphical tools that allow designing and testing statecharts, as well as implementing them in form of C, Java or other software programming languages. Those are part, in most cases, of UML tool suites. However, as described in Section 2, there is a lack of tools that can perform the hardware synthesis of statecharts in a successful manner without sacrificing some of statechart's main characteristics. An automatic tool has been developed by us that is able to produce HDL code starting from a graphical description of a statechart without imposing important requirements on the statechart. For the development of our tool we have used Yakindu Statechart Tools (Yakindu SCT) [7], which is specifically focused on statecharts. Yakindu SCT saves the statechart model in a XML file describing the regions, states, history nodes, etc, and the transitions between them.

The selection of Yakindu for designing the statecharts determines the statechart variant used as a starting point for our application, State Chart XML (SCXML). Nevertheless, since the implementation of statecharts in hardware systems present some unique characteristics compared to the software implementation, which is the main target of SCXML, some aspects have not been implemented in our tool, such as "invoke", the errors and logs, or the data model.

Our application processes the XML description of the statechart using Xerces C++ [24], a validating

XML parser. It provides a shared library for parsing, generating, manipulating and validating XML documents using different APIs; we have chosen the DOM [25] API. Although a tree walker that goes through the document node by node is provided, it does not allow enough flexibility to produce the HDL code in a well structured manner. Therefore, we used the parser in a more flexible way, defining in each step a current node that allows moving back and forth through the structure.

In our current implementation some restrictions are assumed in order to ease the design of the application:

- The names of the elements do not include spaces.
- Only events and internal conditions are accepted as transition triggers: external events, counters or integer variables having a specific value are accepted, but more general conditions, such as "after one second", valid in the Yakindu SCT model, are not accepted. Instead, an internal or external counter is used to measure time spans as a number of clock cycles. Using the clock period as a time unit allows greater accuracy in high-speed signal processing.
- Only shallow history is supported: shallow history is defined as a pseudo state that remembers the last active state inside the region that includes it. Deep history, which remembers all the latest states of a hierarchy of multiple nested states, is not currently supported.
- All the states in the statechart must have different names.

Although a formal study of the consistency and validity of our statechart variant has not been done, the features of SCXML that have not been implemented are not part of the core of statecharts. They are also mainly related to software implementations and do not transfer easily to hardware designs. The only exception to this would be the restriction on deep history. Most of the restrictions are in place to rule out cases that are improbable or even impossible in our goal implementation. Furthermore, some restrictions may reduce the complexity of the implementation. Hence, we firstly considered that transitions only could happen between states in the same region and level of the hierarchy. That did not inhibit the implementation of some systems, but encouraged a better specification of the statechart. However, that restriction has been finally eliminated without a significant complexity increase.

4.2. Synthesis of Super-States

The synthesis of FSMs is well understood [26,27], and a number of tools exist able to convert graphical representations into HDL code. However, statecharts introduce new features, such as super-states and history, that require new synthesis techniques. So far, synthesis has chiefly been addressed from a software point of view [28].

By analysing the XML description of the statecharts provided by Yakindu SCT, a basic implementation of a super-state can be obtained using the same procedure as for FSMs, which consist of defining: (a) state encoding, (b) a state transition function, and (c) an output generation function. However, some characteristics of super-states may be more challenging to implement, such as:

- Orthogonality (or concurrency).
- Depth (or hierarchy).
- History.
- Distributed generation (super-states generating the same output or event).
- Entry, exit and event-driven actions inside a state, conditions, and actions on transitions.

4.2.1. Orthogonality

While FSMs can only have one state active at a time, statecharts can have two or more states active concurrently. This can happen when the two states are in different levels of the hierarchy, as explained in Section 4.2.2, or in the same level (orthogonality). In the latter case they are called AND-states. Each super-state can have different regions, with each region implementing a FSM, and all of the FSMs being active at the same time. Although each region should run almost independently of the concurrent ones, they can affect each other, for example if one event is raised by one of the regions, and that same event has an effect on a different region. Our application implements each region as a different process that runs concurrently with the rest of the processes.

For example, let us consider the statechart shown in Figure 1. Inside the active super-state there are two parallel regions, named send and receive, and in this simple case the two regions implement independent FSMs that do not have any effect on each other. Listing 1 shows an extract of the XML file of the model that describes the two parallel regions, and Listing 2 shows an extract of the VHDL code generated by our application that implements these two regions as processes.

Listing 1. Extract of the XML file from Figure 1 that describes the two parallel regions (edited for clarity).

```
<verticesxsi:type=" sgraph:State "xmi:id=" . . . " name=" active "incomingTransitions=" . . . ">
<outgoingTransitions xmi:id=" . . . " specification=" sleep " target="
. . . "/> <regionsxmi:id=" . . . " name=" send ">
[ . . . ]
</regions>
<regionsxmi:id=" . . . " name=" receive ">
[ . . . ]
</regions>
</vertices>
```

Listing 2. Extract of the output VHDL file implementing the two regions described in Listing 1 (edited for clarity).

```
sendFSM : process ( sleep ,wake ,send , resume ,background ,
                    sendCurrentState )
begin                case
sendCurrentState is
[ . . . ]
end case ;
end
process ;

receiveFSM : process ( sleep , wake , send , resume , background ,
receiveCurrentState ) begin case receiveCurrentState is
[ . . . ]
end case ;
end
process ;
```

