Design of distributed heterogeneous embedded systems in DDFCharts

Ivan Radojevic, Zoran Salcic, Senior Member, IEEE, and Partha Roop

Abstract — The use of formal models of computation in dealing with increasing complexity of embedded systems design is gaining attention. A successful model of computation must be able to handle both control-dominated and data-dominated behaviors, which are most often simultaneously present in complex embedded systems. Besides behavioral heterogeneity, direct support for modeling distributed systems is also desirable, since an increasing number of embedded systems belong to this category. In this paper, we present distributed DDFCharts (DDFCharts), a language based on a formal model that targets distributed heterogeneous embedded systems. Its top hierarchical level is made suitable to capture distributed systems. Behavioral heterogeneity is addressed by composing finite state machines (FSM) and synchronous dataflow graphs (SDFG). We illustrate modeling in DDFCharts with practical examples and describe its implementation on heterogeneous target architecture.

Index Terms—Formal languages, Modeling, Specification languages, heterogeneous systems

I. INTRODUCTION

Design of embedded systems based on formal models of computation (MoC) [1],[2] has been gaining importance due to the increasing gap between technological advances and design productivity. The use of formal models places restrictions on design since certain rules have to be followed. However, it enables automated synthesis and verification with a potential of enormously improving design productivity. In order to be adopted by a wider design community, an MoC must provide enough modeling flexibility and ease of use, as well as a path towards efficient synthesis and verification.

An MoC has to be able to capture behavioral heterogeneity and distributed nature of most modern embedded systems. These two issues make the development of MoCs highly challenging.

According to their behavior, embedded systems can be broadly divided in two groups: control-dominated and data-dominated. Control-dominated systems typically contain large amounts of decision logic that has to quickly produce outputs in response to input events. Some examples of control-dominated systems are lift controllers, vehicle automation and network protocols. Quick reaction to external events is less important in data-dominated systems where intensive computations have to be performed on samples that usually arrive in regular intervals. Good examples of data-dominated behaviors are various digital signal processing algorithms for communications and image processing. Large scale embedded systems are typically a mix of control-dominated and data-dominated parts. Thus, a model used in specification needs to be able to capture both types of behavior.

Besides heterogeneity, increasingly distributed nature of embedded systems must also be addressed. An embedded system may also consist of thousands of physically distributed nodes such as in the case of networks on chip (NoC) [3]. Very often, a node contains both control-dominated and data-dominated behavior. For example, a node in a wireless sensor network must contain data-dominated algorithms for measuring physical quantities as well as control-dominated network protocols.

Obviously, a model of computation that can effectively handle the whole design flow of distributed heterogeneous systems from specification to implementation would be highly beneficial to embedded systems designers. Current models of computation have strengths in certain areas, but they cannot completely handle design of entire heterogeneous distributed systems.

Asynchronous dataflow models have been successfully applied in design of data-dominated embedded systems. In this family of models, Kahn process networks (KPN) [4] is the most general. More restrictive models that have been derived from KPN include synchronous dataflow (SDF) [5], Boolean dataflow (BDF) [6], cyclostatic dataflow (CSDF) [7] and a few others. A Kahn process network consists of concurrent processes that communicate through first-in-first-out (FIFO) channels with blocking reads and non-blocking writes. Because of blocking reads, non-determinism due to different execution orders and relative speeds of processes is eliminated. However, the mandatory use of blocking reads is not suitable for specifying control-dominated systems, where it is often necessary to wait for several events at the same time.

The synchronous/reactive (SR) model [8] is deterministic, but it allows simultaneous waiting on two or more events. This is possible due to the fact that in the SR model all concurrent processes read inputs and produce outputs simultaneously when a tick of the system clock occurs. Also, all
communications and computations are assumed to be instantaneous. The SR model of computation has been used a semantic base for several languages. Some of them, such as Esterel [9] and Argos [10], are suitable for specifying control-dominated systems. Others, such as Lustre [11] and Signal [12] have a dataflow flavor. The use of synchronous languages has been successfully demonstrated in many practical embedded systems. However, a pure synchronous specification of a large scale heterogeneous distributed system with many processes could result in excessive synchronizations leading to inefficient implementations.

Ptolemy [13] targets complex embedded systems by hierarchically combining several models of computation. Although it has code generation capabilities, it is still primarily used as a simulation environment. Process algebra such as CSP [14] and CCS [15] have been useful for analyzing interaction among different models of computation, but they are too abstract to be used for practical specification of large scale distributed heterogeneous systems.

In this paper, we present Distributed DFCharts (DDFCharts), a model of computation with a complete set of features necessary for design of distributed heterogeneous systems including support for control-dominated behaviour, data-dominated behaviour, distributed processes, efficient implementation and verification. DDFCharts has been created by unifying two related models, DFCharts and McCharts. DFCharts [16], [17] addresses heterogeneity in embedded systems by combining synchronously communicating finite state machines with synchronous dataflow graphs (SDFG) [5], but it does not provide mechanisms for modeling distributed systems. McCharts [18] is suitable for design of distributed control dominated systems, but it does not provide any special support for digital signal processing parts. We will use a practical example to demonstrate how DDFCharts can be used for specification of distributed heterogeneous systems. We will also describe a design flow, which enables implementation of DDFCharts specifications on heterogeneous multiprocessor architectures. Previous papers on DFCharts and McCharts contained little material on implementation. Thus, besides the integration of McCharts and DFCharts, the multiprocessor implementation of DDFCharts is also a major contribution of this paper.

The rest of the paper is structured as follows. Section 2 introduces the main concepts for modeling in DDFCharts. Section 3 describes the DDFCharts approach in more detail using a practical example. Section 4 presents some important details of the formal semantics of DDFCharts, which is based around multiclock finite state machines (MCFSM), initially introduced in [18]. Section 5 and 6 cover the implementation of DDFCharts by describing allocation, partitioning and synthesis from specifications. Section 7 presents implementation results for an example presented in section 3. Section 8 discusses related work by making direct comparisons between DDFCharts and other approaches. Finally, section 9 concludes the paper.

II. INTRODUCTION TO DDFCHARTS

In DDFCharts, finite state machines (FSM) and synchronous dataflow graphs (SDFG) [5] are combined using five operators: symmetric asynchronous parallel (SAP), asymmetric asynchronous parallel (AAP), synchronous parallel (SP), refinement (R) and hiding (H). In section 4, where the formal semantics is discussed, the operators will also be labeled as //, \, ||, R, \ respectively. Figure 1 gives the first illustration of a specification in DDFCharts. The specification for simplicity does not contain all details of signal labels. Full semantics will be discussed in the next section with real-world examples. At the top level, a DDFCharts specification represents a network of nodes, which contain compositions of FSMs and SDFGs. FSMs from different nodes are connected using the SAP operator. The other operators are used inside a node. Inside a node, FSMs may be driven by different clocks.