4.2.2. Depth

As opposed to FSMs, statecharts can have different levels of hierarchy with states (or even complete FSMs) living inside other states, giving the statechart a sense of depth. In this case,
when a super-state is left, all the states underneath it should be disabled, and when the super-state is entered, the FSM inside it starts or continues its operation. This can be implemented in two ways:

- Each region is automatically disabled when the super-state containing it is disabled, and it is also re-established automatically when the parent super-state is enabled. In some unusual transitions between states in different regions it may be a sub-state that disables or enables the super-state containing it.
- Each region monitors all the transitions and events happening in any place of the statechart, not only the transitions in, to, or from its neighbouring regions, and if appropriate it reacts to them. In this case each region does not react to states being enabled or disabled, but rather to the transitions themselves, and each region needs to determine if the transition affects it (for example by disabling a super-state that down in the hierarchy includes the region under consideration) to react as expected.

Although in some cases the first option could be more suitable, we have implemented the second one for one main reason: in transitions where the source and destination states are not in the same region and super-state, it would be the sub-states enabling and disabling the super-states where they live, and not the other way around. Then this enabling or disabling action needs to be cascaded down from the super-state downwards in the hierarchy. It is even possible that a transition from a state enables or disables super-states several levels up in the hierarchy, which would trigger a lot of disabling actions. If the regions

or states are aware of the transitions instead, the produced HDL code is shorter and smaller circuits are synthesized. Listing 3 shows the output of our application for the wait region in Figure 1.

Listing 3. Extract of the output VHDL file implementing the wait region (edited for clarity).

```
waitFSM : process ( sleep ,wake ,send ,  resume ,background       ,
                  waitCurrentState )
begin
case waitCurrentState is
when idle =>
if wake ='1 ' then
waitHistReg     <=     0;
waitNextState          <=
waitEntry ; else
if background ='1 ' then
waitNextState          <=
background ; end if ; end
if ;
when  background
=> if wake = '1 '
then
waitHistReg     <=     1;
waitNextState          <=
waitEntry ; else
if background ='1 ' then
waitNextState    <=
idle ; end if ; end if ;
when   waitEntry
=> if sleep = '1 '
then
case waitHistReg is
when       0       =>
waitNextState    <=
idle ;
when 1 =>
waitNextState <= background ;
end  case  ;
end   if   ;
end  case  ;
end
process ;
```

The hierarchy can have any number of levels, so a sub-state in a super-state can at the same time be a super-state. Our application uses recursive functions to go through all the levels of the hierarchy to complete the statechart model.

4.2.3. History

A super-state has history if it can remember its present state and return to it later after being disabled for a while. A super-state without history, however, will always resume its activity starting at its initial state. Implementing this behaviour can basically be achieved in two manners. In the first way, a wait state is added, and a copy of the current state is kept on a register. A disabled super-state will be running in its wait state until a resume event is triggered, then the saved state will be used. Alternatively, it may be simpler to add a wait

state attached to each normal state and transit to it when the super-state is disabled. The convenience of each solution should be assessed for each particular case but, in our current application, only the former is implemented, which can be seen in Listing 3. In this example the wait state is named waitEntry and it stores the state in the waitHistReg register.
The wait state is also used as the entry node, but its initialization is not shown in the listing.

Deep history consists of the ability of resuming to the right state even it is deeply buried in a hierarchy of regions. Currently, our tool does not implement deep history. Instead, processing will resume to the highest region in the hierarchy, whether it was previously active or not. If that region has history, it will be correctly implemented. Furthermore, when the previously active region becomes active again, processing will resume to the right state, as history is not reset.

Despite not being operational in our tool, a simple mechanism would allow implementing deep history. It would consist of adding an extra wait state, or deep history inner state, in the region with the deep history state, and all the regions inside it in the hierarchy. This extra wait state does not replace the normal wait state that may be present in the regions, since they will be activated under different conditions and transitions. Nevertheless they can share the register that holds the saved state, so the extra wait state becomes just a flag.

4.2.4. Distributed Generation

Ideally, each super-state generates a sub-set of outputs and events different to other super-states. However, this is not always true, either because the statechart is not neatly designed or because using different super-states actually helps to organize the statechart. Hence, the implementation of each super-state may depend on others. A preliminary parsing of the statechart description finds which outputs and events are produced by different super-states, and asserts that only one of them is active at the same time. Next, output and event signals are renamed by appending a suffix that relates each signal to the super-state in which it is produced. Finally, the global signals are obtained by reduction as: *globalSignal* = (*signal_ss*1 *and active_ss*1) *or* (*signal_ss*2 *and active_ss*2) *or*….

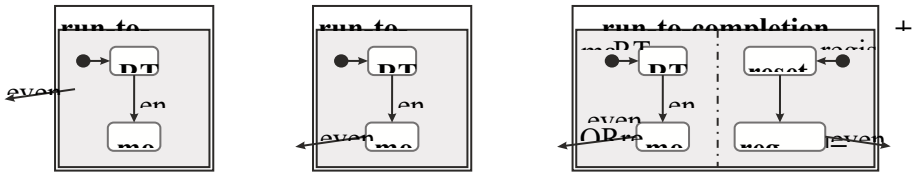
4.2.5. Actions and Conditions

Entry and exit actions are not unique to statecharts, since FSMs can also model them as part of their inputs and outputs. However, in statecharts, they are expanded by allowing actions to happen when some conditions are met at any moment during the active phase of a state. For example, a state may increase a counter when an event is raised anywhere in the statechart, but the counter would not increase if the specific state is not active. In this case the statechart models the counter-increase action as being performed by the active state. Conditions can also apply to any transition, action, event, etc.