A clock represents a sequence of reactions in a similar way as in synchronous languages. At the specification level, each tick is assumed to be instantaneous. When the system is implemented each tick has a finite duration. Since DDFCharts specifications are primarily intended for software implementation on multiprocessor architectures, DDFCharts clocks should not be thought of as hardware clocks. Thus, clock ticks are not necessarily equal in duration at the implementation level.

Each node has a special clock called master clock or global clock (gclk). FSMs that are driven by the same clock can be connected with the SP operator. They execute in lockstep and may communicate via local signals that are handled using the hiding operator in a similar way as in Argos [10]. A state of an FSM can be refined using the refinement (R) operator. If the refined FSM is driven by the master clock the refining FSM
can be driven by any clock. However, if the refined FSM is not driven by the master clock, the refining FSM must be driven by the same clock. For example, FSM4 could only be refined by an FSM driven by \texttt{clk1}. An FSM that is driven by the master clock can be connected with an FSM driven by another clock using the AAP operator. In that case communication is performed through rendezvous channels in a way that resembles communication in process algebra, in particular CCS [15]. The same type of communication is applied with the SAP operator. An FSM driven by the master clock can also be connected with an SDFG using the AAP operator.

Unlike a globally asynchronous locally synchronous (GALS) node, a DDFCharts node is not synchronous in general since it can contain multiple unrelated clocks driving FSMs. In addition, SDFGs operate at their own speeds and their execution can be modeled by separate logical clocks as will be discussed in section 4. A clock domain is a set of FSMs that are driven by the same clock. The distinguished clock domain is represented by master clock. Master clock driven FSMs can activate and terminate other clock domains, hence the name master or global clock. These “slave” clock domains can be used to model unrelated tasks that are often data-intensive and time-consuming. Thus, slave clock domains are usually slower than the master clock domain. This situation can be found in many large embedded systems. For example, in a mobile phone, channel coding and a video game are examples of processes that can be modeled as slave clock domains in DDFCharts. The master clock domain would control these processes, but it would also handle user interface, network protocols and other activities where control logic with quick reactions is important.

III. Modeling in DDFCharts

We will describe modeling in DDFCharts using the system called frequency relay as an example. Power systems need protection from overloading. When a power system is overloaded some loads must be disconnected in order to prevent damage. A significant decrease in the frequency level of the main AC signals whose normal value is 50 (or 60) Hz indicates that the system may be dangerously overloaded. The same problem is also detected when the rate of change of the AC signal is too fast. The frequency relay is a sensor that measures the frequency and its rate of change in a power network. Measurement results are compared against a set of thresholds that can be modified during the system operation. If the current thresholds indicate that the frequency is too low or its rate of change too fast some loads are disconnected from the network by opening one or more switches as determined by a decision algorithm. The decision algorithm also uses a timing parameter that is received through a CDMA link. It should be noted that the frequency relay case study was also used in [17], but here it is extended with the CDMA link. While increasing the complexity of the system, this change has also made the system distributed. Hence, DDFCharts becomes a suitable formalism.

The general form of an FSM transition in DDFCharts is \( t[c]/O,P. \) \( t \) is the transition trigger which represents a Boolean expression on signals; \( c \) is the condition on variables; \( O \) is the set of emitted output signals; \( P \) is the set of invoked procedures. Procedures are sequential and are written in an imperative language. It is important to emphasize that procedures must be executed in a finite amount of time because the duration of each tick has to be bounded when the system is implemented. Thus, infinite loops inside procedures are not allowed. Local variables can be defined inside procedures, but they cannot retain values between procedure calls. Thus, all results have to be written to variables that are defined inside FSMs.

Figure 2 shows the top level, which consists of two nodes. All FSMs in N1 are driven by the master clock \texttt{gclk1} while all FSMs in N2 are driven by the master clock \texttt{gclk2}. The main operation of the system is contained inside N1 (Figure 3). N2 is used to receive the timing parameter using a CDMA link, which is passed through channel \texttt{on} to N1 where it is stored in variable \texttt{tp} (timing parameter).

The operation of SAP operator can be explained using the FSMs at the top level. FSM1 from node N1 and FSM8 (Figure 4) from node N2 are connected with the SAP operator. They communicate through channel \texttt{on}. A channel status signal (CS signal) is associated with every rendezvous channel. When communication is happening on a rendezvous channel, the corresponding CS signal is true. Otherwise, it is false. A CS signal can trigger a transition in an FSM just like any other signal. Communication on a rendezvous channel occurs when both sides are ready. It is instantaneous. This is exactly the reason why a CS signal can be easily associated with communication through a rendezvous channel. If communication through a rendezvous channel was not atomic i.e. it takes multiple clock ticks instead of a single clock tick, it would be more difficult to define how channel communication can trigger transitions in FSMs.

An FSM is ready to communicate through a channel when it enters the rendezvous state corresponding to the channel. Only one channel can be associated with a rendezvous state. A rendezvous state must have an outgoing transition that is triggered by the corresponding CS signal. In this example, the rendezvous states for channel \texttt{on} are \texttt{S11} in FSM1 and \texttt{S81} in FSM8. When a rendezvous occurs on channel \texttt{on}, FSM1 makes the transition from \texttt{S11} to \texttt{S12}. In this transition, it receives an integer value from the channel and stores it into variable \texttt{tp}, and completes procedure \texttt{init_thresh()}, in which thresholds (thr1 to thr6) are initialized. On the other side of the channel, FSM8 makes the transition from \texttt{S81} to \texttt{S82}, in which it sends an integer value from variable \texttt{tp} to the channel. Both transitions are triggered by CS signal \texttt{on}. We use the CSP [14] notation where “!” denotes output while “?” denotes input. When a rendezvous on channel \texttt{on} happens, \texttt{gclk1} and \texttt{gclk2} are synchronized i.e. their ticks
coincide. Clock synchronization will be discussed in detail in implementation sections.

It should be noted that the symmetric asynchronous parallel operator only allows FSMs that are driven by master clocks to communicate through channels. This essentially means that communication between different DDFCharts nodes can only be achieved through master clocks. This restriction does not significantly reduce the modeling power of DDFCharts but it does make implementation a lot easier. If slave clocks from different nodes were allowed to be directly connected by the SAP operator, it would be much more difficult to implement rendezvous communication since more than two clocks would have to be handled. Two slave clocks would need to be synchronized, but in addition their master clocks would have to be involved, because master clocks can preempt slave clocks that need to synchronize. The definition of the AAP operator, which will be covered in more detail later in the section, also allows only two clocks (master and slave) to be involved in a rendezvous.

When FSM1 enters state S12, the main operation in node N1 begins. S12 is refined by five FSMs and one SDFG. They will be used for explaining the semantics of the SP operator and the semantics of the AAP operator when it connects an FSM and SDFG.

The synchronous dataflow graphs (SDFG) in DDFCharts follow the rules described in [5]. An SDFG consist of actors that are connected by first-in-first out (FIFO) buffers. The sizes of those buffers are finite and are determined at compile time. An important property of an SDFG is its iteration. An iteration is a series of actor firings that return FIFO buffers to their original state. Each actor has a firing rule, which specifies how many tokens are consumed from each input buffer and how many tokens are produced in each output buffer in a single firing.

SDF1 (shown in Figure 5) consists of three actors that communicate through single-place FIFO buffers. Its iteration is carried out by firing averaging filter, symmetry function and peak detection once. The numbers next to actors indicate their firing rules. In this case, each actor consumes one token and produces one token when it is fired. The purpose of SDF1 is to detect peaks in the AC waveform following the algorithm from [19]. It sends the time between each two consecutive peaks to FSM3 (Figure 6) through channel fpout.
variables.

<table>
<thead>
<tr>
<th>FSM4 (gclk1)</th>
<th>FSM3 (gclk1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable: float ave_freq=0; int fs=0, rs=0;</td>
<td></td>
</tr>
<tr>
<td>S41</td>
<td>S31</td>
</tr>
<tr>
<td>start_roc/aroc_calc()</td>
<td>[din ==0]</td>
</tr>
<tr>
<td>true/rd</td>
<td>S32</td>
</tr>
<tr>
<td></td>
<td>[din !=0] laf_calc()</td>
</tr>
<tr>
<td>S43</td>
<td>S34</td>
</tr>
<tr>
<td>true/rs_calc()</td>
<td>true/fs_calc()start_roc</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

signal: start_roc, rd;

Figure 6: Frequency and rate of change calculations

Given the information from fpout, FSM3 calculates the frequency, while FSM4 (Figure 6) calculates the rate of change. FSM3 and FSM4 also compare the frequency and rate of change against the thresholds (defined in Figure 3) and store the result in variables fs and rs (frequency status and rate of change status). FSM3 and FSM4 are connected by the SP operator. Thus, they must have the same clock, which is in this case gclk1. FSMs that are connected by the SP operator can communicate with local signals and shared variables. The only local signal used by FSM3 and FSM4 is start_roc (start rate of change calculation). When FSM3 takes the transition from S33 to S34 it emits start_roc. In the same clock tick, start_roc triggers the transition from S41 to S42 in FSM4. The variable that is shared between FSM3 and FSM4 is ave_freq (average frequency). FSM3 is the writer while FSM4 is the reader. FSM3 writes ave_freq in procedure af_calc(). Among concurrently running FSMs only one FSM is allowed to write a shared variable. The new value becomes available to the readers in the next tick. On the other hand, local variables that are used by single FSMs can be written multiple times in procedures within the same tick. When a local variable is written, the value takes effect immediately, instead of waiting for the next tick to occur as in the case of shared variables.

<table>
<thead>
<tr>
<th>FSM9 (gclk2)</th>
<th>FSM10 (clk2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S91</td>
<td>S101</td>
</tr>
<tr>
<td>ch1?tp/update</td>
<td>ch2!</td>
</tr>
<tr>
<td>S93</td>
<td>S102</td>
</tr>
<tr>
<td></td>
<td>tp_res 2 tp_ena</td>
</tr>
</tbody>
</table>

Figure 7: FSM9 and FSM10

Figure 8. Decoding in CDMA receiver

N2 consists of three FSMs and one SDFG. The top level FSM is shown in Figure 4. In state S81, FSM8 sends the current value of tp to N1 and then goes to S82 where it obtains a new value for tp. S82 is refined by two FSMs and one SDFG as shown in Figure 4. FSM9 (Figure 7) is driven by the master clock of node N2. It receives the value of the timing parameter from SDF2 which represents a CDMA receiver. Beforehand, it has to complete a rendezvous with FSM10 (Figure 7) that is driven by clk2. FSM10 represents a simple user interface, which enables the reception of the timing parameter. The rendezvous between two FSMs connected by the AAP operator is the same as that of the SAP operator. Both FSMs must be in rendezvous states for the rendezvous to happen. In this case, the rendezvous states are S91 for FSM9 and S102 for FSM10. Figure 7 shows that state S102 can be left before the rendezvous occurs if signals tp_res (transition parameter reset) is present. However, if both transitions are enabled in the same clock tick, the rendezvous transition (ch2!) has priority, as indicated by the priority labels. Transition priorities can be freely assigned in DDFCharts except in rendezvous states, where the rendezvous transition always must have the highest priority.

After FSM9 has completed the rendezvous with FSM10, it has to complete another rendezvous with SDF2 to receive the value of tp, which will then be used by FSM8. The receiver (SDF2) decodes an incoming packet of 384 bits to produce a message of 80 information bits, which are then converted into a single integer value. Note that this is only a partial model for the receiver since demodulation and RF stages are not shown.

FSM8, which is driven by the master clock, could further be refined by both FSMs that are driven by gclk2 and those that are driven by other clocks. On the other hand, FSM10 can only be refined by FSMs that use its clock i.e. clk2. With this refinement rule, all clocks could easily be related to the master clock, so that the whole node can be made synchronous if this is beneficial for design steps after specification.

Different clocks in DDFCharts specifications are unrelated initially. However, it is possible for the designer to enforce a relation between a master and slave clock in order to reduce verification effort. This issue will be discussed in detail in section 6. For example, it can be specified that every tick of clock clk2 takes four ticks of gclk2 in Figure 7 using the label SDF10(clk2=4*gclk2). This is also possible with SDFG iterations. For example, it can be specified that every iteration of SDF1 takes five ticks of gclk1 using the label SDF1(5*gclk1). Allowing such property does not change the semantics of DDFCharts operators presented in this section. It
only reduces the set of possible behaviors. Communication between related clocks would still be performed from rendezvous states, but in this case it becomes predictable when it will take place.

IV. FORMAL SEMANTICS

The formal semantics of DDFCharts represents an extension of the McCharts semantics, which was presented in [18] and draws from synchronous language Argos [10]. In Argos, two or more FSMs are combined by the operators to produce an equivalent FSM. The same is done by McCharts operators, but instead of single clock FSMs, multiclock FSMs (MCFSM) are used. DDFCharts uses the same set of operators as McCharts, SAP (\), AAP (\), SP (||), refinement (\), and hiding (\). However, it extends that model with the use of variables and its AAP operator works with SDFGs. Therefore, the main purpose of this section is to demonstrate MCFSM with variables and how SDFGs are incorporated into the MCFSM-based operators.

We present a simple multiclock FSM in Figure 9 before the formal definition. In a multiclock FSM, transitions can be driven by different clocks. The FSM in Figure 9 has two clocks, k, which is its master clock, and k1. At the beginning of each transition label, it is indicated which clock drives it. Synchronization of clocks is indicated by using brackets in the clock label. Inputs are tied to specific clocks. At each tick of k, input a is read. On the other hand, input b is tied to clock k1. In states S1, S2 and S3, where k and k1 are not synchronized, their transitions are simply interleaved. a and b are read separately. Depending on implementation, ticks of asynchronous clocks may happen to coincide. In states S1, S2 and S3, if ticks of clocks k and k1 happen to occur at the same time and their transitions are enabled simultaneously, the outgoing transition is selected randomly. In state S4, where k and k1 are synchronized, a and b are read at the same time. Clock synchronization is denoted as \( <k(k1)> \). In this notation, the clock outside the parentheses has to be a master clock. Other clocks are placed in the parenthesis. In each state there is a transition for every possible input combination. A dot between inputs means logical AND while a bar over an input means NOT.