Transitions are condition-dependent. Conditions are evaluated using if-elsif-else statements, which in VHDL introduces a priority order even if, in many cases, conditions are mutually exclusive. However, as it is pointed out in [29], conflicting transitions may happen when more than one condition applies at the same time on a given state leading to different transitions. In such a case, our tool will give priority to transitions specified at a higher level. In the case of transitions in the same region, those with higher priority will be coded first. Actually, the latter does not require any specific programming, as Yakindu XML-files list the transitions in order of priority.

Some tasks may require several clock cycles before completing. In those cases, it may happen that exiting the current superstate before completion is not desired. This is called Run-to-Completion (RTC). In the context of this tool, RTC is specified and implemented as shown in Figure 2. The first superstate will exit if event is triggered, even if that it is not desired. The second one, however, will not react to event until the task in completed.

However, there could happen that event is only active during some cycles before completion. Therefore, the third superstate registers event using a concurrent superstate so that the event is not missed. Similarly, more than one event could be registered and a priority scheme could be implemented.



RTC not guaranteed RTC could miss event RTC with 1 event registered

Figure 2. The problem or Run-to-Completion is presented. RTC is not guaranteed if the event affects the whole superstate. A simple implementation of RTC is achieved when only more reacts to the event. In order to avoid missing an event, however, a concurrent process may be required to register incoming events before completion.

4.2.6. VHDL Implementation Steps

Our application performs the parsing of the XML file and generation of the VHDL file in a series of steps:

- Previous to the HDL generation, the XML file is parsed once to get a structural description of the statechart. This includes creating lists of: associated ids and names, associated transitions and triggers, states, and history nodes.
- When the main parsing is performed, the first step is the declaration of the entity, with the name of the statechart and the list of in and out events from the statechart interface list.
- Then the architecture body starts defining each of the types of regions' states and history nodes and defining two signals that will hold the current and next state for every region. Signals for internal variables and registering and saving the active states in transitions with history are also defined in this step.
- Next the architecture body begins. Each region is implemented as a VHDL process which is basically a case-when list of the current state in that region. This is the most complex step, and where the considerations explained by Section 4.2 express themselves. Recursive functions are used to ensure that the statechart is implemented completely.
- The VHDL code is finished with a synchronization process that, synchronously, updates the current state of each region process with the next state as determined by the normal operation of the statechart.

5. Microprogrammed Architecture

A microprogrammed architecture [1] for statecharts is now described. The basic element of this architecture is a circuit able to run one or more superstates in a non-concurrent manner. The circuit is able to switch between superstates connected by transitions and implement shallow history.

In order to implement the concurrency required by AND-superstates, as many of those circuits must be instantiated as superstates may run in parallel at the same time. Each circuit is made of a micro-memory that stores one micro-instruction for each basic state. Each micro-instruction encodes the actions to be carried out by that state, and all the possible transitions and the conditions (event) that would trigger those transitions. The circuit will read one micro-instruction each cycle using a program counter (PC), decode it, execute the actions, and select the next value for PC after evaluating the list of conditions and transitions.

All instances have access to the same set of inputs, outputs and internal variables. Events derived from inputs and variables influence the transitions on different instances, establishing connections between superstates. For example, one superstate may change a common variable in order to trigger an event in another concurrent one. In the following paragraphs, the implementation is described in detail.

Each instance is made of the following elements: micro-memory, PC, a set of registers to implement history, the logic to decode and evaluate the conditions and transitions, and the logic to decode the actions to be performed. At least, as many instances are needed, as concurrent superstates but, in order to enable future upgrades, a slightly larger number is recommended. The use of double-port memories allows sharing part of the cost between instances, especially when each of them implements only a few states. The length of the micro-instructions may be significantly large, especially in cases with many inputs and/or variables and outputs. Therefore, several memory modules are often required for each instance, and the advantage of sharing becomes more evident.

The program counter represents addresses in the micro-memory, but this may require a translation in some cases. Thus, when switching to a new state within the same superstate, the microinstruction provides the exact address of the next state. However, when switching to a different superstate, a special code is given instead. That code is an index in the range of 0 to 7, for example, and it is translated to a real address using the history register at that index. Thus, the first addresses are reserved and any transition to one of those addresses is translated to a real one. At the same time, if that superstate implements history, the current content of PC is stored in the history register of the current superstate. In this way, future transitions back to the exiting superstate will resume at the right state/micro-instruction. The initial values for the history registers are loaded at configuration time, at the same time as the content of the micro-memory, and a set of flags that signal whether each superstate implements history or not. Figure 3 shows how micro-memories, PC, and history registers work together.

Inputs to the system are key in decision making. Events based on inputs are produced by comparing each input value to a set of reference values (we propose 2 for each input). Hence, events such as input1 is lower than ref1 can be used as conditions for any of the running microprograms.