**Definition 1:** MCFSM with variables

\[ A = (CLK, Q, q_0, I, R, O, V, C, T, RQ, Proc) \]

\( CLK \) is the set of clocks that drive MCFSM transitions. \( Q \) is set of states where \( q_0 \) is the initial state. \( I \) is the set of input signals. Each input signal can either be present (true) or absent (false). \( R \) is the set of channel status signals (CS signals). In DDFCharts semantics, CS signals are tested in the same way as input signals. When a clock tick occurs, a CS signal is true if a rendezvous is happening on the corresponding channel and false otherwise. However, while input signals are tested in all states, CS signals are only tested in rendezvous states. \( O \) is the set of output signals. \( V \) is the set of variables. \( C \) is the set of conditions on variables. Like input signals, CS signals and output signals, conditions are Boolean; they can be true or false. \( T \) is the set of transitions. \( RQ \) is the rendezvous mapping function, which maps channels to states. \( Proc \) is the set of procedures on variables.

**Definition 2:** MCFSM transitions

\[ T \subseteq CLK \times 2^{CLK} \times O \times B(I) \times B(C') \times B(R') \times 2^O \times Proc \times Q \]

A transition is driven by a clock \( clk \) taken from the set \( CLK \). In addition, there could be a set of clocks taken from \( 2^{CLK} \) (power set of \( CLK \)) that have to synchronize with \( clk \) when the transition occurs. \( B(I') \) where \( I' \subseteq I \) is the set of Boolean formulas on input signals that are bound to \( clk \) and clocks synchronized with it. Each formula is a complete monomial, or in other words a conjunction that has to contain every input signal in \( I' \). \( B(C') \), where \( C' \subseteq C \), is the set of Boolean formulas on variable conditions that are bound to \( clk \) and clocks synchronized with it. Each formula is a complete monomial. \( B(R') \) where \( R' \subseteq R \) is the set of Boolean formulas (again complete monomials) on CS signals that are linked to the source state of the transition and bound to \( clk \) and clocks synchronized with it.

We can write a transition as a tuple \((clk,q,i,o,p,q')\). \( i = mcr \) where \( m, c \) and \( r \) are monomials over input signals, conditions on variables and CS signals respectively. The dot between monomials denotes the AND operation exactly in the same way as inside monomials. \( o \) and \( p \) denote output signals and procedures.

Variables in DDFCharts are unbounded i.e. each variable can take an infinite range of values. If each value was represented explicitly, any formal analysis would be difficult. Hence, it is necessary to abstract data in some way. In DDFCharts, this is done by representing each condition on variable as a signal. It only matters whether it is true or false.
A similar approach for data abstraction is used in Esterel Studio [9] and Polis [20]. Data abstraction is illustrated in Figure 10 where the specification (Figure 10 (a)) has one input signal and two conditions on variables. In Figure 10 (b), b and c stand for conditions \([v>4]\) and \([v<6]\), while p stands for procedure \(\{v=v+1\}\). In Figure 10(a), when both transitions are enabled in either state, transition priorities decide which transition will be taken. For example, when the FSM is in state S1, signal a is true and variable v is equal to 5, transition \(a[v<6]\) is taken.

Treating conditions on variables as signals in the formal semantics of DDFCharts does not mean that variables themselves are just a syntactic feature. Without the existence of semantics of DDFCharts does not mean that variables a[v<6] is taken. S1, signal a is true and variable v is equal to 5, transition \(a[v<6]\) is taken.

In Figure 10 where the specification (Figure 10 (a)) has one input signal and two conditions on variables. In Figure 10(a), when both transitions are enabled in either state, transition priorities decide which transition will be taken. For example, when the FSM is in state S1, signal a is true and variable v is equal to 5, transition \(a[v<6]\) is taken.

Due to limited space, we will not present the formal definitions for the DDFCharts operators that were presented in [18]. The key difference is in that MCFSMs in DDFCharts contain variables, while those in McCharts do not. We will present however, the way in which SDFGs can be incorporated into the MCFSM semantics.

**Definition 3**: SDFG with \(m\) inputs and \(n\) outputs

\[ SDF = \top \downarrow^{io} \text{IOPAR} \]

\[ \top = (\{\text{sdclck}\},Q,q_0,I,R,O,V,C,T,RQ,Proc) \]

\[ Q = \{io,\text{processing}\}, q_0 = io. \]

\[ I = \{gs\} \cup \{in_qi,1 \leq i \leq m\} \cup \{in_qo,1 \leq i \leq n\}, \quad R = \emptyset, \quad O = \emptyset, \quad V = \{data_{in},1 \leq i \leq m\} \cup \{data_{out},1 \leq i \leq n\}, \quad C = \emptyset, \quad T = \{t_1,t_2\}, \quad RQ = \emptyset. \quad \text{Proc} = \{\text{assign}_{\text{output}}_{\text{values}}\} \]

\[ t_1 = (\text{sdclck},\emptyset,io,in_{q_i},in_{q_i},...in_{q_i},in_{q_o},in_{q_o},...in_{q_o}, \quad \text{processing}) \]

\[ t_2 = (\text{sdclck},\emptyset,processing gs,\text{-},\emptyset,\{assign_{output}_{values}\},io) \]

\[ \text{IOPAR} = IN_1 \iff IN_2 \iff ... \iff IN_n \iff OUT_1 \iff OUT_2 \iff ... \iff OUT_n \]

\[ IN = \{\text{sdclck}\},Q,q_0,I,R,O,V,C,T,RQ,Proc \]

\[ Q = \{q_1,q_2\}, \quad q_0 = q_1, \quad I = \emptyset, \quad R = \{gd\}, \quad O = \emptyset, \quad V = \{data_{in}\}, \quad T = \{t_1\}, \quad RQ = \{(q_1,\text{gd})\}, \quad \text{Proc} = \{\text{gd}_{\text{proc}}\} \]

\[ t_1 = (\text{sdclck},\emptyset,q_1,\text{-},\text{-},gd,\emptyset, \text{gd}_{\text{proc}}.q_2) \]

\[ OUT = \{\text{sdclck}\},Q,q_0,I,R,O,V,C,T,RQ,Proc \]

\[ Q = \{q_1,q_2\}, \quad q_0 = q_1, \quad I = \emptyset, \quad R = \{pd\}, \quad O = \emptyset, \quad V = \{data_{out}\}, \quad T = \{t_1\}, \quad RQ = \{(q_1,\text{pd})\}, \quad \text{Proc} = \{\text{pd}_{\text{proc}}\} \]

\[ t_1 = (\text{sdclck},\emptyset,q_1,\text{-},\text{-},pd,\emptyset, \text{pd}_{\text{proc}}.q_2) \]

Note that in the definition of \(IN\) and \(OUT\), \(gd\) (get data) and \(pd\) (put data) appear as CS signals. For communication with the external environment they would appear as input signals. It should also be noted that the subscript of the down-arrow in the refinement operator denotes the state that is refined. In definition 3, FSM IOPAR refines state io.