Counters are used as internal variables. The value of each counter is also compared to reference values producing events similarly to inputs. A micro-instruction may update the content of any counter by issuing a command that specifies loading or adding a reference value, incrementing or decrementing it.

Finally, outputs are selected by the micro-programs assigning one of the counters or a predefined value. Optionally, each individual output may be configured to be registered. Figure 4 shows the basic structure of the three types of components.

In order to deal with a mix of OR and AND superstates, all the microprograms run concurrently, even if no real work is done. Hence, a microprogram may be running a ghost superstate until it branches into an activated AND superstate. Ghost superstates are made of micro-instructions that evaluate the conditions that may lead to branch to another superstate, but they do not change any internal variable or produce any output. Figure 5 shows how to implement the complete statechart in Figure 1 using two microprograms. The second one will mimic all branches until arriving to receive, where real actions will be carried out. The same scheme may be applied for a larger number of micro-programs and deeper hierarchies.

It is not possible to implement deep history using the proposed architecture, as it does not implement call-return, just branching. Hence, the engineer must choose a fixed superstate to return to. However, each superstate history is not lost, so that when entering that superstate, execution will resume at the right micro-instruction.

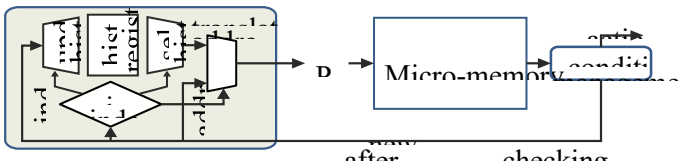
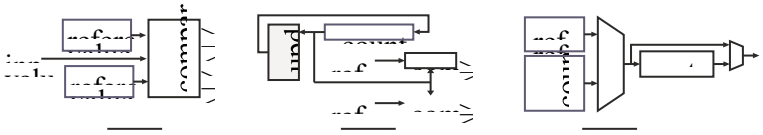


Figure 3. Basic microprogrammed circuit with micro-memory. Microinstructions are evaluated and condition management produces a new address. If the address is actually a super-state index, a translation obtain the real address and the history registers are updated if history is implemented for the exiting super-state.



input counter output

Figure 4. Basic scheme of inputs, counters and outputs in the microprogrammed architecture. Every input value is compared against 2 reference values. The resulting 6 flags are used as conditions by the microprograms. Counters work in a similar way, but they are updated according to commands issued by the microprograms. Output modules a basically a mux with an optional output register.

wait
idle
background
read
check
ack
send
idle
send
update
wait

ghost wait
idle
ghost backgd
read
check
ack
receive
read
save
ACK

Figure 5. Main and ghost micro-programs for the statechart in Figure 1. The micro-program on the right side mimics all the transitions of the main one without performing real work. When branching to receive, real actions are taken until that super-state is left again.

5.1. Micro-Instruction Format

Micro-instructions are made of two fields. First, a set of conditions to decide where to branch. Each condition is made of: the index of the event to be evaluated; a comparison (<=>); inverting flag; the reference value to use; the chain bit; and the target address. Simple conditions may be combined (and operand) by setting the chain bit. The or operand is implemented by using the same target for different conditions. The second field specifies the actions to be taken: updating one or more counters or producing a given output. Figure 6 shows the format for the statechart in Figure 1. Allowing several conditions increase micro-instruction length, but limiting the number requires splitting the evaluation. Figure 7 shows an example, when the format is so narrow that only two conditions may be encoded in the

same micro-instruction. In such cases, engineers must assess the impact of using extra cycles.

5.2. Configuration

A chain of registers store the initial addresses for each superstate; the flags that support history; the reference values for the inputs, counters and outputs; and the initial values of the counters.

The configuration is loaded byte by byte and it propagates to the last element in the chain, where the micro-memories are located.

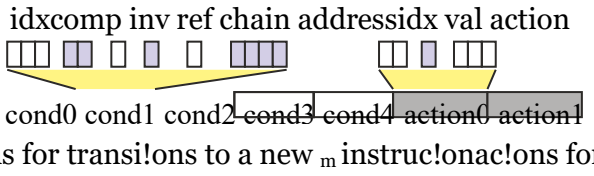


Figure 6. Micro-instruction format for the statechart in Figure 1 including bit-fields. At least five conditions and two actions are needed. Each condition requires 12 bits; each action 6.

	condi!on	target	condi!on	target
start:	if a	eval_b	else	state_not a
eval_b:	if b	state_ab	else	state_a_notb
state_not a:	if b	state_not a_b	else	eval_c
eval_c:	if c	state_c	else	others
…				

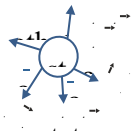


Figure 7. Three-step evaluation of five conditions when only two may be addressed at the same time. This situation corresponds to mapping a statechart onto a micro-programmed architecture that was originally planned for simpler designs.

6. Evaluation

The proposed methods are now evaluated using four different use cases: the example from Figure 1, the digital watch proposed by Harel in his original work [4] and the two main components of the ESS timing system—the event generator (EVG) and the event receiver (EVR)—which provides the fast and jitter-free synchronization that is required to successfully run such a complex machine as ESS. Hence, one engineer introduced the diagrams in Yakindu and used our tool to produce VHDL code in an automatic way and implemented the statecharts. A second engineer analysed the statecharts and produced VHDL code by hand and implemented them using the microprogrammed architecture.