Definition 3 can be converted into a single “SDF-FSM” that can be used as the right operand of the AAP operator. Besides SDF, any dataflow model that has clearly defined iterations.

![Figure 10: Data abstraction in DDFCharts](image)

![Figure 11: Operation of SDFG with two inputs and two outputs](image)
can be easily incorporated in the DDFCharts semantics. An iteration is a convenient point for rendezvous with gelk-driven FSMs. Kahn process networks [3] and other dynamic dataflow models where iterations have no meaning would present more challenge.

V. ALLOCATION AND PARTITIONING

DDFCharts can be used as an underlying model for a top-down design flow that includes specification, allocation and partitioning, and synthesis. Specification in DDFCharts was discussed in detail in section 3. In this section, we will adopt a simple solution for allocation and partitioning and then concentrate on synthesis in the next section where the strengths of DDFCharts model are exploited and demonstrated.

DDFCharts does not require any special architecture for implementation. However, it is desirable to match the behavioral heterogeneity in DDFCharts with a correspondingly heterogeneous architecture in which some processing elements would be specialized for executing FSMs while others would be specialized for executing SDFGs. It is not difficult to justify this division of tasks considering vastly different requirements for execution of FSMs and SDFGs. In DDFCharts, FSMs are mainly used for modeling control-dominated processes that include minor computations. Thus, a processor executing FSMs does not need to have powerful computational units but it has to provide fast interaction with the environment so that it can promptly react to external events that can arrive at any time. On the other hand, for the execution of SDFGs a processor should contain a multiply and accumulate (MAC) unit, shifters, multiple buses and other features that support digital signal processing, or even be a dedicated application-specific processor.

Following the principles outlined above, we have created an architecture consisting of two types of processors: reactive processor, in this case ReMIC [21], and conventional microprocessor that can be further customized, in this case Altera Nios II [22]. ReMIC is used for execution of FSMs as it has special features for supporting reactivity, which were motivated by the synchronous language Esterel. Certain instructions allow quick communication with input and output control signals. For example, it can be tested if an input signal is set with a single instruction called present. Similarly, an output signal can be set with a single instruction called emit. Nios II is used for execution of SDFGs. It is a soft core processor of RISC type with a straightforward way to enhance (customize) its DSP capability by attaching additional execution units such as multiply and accumulate (MAC) unit.

In the proposed architecture which is a variant of HiDRA architecture [23], ReMIC and Nios II communicate through shared memory. In addition, control I/O signals of ReMIC are directly connected to the parallel I/O (PIO) port of Nios II. In a distributed system, a separate ReMIC/Nios II pair should be used for each DDFCharts node. Communication between ReMICs implementing different nodes is achieved through shared memory as it is assumed in the rest of the paper. Figure 12 shows how the frequency relay example would be mapped on the architecture. Of course, it is also possible to map multiple DDFCharts nodes on a single ReMIC/Nios II pair. This option does not significantly increase the complexity of implementation since the only significant difference would be in the implementation of the SAP operator, which would have to be contained entirely within a single processor.

![Figure 12: Mapping frequency relay example on heterogeneous architecture](image)

Each SDFG is executed on a single processor in the architecture we described above. We adopted this simple solution so that we could concentrate on the implementation of the DDFCharts operators. It should be satisfactory for most applications. In some cases though, it may be beneficial to distribute the execution of an SDFG across multiple processing units. Plenty of work has already been done in this area. For example, an early paper on SDF model of computation [5] describes in detail how a single SDFG can be partitioned across multiple processors. Even techniques for HW/SW co-design based on a single SDFG have been developed [24]. We could easily extend our proposed architecture to adopt some of these approaches.

VI. SYNTHESIS

When a DDFCharts node is synthesized for execution on a ReMIC/Nios II pair, several low-level data structures and programs emerge: FSM threads (FT) implementing FSMs, SDF threads (ST) implementing SDFGs, communication mechanisms used by the operators (synchronous signals, shared variables, rendezvous channels, and buffers), FSM scheduler, SDF scheduler and tick handler. They are described in the following subsections.

**FSM and SDF threads**

Each FSM is synthesized into an FSM thread (FT). An FT implements all states and procedures that appear in the corresponding FSM. If an FSM can be preempted i.e. it refines another FSM, the FT will also contain an additional state which is entered upon preemption. Each SDFG is synthesized into an SDF thread (ST). An ST simply consists of all the functions that are called during the execution of the corresponding SDFG. It operates in two phases according to definition 3. In the processing phase, all SDF outputs are computed. Subsequently, outputs are sent and new inputs are received through rendezvous channels in the I/O phase.
Operators

The communication mechanisms that are necessary for implementation of the synchronous parallel operator are synchronous signals and shared variables. Synchronous signals are used for communication between FSMs that are driven by the same clock. In a DDFCharts specification, those signals are hidden with the hiding operator. It only takes a single bit in memory to implement the status of a signal, which can be present or absent. However, an additional bit is needed to indicate whether the signal has been resolved in the current clock tick. The status of a signal cannot be read unless the resolution bit is set, which ensures that a signal cannot be read before it has been written.

The implementation of a shared variable requires twice as much memory as the actual variable size, since two values must be held, the current value and the next value. When a shared variable is written, the data is placed in the memory location holding the next value. At the end of the clock tick, the tick handler copies the data to the memory location holding the current value. When an FSM reads a shared variable, it always gets the data from the current value memory location.

Besides local signals and shared variables that are visible in a specification, the implementation of the hierarchical operator also requires signals that handle activation and termination of FTs and STs. Every hierarchical state is associated with activation and termination signals, which are implemented with status and resolution bits just like specification-visible local signals. When an FT enters a hierarchical state it sets the corresponding activation signal. When it leaves the hierarchical state it sets the corresponding termination signal. The FTs and STs that refine the state must read these signals.

The implementation of both asynchronous parallel operators requires the construction of rendezvous channels. A rendezvous channel can connect an FSM and an SDFG or two FSMs that belong to different clock domains. This communication mechanism requires more implementation effort than others, especially in the case when two FSMs are involved. However, it should be noted that rendezvous in DDFCharts is significantly easier to implement than rendezvous in languages that allow strong preemption such as CRP [25]. As in Argos, only weak preemption is available in DDFCharts. Strong preemption would make synchronization during rendezvous a lot more difficult.

The implementation of rendezvous between two FSMs involves a few steps on both the sending and receiving side. The protocol for the sender is illustrated in Figure 13 while the protocol for the receiver is illustrated in Figure 14. When the sending FSM is in the rendezvous state, it reads a flag (RCV_RND) which indicates whether the receiving FSM is in the rendezvous state as well. If it is not, the sending FSM can stay in the rendezvous state or it can leave it without completing communication with the receiving FSM. For example, in Figure 7 FSM10 would leave state S102 if event tp_res was present. If the receiving FSM is in the rendezvous state, the rendezvous will be completed in the current tick. The sending FSM places data in a buffer and sets a flag (DATA_IN) to indicate this operation.

When the receiving FSM is in the rendezvous state, the receiving FSM reads data from the buffer and sets a flag (RCV_ACK) and then takes the transition out of the rendezvous state. It then waits for the acknowledgement (RCV_ACK) from the receiving FSM. Once it receives the acknowledgement, it can take the transition out of the rendezvous state.

From the sender’s protocol, the receiver’s protocol becomes obvious. If the receiving FSM is in the rendezvous state and a flag (SND_RND) shows that the sending FSM is not, the receiving FSM may leave the state without completing communication through the channel. Otherwise, it waits for the sending FSM to place the data in the buffer. When it reads the data it sets the acknowledgement flag (RCV_ACK) and then takes the transition out of the rendezvous state.

It was briefly mentioned in section III that transition priorities can be freely assigned except in a rendezvous state where the rendezvous transition must have the highest priority. This has important implications for the implementation of rendezvous communication. When both the sending and receiving FSMs enter their rendezvous states, communication over the rendezvous channel has to happen. At that point, whether non-rendezvous transitions are enabled or not becomes irrelevant.

It should be noted that when two FSMs completing rendezvous are executed on different processors, simultaneous
access to various flags that are stored in shared memory must be forbidden. Otherwise, the protocol could fail.

The protocol for rendezvous between an FSM and an SDFG is very similar to the one just described. When an SDFG reaches the end of an iteration, it enters its 'rendezvous state'. Unlike an FSM, an SDFG cannot leave this state until the communication is completed.

The rendezvous protocol has interesting implications for the possibility of pipelined execution of SDFGs. As described in section III, an SDFG normally consumes inputs that are needed for a single iteration. However, it is also possible for an SDFG to consume inputs for multiple iterations, which would not be in conflict with the formal semantics of DDFCharts. This type of modeling would be useful for example, when inputs arrive at irregular intervals and have to be buffered. When multiple inputs are passed into and SDFG, they can be processed in a pipeline formed from the internal SDFG blocks. The results are then passed to the output channels, but only when all data has been processed. Therefore, pipelined execution of SDFGs in DDFCharts is possible, but it is constrained to a certain extent due to the fact that intermediate results are not available immediately. Allowing this option would require a considerable change in the semantics of DDFCharts.

Clocks, scheduling and synchronization

The tick of a clock is completed when each FSM belonging to the clock domain has made a transition. Obviously, the duration of a tick will depend on how much computation has to be done for each transition. Transitions with lengthy procedures will take longer to execute. Different transitions will have different execution times. Thus, tick durations will be variable in general. When FSMs are distributed on multiple processors clock ticks will be executed faster. However, the amount of acceleration may be limited by signal dependencies between transitions. The duration of a tick may also be affected by a rendezvous with a different clock domain. When two clocks are involved in a rendezvous, the next tick of one clock cannot begin before the current tick of the other one has been completed. Figure 15 shows an example of two clocks that synchronize at one point.

![Figure 15: Typical clock ticks](image)

Constraints on maximum duration of clock ticks are imposed by the environment. If at least one of those constraints is not satisfied, the designer has to partition FSMs and SDFGs in a different way, modify the architecture, or return to the specification and make changes on that level.

For each clock domain, the FSM scheduler contains the order in which the FSMs are executed. When a particular schedule is run, it may be necessary to visit an FSM more than once before it completes its transition. For example, when visited by the scheduler, an FSM may not be able to complete its transition because it cannot read a local signal that has not yet been resolved. Or, the FSM may not be able to complete a transition out of a rendezvous state because it is waiting for the acknowledgement from the receiving FSM. In such case, the FSM scheduler would leave the FT and then revisit it in the next run.

Efficient static scheduling is probably the strongest advantage of using synchronous dataflow graphs. The SDF scheduler is constructed as described in [5].

The tick handler performs various operations at the end of a tick for each clock domain. Some of those operations are updating shared variables, writing outputs and clearing resolution bits of local signals. In addition, the tick handler is responsible for synchronizing clock domains if requested by the designer. As mentioned earlier, clocks in DDFCharts are asynchronous and the speed of SDFGs is not controlled, but the designer may choose to set the relation between a master clock and another clock. For example, it can be specified that $\text{clk1} = 5\times\text{gclk1}$ which means that each tick of clk1 takes five ticks of gclk1. Furthermore, the relative speed of an SDFG can be specified in terms of the master clock. For example, $\text{sdff} = 4\times\text{gclk1}$ means that each iteration of sdfg takes four ticks of the master clock. The possibility of synchronizing initially unrelated clocks is a key feature in the implementation of the DDFCharts model, which needs to be discussed thoroughly.

When slave clocks and SDFGs are not related to the master clock, a DDFCharts node may produce different output sequences given a single input sequence, depending on relative speeds of clocks. From this point of view, a DDFCharts node is generally non-deterministic. The designer may choose to enforce determinism by setting the length of slave clock ticks and SDFGs in terms of the number of ticks of the master clock. In this way, verification effort could be drastically reduced since a single set of inputs cannot produce multiple behaviors. This is especially important for safety-critical applications where incorrect system operation can be disastrous. On the other hand, having a completely synchronous system could significantly degrade the system performance due to excessive synchronizations. This is especially harmful for applications where fast execution and low memory consumption are the most important requirements. In particular, forcing control-dominated parts of the system (modeled with FSMs) and data-dominated parts of the system (modeled with SDFGs) to operate at related speeds should be carefully considered. This issue will be discussed further in section 7 when implementation of node N1 in the frequency relay example is presented.

Completely unrelated clock domains and completely related clock domains are two extremes. In many cases, the best choice could involve having both related and unrelated clock domains. By supporting the trade-off between verification effort and implementation efficiency the
DDFCharts approach aims to support a wide range of applications.

VII. FREQUENCY RELAY RESULTS

To illustrate implementation of DDFCharts specifications we present results for node N1 from the frequency relay example. The implementation architecture consisting of a ReMIC/Nios II pair is shown in Figure 16. It was implemented on EP1S60F1020C5, an FPGA from the Altera Stratix family. The number of logic elements (LE) consumed by the architecture is 2677. The program memory for Nios II requires around 25KB while the program memory for ReMIC requires around 4KB. The frequency of the Nios II clock is 110 MHz. ReMIC has a slower clock running at 61 MHz.

The execution time of ST iterations running on the Nios II processor is just below 90 μsec and it has very little variation. This is not surprising considering that the ST code has very little branching logic. The execution on the ReMIC processor is quite different. The time taken to execute a tick of the master clock varies widely depending on the current states and the destination states of the seven FSMs in N1. Table 1 shows some examples of state transitions and execution times for four possible ticks (n, m, p q) for seven FSMs. The last column shows the worst case execution time in terms of ReMIC clock cycles. The longest tick in the system is given in row 2 (Tick m). In this tick, FSM7 is inactive (denoted by x) while FSM1, FSM2, FSM3, FSM4, FSM5 and FSM6 take the transitions from states S11, S23, S33, S41, S51 and S61 to states S12, S21, S34, S42, S51 and S61, respectively. By multiplying 1302 with the ReMIC clock period we find the real time taken is 21.3 μsec.