A recreation of the original Harel's diagram is presented in Figure 8. From it, alarm1-status, alarm2-status, and dead have been later removed as they have been found to be redundant with other states and the reset signal of the circuit. All the other aspects have been implemented, except deep history. In the hand-coded version; however, deep history has been included without much effort.

The EVG and EVR are shown in Figures 9 and 10. Their structure is not as complex as that of the digital watch, apart from the fact that a significant number of AND superstates are required. In that sense, the digital watch tests the capability of correctly implementing

most of the features of statecharts, while the remaining ones are used to compare the quality of the code in relation to the hand-written one.

These examples illustrate the main concepts of statecharts but they differ greatly in complexity. Harel's watch, EVG and EVR specify a large number of AND-superstates, several inputs and conditions, but only Harel's requires deeply nested hierarchy. We refer to the original papers [4,30]. Table 1 summarizes the main characteristics of the examples. Some of the figures have been expanded manually (between parentheses) in order to allow for some upgradability, which could extend the useful life of an embedded system.

Table 1. Evaluation of four statecharts: main characteristics. For the sake of upgradability, some figures have been expanded (shown between parentheses).

		Figure 1	Harel (Figure 8)	EVG (Figure 9)	EVR (Figure 10)
states					
superstates					
max. superstates	AND	2 (4)	7 (8)	9 (10)	12 (14)
events		7 (16)	17 (32)	28 (32)	23 (32)
counters		1 (4)	8 (16)	6 (8)	12 (16)
outputs		1 (4)	4 (16)	8 (16)	12 (16)

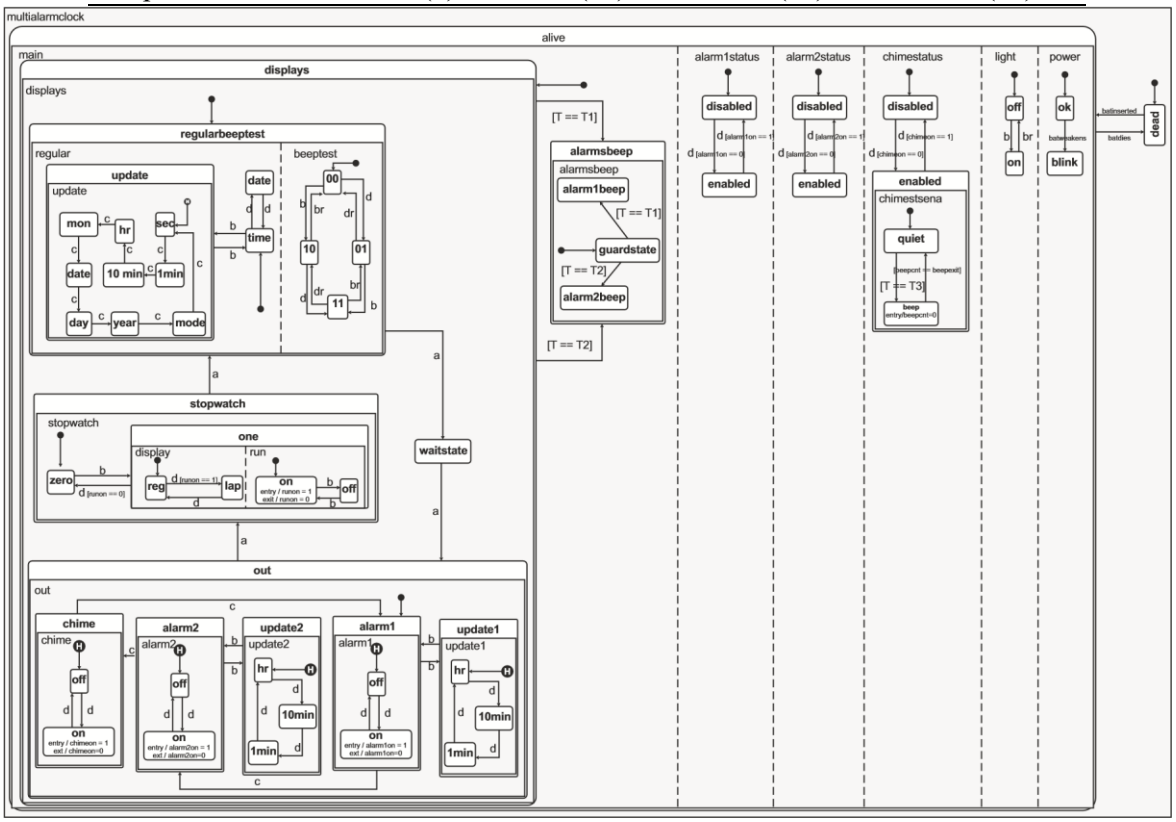


Figure 8. Recreation of Harel's description of a statechart to control a digital watch.