Figure 16: Architecture for node N1 in frequency relay

The relative speed of SDF1 does not have a significant impact on the behavior of node N1. Thus, the option of forcing each iteration of SDF1 to take certain number of gclk ticks was not selected in the final implementation. However, it is still important to consider performance implications that would arise if this option was selected. When controlling the length of SDF iterations, the most important question is how many ticks of the master clock should each iteration take. The only valid approach in finding the answer is to work with the longest tick in the system. What has to be found out is how many of those ticks fit within the time taken by each SDFG iteration. It was mentioned earlier that Nios II completes each SDFG iteration in slightly less than 90 μsec. However, it cannot start the next iteration until a new sample arrives, which happens every 125 μsec (since the sampling frequency is 8 KHz). Thus each iteration effectively takes 125 μsec. Obviously, the longest tick has to be stretched from 21.3 to 25 μsec so that each iteration consumes five ticks. This is an increase of nearly 20%. It is not difficult to imagine that the adjustment could be greater if there were multiple SDFGs in the node. Moreover, event-driven execution is no longer possible. ReMIC now has to read inputs periodically every 25 μsec. This may not be the preferred option, considering that its inputs arrive asynchronously from the environment.

VIII. RELATED WORK

DDFCharts overlaps to some extent with Ptolemy, synchronous languages, globally asynchronous locally synchronous (GALS) languages and process algebras. While these models of computation and languages have been used successfully for certain types of embedded systems, they lack a complete set of features necessary for design of distributed heterogeneous systems. DDFCharts aims to fill this gap. In the following paragraphs we will discuss each group of models mentioned above and compare it against DDFCharts.

With the presence of SDF graphs, DDFCharts resembles Ptolemy [13], where SDF is the most developed domain. In particular, DDFCharts is similar to a subset of Ptolemy called *charts [26], which combines finite state machines with a few other models of computation. Ptolemy has a wider scope than DDFCharts since it includes models of computation that are suitable for modeling analog systems. In terms of modeling, the biggest difference is that Ptolemy combines models of computation only hierarchically while DDFCharts also allows parallel compositions. In Ptolemy models, at each hierarchical level blocks have to obey the semantics of a single model of computation, but internally each block can be refined into a system that behaves according to some other model. The advantage of purely hierarchical heterogeneity is a strong emphasis on modularity. On the other hand, it may be difficult to devise a meaningful communication mechanism between the outer and inner models, which could make specification more difficult and reduce implementation options. While Ptolemy has some code generation capabilities, a complete design flow which maps heterogeneous specifications onto a multiprocessor architecture has still not been developed. The inner model may lose some properties while adjusting to the outer model. This issue becomes apparent when the frequency relay is specified in Ptolemy. Figure 17 shows a possible solution for N1. The top level MoC is FSM. State S2 is refined by a network of blocks that follow the semantics of the synchronous reactive (SR) model. All blocks are internally
FSMs except for find peaks, which is refined into an SDFG. While synchronous communication is a perfect match for the FSMs, it is not suitable for the SDFG.

Since it also has to conform to the synchrony hypothesis, its iteration is assumed to be instantaneous and must be executed simultaneously with other SR blocks. As a result, an inefficient implementation is likely to emerge from this specification. Another option is to use a dataflow model such as KPN at the second hierarchical level (state S2). In that case, the interface between KPN and SDF would be straightforward since both models use FIFO channels for communication. However, when blocking reads of KPN are imposed on parameter settings FSM, it cannot react to external events. With parallel heterogeneity used in DDFCharts, FSMs are free to react to external events and SDFGs can run at their own speed.

<table>
<thead>
<tr>
<th>Tick</th>
<th>FSM1</th>
<th>FSM2</th>
<th>FSM3</th>
<th>FSM4</th>
<th>FSM5</th>
<th>FSM6</th>
<th>FSM7</th>
<th>ReMIC clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>S11→S12</td>
<td>→S21</td>
<td>→S31</td>
<td>→S41</td>
<td>→S51</td>
<td>→S61</td>
<td>→S71</td>
<td>933</td>
</tr>
<tr>
<td>m</td>
<td>S12→S12</td>
<td>S23→S21</td>
<td>S33→S34</td>
<td>S41→S42</td>
<td>S51→S51</td>
<td>S61→S61</td>
<td>x</td>
<td>1302</td>
</tr>
<tr>
<td>p</td>
<td>S12→S12</td>
<td>S22→S22</td>
<td>S32→S33</td>
<td>S41→S41</td>
<td>S51→S51</td>
<td>S61→S61</td>
<td>S74→S75</td>
<td>1047</td>
</tr>
<tr>
<td>q</td>
<td>S12→S12</td>
<td>S23→S21</td>
<td>S34→S31</td>
<td>S43→S41</td>
<td>S51→S54</td>
<td>S61→S62</td>
<td>x</td>
<td>1146</td>
</tr>
</tbody>
</table>

Table 1: Examples of master clock ticks in frequency relay case study

Figure 17: Frequency relay in Ptolemy (node N1)

The languages based upon the synchronous model of computation [8] are often divided in two groups: state based languages and dataflow languages. Two of the most popular state-based synchronous languages are Esterel [9] and Argos [10]. They are convenient for specifying control-dominated behavior but at the same time it is difficult to use them for specification of data-intensive digital signal processing algorithms. In the other group of synchronous languages, Lustre [11] and Signal [12] are probably the most popular ones. Dataflow synchronous languages have been applied successfully in design of digital signal processing systems where samples arrive at a regular rate. However, they are unsuitable for specification of control-dominated behaviors. To avoid any confusion, we need to mention that Lustre has been successfully applied in design of linear control systems with feedback loops where one or more variables have to be kept close to their reference point. However, according to the definition in this paper this is a data-dominated behaviour since it involves intensive calculations at regular intervals. Control-dominated behaviour involves decision-making logic.

It is interesting to note that multclock semantics of the Signal language may appear similar to semantics of DDFCharts where multiple clocks are also important. However, the formal semantics of DDFCharts is state based. Clocks in DDFCharts are strongly linked to states. This can best be seen from the rendezvous communication when two clocks synchronize. A rendezvous is always defined with respect to a particular set of rendezvous states. The multclock semantics of DDFCharts probably has more in common with the semantics of Multiclock Esterel, which will be discussed later in the section.

While any single synchronous language would have difficulties specifying a heterogeneous system consisting of control-dominated and data-dominated parts, a combination of a state-based language and a dataflow language would certainly look promising. The most developed such combination is probably the mix of Esterel and Lustre in SCADE design environment [27]. The Esterel/Lustre combination is comparable to a single DDFCharts node. The key difference is in the fact that the Esterel/Lustre combination is purely synchronous while the DDFCharts contains both synchronous and asynchronous compositions.