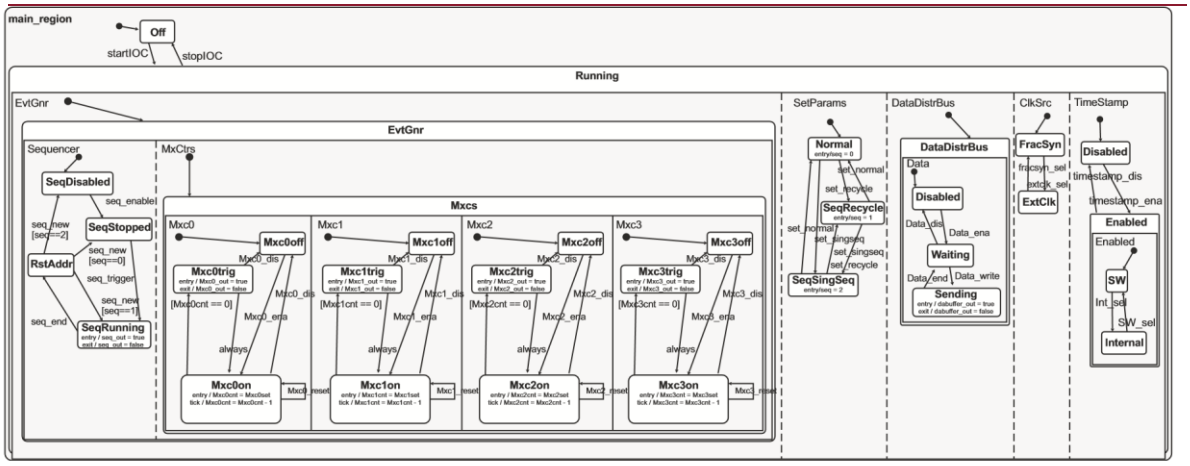


Figure 9. Statechart of an EVG from the ESS control system.

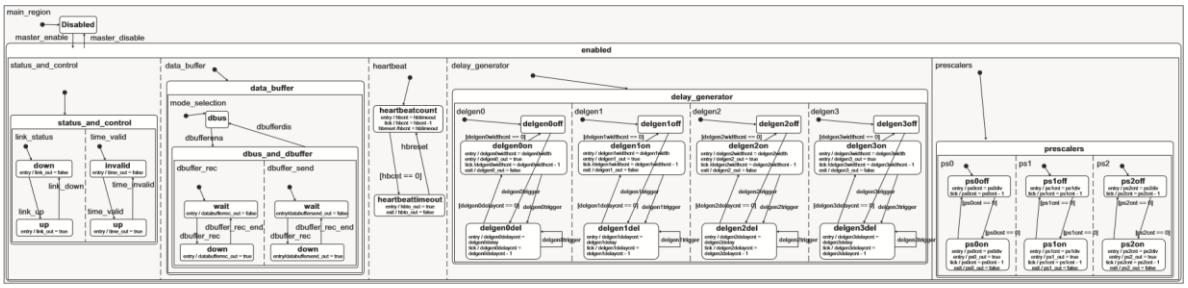


Figure 10. Statechart of an EVR

The implementation of the example from Figure 1 will be analysed first using the data from Table 2. As it can be seen, the hand written code is more concise that any other option as: the automatic tool produces verbose code; and the microprogrammed architecture describes large multiplexers for condition and action selection (those 689 lines of VHDL code do not include the firmware). The number of logic blocks (Xilinx Kintex-7 look-up-tables) and flip-flops are similar for the hand-written and automatic implementations, but the microprogrammed architecture results in a large overhead. The latter architecture is the only one that requires RAM memories. Each microinstruction is 102 bits wide, requiring four 32-bit words. Thus, four double-port memories will support two AND-superstates. Then, twice that amount is needed to host up to four concurrent AND-superstates. Finally, the microprogrammed architecture is significantly slower than the hardwired ones, mainly due to the use of memory blocks and large multiplexers. This may have an impact on the performance of the systems or be irrelevant for a particular application. The difference in speed between the two hardwired circuits is not significant.

Table 2. FPGA resource utilization and clock cycle (delay) for the statechart in Figure 1 using three methodologies.

	Hand-Coded	Automatic Tool	Microprogrammed
Lines of code			
Logic blocks			8416
Flip-flops			1872
RAM (18 Kbit)			
delay (ns)	1.29	1.33	8.56

Table 3 shows results for Harel's watch. As in the previous case, hand-written code is shorter and produces the smaller circuit. The automatic tool produces a similar circuit using a more verbose code. Again, the microprogrammed architecture is both significantly larger

and slower than the other two architectures. In this case, each micro-instruction is 130-bit long, requiring five 32-bit words. Hence, five block of memory may host two AND-superstates, and four times that amount will host the planned eight concurrent AND-superstates.

Table 3. FPGA resource utilization and clock cycle (delay) for the statechart proposed by Harel [4] using three methodologies.

	Hand-Coded	Automatic Tool	Microprogrammed
Lines of code		1483	
Logic blocks			16,203
Flip-flops			4776
RAM (18 Kbit)			
delay (ns)	1.59	1.61	8.70

The implementation results for EVG and EVR are similar and they are shown in Table 4. As with the digital watch, the hand-coded implementations use slightly less resources than those generated by the tool. A quick inspection revealed that, in all three cases, the tool was using more flip-flops to encode the hierarchy of states, whereas the engineer decided to flatten the structure. As mentioned in Section 3, superstates that only contain other superstates are not implemented. The microprogrammed implementations are similar to the digital watch, as the number of resources where expanded in the same way (see Table 1). The differences are mainly attributed to the cost of managing a different number of AND superstates. Importantly, neither EVG nor EVR require deep-history. Harel's watch does, but it may be implemented using simple history without losing usability.

Table 4. FPGA resource utilization and clock cycle (delay) for the event generator (EVG) and event receiver (EVR) statecharts using three methodologies.

	EVG			EVR		
	Hand	**Tool**	***μ*Prog**	**Hand**	**Tool**	***μ*Prog**
Lines of code						
Logic blocks			16,422			19,857
Flip-flops			4686			5108
RAM (18Kbit)						
delay (ns)	1.73	1.60	8.81	1.84	1.68	8.70

Next, each strategy will be analysed comparing the implementation of the four statecharts. For the hardwired architectures, the number of lines of code and hardware components is proportional to the complexity of the statechart. The delay, however, does not increase significantly, as the critical path is defined by the most complex super-state, not the aggregation of many. This suggests that the synthesis of even larger statecharts should still produce fast circuits.

The circuits obtained using the automatic tool are comparable to those made by hand. A small resource overhead has been detected due to the implementation of some superstates. As hierarchy is more neatly implemented in the current way, we do not consider the need of rewriting the tool to optimize those cases. The main advantage of using our tool is the possibility of applying changes to the statechart in minutes and obtaining the HDL code in seconds. As a drawback, deep-history cannot be implemented.

The code size for the microprogrammed architectures is similar in all cases, as many of the components are the same even if they are instantiated in different amounts. The maximum differences are a factor 2.4 in logic blocks; and a factor 2.7 in flips-flops. In all cases the delay is very similar. The complexity of the microprogrammed architecture is largely due to the micro-instruction format, which is defined by the worst case. Particularly, state check in Figure 1 requires evaluating five conditions. As a consequence, the architectures are not very different. A shorter format could have been chosen, as shown in Figure 7, if some extra delay is acceptable. Eventually, the main difference is in the number of registers for reference values and the multiplexers that, as Table 1 shows, are double the size in most cases. The delay in these architectures is mainly due to accessing the micro-memories and propagating the signals through large multiplexers. It can be seen that there is not a great increase, as the delay in multiplexers grows logarithmically with the number of inputs.

Any of the proposed examples can be implemented in any modern FPGA, with the exception of the smaller devices of some series. Furthermore, as cost does not grow exponentially with the complexity of the statechart, it is feasible to synthesize an extended architecture with extra inputs, counters, and outputs.

Although difficult to quantify, there are other advantages of using our tool and methodology in terms of ease and speed of support, maintainability and upgradability, which are compulsory in some cases, specifically in research facilities. The application that motivated our work presented in this paper, the ESS timing system, has some components that have a very flexible hardware, including FPGA Mezzanine Cards (FMCs), that require hardware reconfiguration. Some other timing system components are located in places where access is difficult or restricted, for example because the physical dimensions of the component location are tight or blocked, or because there may be radiation in the environment. In these cases it is important that the update process is fast and simple, since the time to perform it may be short (for example during a shutdown period of the facility) and it may be a technician without HDL experience performing the update, so using a visual tool such as statecharts to model the system is a big advantage. The new configuration should also be error-free, since the problems that could arise may be impossible to solve until the next shutdown or update period (in some cases planned for months in the future). They may even prevent the correct operation of some instruments, causing delays and extra costs in the experiments. In this kind of situations the work here presented is better and faster than the classical approach of writing by hand the hardware configuration of the systems. Our tool makes it easy and simple to update the hardware configuration even with no HDL programming experience, using a graphical approach and importing only one file to our tool. It also keeps the chance of errors as low as possible by automatically generating a correct VHDL file. Our methodology based on microprogramming allows updating and deploying configurations quickly without logic synthesis, since for the deployment only a new configuration is needed instead of a new bitfile. Furthermore, there is no dependency on the version of the synthesis software, so maintainability is easier. These characteristics are very important in some applications. Although the needs of ESS triggered the development of the work here presented, both the tool and methodology can be used for any other cases where the target system can be implemented as a statechart.

Each of these implementation options may produce circuits that are faithful to the original design in different degrees. Despite being time-consuming and error-prone, hand-written code has the potential to implement any characteristic of the statechart. The automatic tool and the micro-programmed architecture have the main limitation of not implementing deep-history. However, it is not expected that this produces malfunctioning problems. In the case of microprograms, it is possible that the architecture cannot evaluate

complex conditions in a single cycle. In that case, evaluation is split in several cycles, and the resulting latency may be noticeable.

7. Conclusions

Statecharts are a useful tool for specifying embedded control systems in both software and hardware. Three methods are compared for hardware synthesis. We have obtained small and fast implementations using our automatic tool, almost as efficient as human-written code. Statecharts may also be implemented using a microprogrammed architecture. In that case, a significant overhead must be expected in both the number of components and clock-cycle length. However, the benefits of an upgradable architecture must be considered, as it allows to deploy mission-critical control systems that may be updated as needed without physically accessing the hardware, such as nuclear facilities or space probes.