Whether a pure synchronous specification can always produce a fast and efficient implementation is debatable. Synchronous specifications are not constrained to single-processor implementations. A variety of desynchronization techniques [28] developed in recent years enable synchronous specifications to be mapped onto multiprocessor architectures. So far, there is no undisputable evidence showing that the pure synchronous model of computation cannot cope well with complex embedded systems. However, it is reasonable to argue that forcing all processes at the specification level to operate at related rates can make the resulting implementation slower and more expensive. In particular, this can easily happen when control-dominated processes, which receive inputs in irregular intervals, have to be interfaced with data-dominated processes, which have to perform intensive computations on data samples arriving periodically. This was illustrated in section 7. It is also worth mentioning that experiments conducted with Polis design environment [20] showed that mixed synchronous/asynchronous specifications

### Table 1: Examples of master clock ticks in frequency relay case study

<table>
<thead>
<tr>
<th>Tick</th>
<th>FSM1</th>
<th>FSM2</th>
<th>FSM3</th>
<th>FSM4</th>
<th>FSM5</th>
<th>FSM6</th>
<th>FSM7</th>
<th>ReMIC clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>S11→S12</td>
<td>→S21</td>
<td>→S31</td>
<td>→S41</td>
<td>→S51</td>
<td>→S61</td>
<td>→S71</td>
<td>933</td>
</tr>
<tr>
<td>m</td>
<td>S12→S12</td>
<td>S23→S21</td>
<td>S33→S34</td>
<td>S41→S42</td>
<td>S51→S51</td>
<td>S61→S61</td>
<td>x</td>
<td>1302</td>
</tr>
<tr>
<td>p</td>
<td>S12→S12</td>
<td>S22→S22</td>
<td>S32→S33</td>
<td>S41→S41</td>
<td>S51→S51</td>
<td>S61→S61</td>
<td>S74→S75</td>
<td>1047</td>
</tr>
<tr>
<td>q</td>
<td>S12→S12</td>
<td>S23→S21</td>
<td>S34→S31</td>
<td>S43→S41</td>
<td>S51→S54</td>
<td>S61→S62</td>
<td>x</td>
<td>1146</td>
</tr>
</tbody>
</table>

Table 1: Examples of master clock ticks in frequency relay case study
of control-dominated systems resulted in significantly better performance than purely synchronous ones.

A strong advantage of purely synchronous specifications is a large reduction of verification effort due to the absence of interleaving among concurrent processes. The DDfCharts model of computation allows every SDFG to operate at its own speed in order to support implementation efficiency. However, it also possible to relate the iteration time of a SDFG to a master clock when verification effort needs to be reduced. This flexibility in DDfCharts design flow makes it possible to address both implementation and verification issues effectively.

The globally asynchronous locally synchronous (GALS) model of computation is highly suitable for design of distributed systems due to its composition. The top level consists of a set of nodes that are asynchronously composed, but the processes inside each node must be synchronous. Several GALS languages with formal semantics have been proposed including Multiclock Esterel [9], Polis [20], Communicating Reactive Processes (CRP) [29], Communicating Reactive State Machines (CRSM) [30] and SystemJ [31]. Most of these languages have adopted various constructs from Esterel to support control-dominated behaviour. DDfCharts is more general than GALS since asynchronous compositions are not restricted to the top level of hierarchy. This, however, is not the principal difference between DDfCharts and these languages. Some of the mixed synchronous/asynchronous languages developed from Esterel do not strictly follow GALS model either. For example, a version of Multiclock Esterel described in [32] allows asynchronous compositions at any hierarchical level. The principal difference lies in the level of support for digital signal processing algorithms. DDfCharts directly supports digital signal processing because of the presence of SDFGs. Some of the GALS languages derived from Esterel contain powerful data types but do not contain any specific features that are suitable for DSP operations. Thus, they cannot handle heterogeneous systems effectively.

Due to the use of rendezvous, DDCharts also has similarities with process algebras such as Calculus of Communicating Systems (CCS) [14] and Communicating Sequential Processes (CSP) [15]. While process algebras provide a good semantic framework for analyzing various models of computation, it is difficult to use them for practical specifications since their processes are quite simple. They are just sequential threads, which do not contain enough features to address requirements for control-dominated and data-dominated behaviors.

IX. CONCLUSION

We have presented DDfCharts, a language based on a formal model of computation targeting distributed heterogeneous embedded systems that combine control-dominated and data-dominated behaviors. Using several operators, DDfCharts allows compositions of finite state machines (FSM) with synchronous dataflow graphs (SDFG). In addition, the top level of DDfCharts specification consists of nodes, which are convenient abstraction for describing distributed systems. In DDfCharts, FSMs may be driven by different clocks. Moreover, SDFGs represent separate clock domains as they run at their own speed. Initially, all clock domains run asynchronously. However, it is possible to relate clock speeds in order to achieve deterministic behavior and reduce verification effort. This is an important strength of the DDfCharts design methodology, which allows the designer to exploit the benefits of both asynchronous and synchronous implementations. We have also described low level data structures that are needed for implementation of DDfCharts. While no particular architecture is required for implementation, a heterogeneous architecture where some processing elements perform control-dominated tasks while others perform data-dominated tasks efficiently is likely to excel. We have suggested a possible heterogeneous architecture consisting of reactive and traditional processors, and presented important results from the frequency relay case study.

REFERENCES


[22] www.altera.com


Ivan Radojevic received the BE and PhD degrees in electrical engineering from the University of Auckland in 2001 and 2007 respectively. He is a research engineer at Defence Technology Agency, a branch of New Zealand Defence Force that provides science and technology solutions. His research interests include languages and models of computation for embedded systems, automated synthesis, heterogeneous computing architectures and digital signal processing algorithms for wireless communications.

Zoran Salcic is a professor of computer systems engineering at the University of Auckland. He received the BE (1972), ME (1974), and PhD (1976) degree in electrical engineering from the University of Sarajevo while doing most of his PhD research at the City University of New York (CUNY). He published more than 200 journal and conference papers in the areas of complex digital systems design, custom-computing machines, reconfigurable systems, FPGAs, processor and computer systems architectures, embedded systems and their implementation, design automation tools for embedded systems, hardware-software co-design, new computing architectures and models of computation for heterogeneous embedded systems. He is a fellow of the Royal Society New Zealand (Academy of Science) and senior member of the IEEE.

Partha S Roop is a Senior Lecturer in the Department of Electrical and Computer Engineering at the University of Auckland in New Zealand. He has a PhD in Computer Science from the University of New south Wales, Sydney, Australia (2000), an MTech from the Indian Institute of Technology, Kharagpur, India (1992) and a BE from the College of Engineering, Anna University, Madras, India (1989). His primary research interests are in the design and verification of embedded systems. Of particular interest to him are formal verification techniques such as model checking and module checking and their applications in embedded systems. Roop is on the editorial boards of Elsevier journal of Microprocessors and Microsystems and EURASIP Journal on Embedded Systems.