References

1. Milkes, M. The Genesis of Microprogramming. *IEEE Ann. Hist. Comput.* **1986**, *8*, 116–126. [CrossRef]
2. State Chart XML (SCXML). Available online: https://www.w3.org/TR/scxml (accessed on 3 July 2020).
3. European Spallation Source. Available online: https://europeanspallationsource.se (accessed on 3 July 2020). 4. Harel, D. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* **1987**, *8*, 231–274. [CrossRef]
5. Unified Modeling Language. Available online: https://www.uml.org/ (accessed on 3 July 2020).
6. Mathworks. Stateflow HDL Coder. 2018. Available online: https://www.mathworks.com/products/hdlcoder.html (accessed on 3 July 2020).
7. Yakindu Statechart Tools. Available online: https://www.itemis.com/en/yakindu/state-machine/ (accessed on 3 July 2020).
8. IBM. Rhapsody. 2020. Available online: https://www.ibm.com/products/systems-design-rhapsody (accessed on 3 July 2020).
9. Mealy, G.H. A method for synthesizing sequential circuits. *Bell Syst. Tech. J.* **1955**, *34*, 1045–1079. [CrossRef]
10. Moore, E.F. Gedanken-Experiments on Sequential Machines. In *Automata Studies*; Princeton University Press: Princeton, NJ, USA, 1956; pp. 129–153.
11. Drusinsky, D.; Harel, D. Using statecharts for hardware description and synthesis. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **1989**, *8*, 798–807. [CrossRef]
12. Drusinsky-Yoresh, D. A state assignment procedure for single-block implementation of state charts. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **1991**, *10*, 1569–1576. [CrossRef]
13. Clemente, P.; Rundstadler, P.; Specter, L.; Walsh, K. From Statecharts to Hardware FPGA and ASIC Synthesis. In Proceedings of the Spring VHDL International Users' Forum, Scottsdale, AZ, USA, 3–6 May 1992.
14. Kol, R.; Ginosar, R.; Samuel, G. Statechart methodology for the design, validation, and synthesis of large scale asynchronous systems. In Proceedings of the Second International Symposium on Advanced Research in Asynchronous Circuits and Systems, Fukushima, Japan, 18–21 March 1996; pp. 164–174.
15. Salapura, V.; Waleczek, G.; Gschwind, M. A comparison of VHDL and Statecharts-based modeling approaches. In Proceeding of the ITI, Pula, Croatia, 14–17 June 1994.

16. Veith, C.; Buchenrieder, K.; Pyttel, A. Mapping Statechart Models Onto an FPGA-based ASIP Architecture. In Proceedings of the EURO-DAC'96, European Design Automation Conference, Geneva, Switzerland, 16–20 September 1996; pp. 184–189.
17. Buchenrieder, K.; Veith, C. A Prototyping Environment for Control-oriented HW/SW Systems Using State-charts, Activity-charts and FPGA's. In Proceedings of the EURO-DAC'94, European Design Automation Conference, Grenoble, France, 19–22 September 1994; pp. 60–65.
18. Muller-Wipperfurth, T.; Hagelauer, R. Graphical entry of FSMDs revisited: putting graphical models on a solid base. In Proceedings of the Design, Automation and Test in Europe, Paris, France, 23–26 February 1998; pp. 931–932.
19. Salapura, V.; Hamann, V. Implementing fuzzy control systems using VHDL and statecharts. In Proceedings of the EURO-DAC'96, European Design Automation Conference, Geneva, Switzerland, 16–20 September 1996; pp. 53–58.
20. Qin, S.; Chin, W.N. Mapping Statecharts to Verilog for Hardware/Software Co-specification. In Proceedings of the FME 2003: Formal Methods, Pisa, Italy, 8–14 September 2003; pp. 282–300.
21. Qin, S.; Chin, W.N.; He, J.; Qiu, Z. From Statecharts to Verilog: A Formal approach to hardware/software co-specification. *Innov. Syst. Softw. Eng.* **2006**, *2*, 17–38. [CrossRef]
22. Tran, V.A.V.; Qin, S.; Chin, W.N. An Automatic Mapping from Statecharts to Verilog. In Proceedings of the Theoretical Aspects of Computing—ICTAC 2004, Guiyang, China, 20–24 September 2004; pp. 187–203.
23. Findenig, R.; Leitner, T.; Esen, V.; Ecker, W. Consistent SystemC and VHDL Code Generation from State Charts for Virtual Prototyping and RTL Synthesis. In Proceedings of the DVCon 2011, San Jose, CA, USA, 28 February–3 March 2011.
24. Apache Xerces Project. Available online: http://xerces.apache.org/ (accessed on 3 July 2020).
25. DOM, Document Object Model. Available online: https://dom.spec.whatwg.org/ (accessed on 3 July 2020).
26. Kam, T.; Villa, T.; Brayton, R.; Sangiovanni-Vincentelli, A. *Synthesis of Finite State Machines: Functional Optimization*; Springer: Boston, MA, USA, 1997.
27. Villa, T.; Kam, T.; Brayton, R.; Sangiovanni-Vincentelli, A. *Synthesis of Finite State Machines: Logic Optimization*; Springer: Boston, MA, USA, 1997.
28. Ziadi, T.; Helouet, L.; Jezequel, J.M. Revisiting statechart synthesis with an algebraic approach. In Proceedings of the 26th International Conference on Software Engineering, Edinburgh, UK, 28 May 2004; pp. 242–251.
29. Harel, D.; Naamad, A. The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.* **1996**, *5*, 293–333. [CrossRef]
30. Cereijo-García, J.; Korhonen, T.; Lee, J.H.; Piso, D.; Osorio, R.R. Timing System at ESS. In Proceedings of IPAC, Copenhagen, Denmark, 14–19 May 2017.