Adaptive software synthesis from extended dataflow specifications

Ivanilton Polato[1*] and Antonio Mendes da Silva Filho[2]

*[1]Faculdades de Ensino Superior do Centro do Paraná (UCP), Pitanga, Paraná, Brazil. [2]Centro de Estudos e Sistemas Avançados do Recife (CESAR), Recife, Pernambuco, Brazil. *Autor para correspondência. e-mail: ipolato@ucppitanga.edu.br*

ABSTRACT. Embedded software development approaches used models of computation such as dataflow, discrete events, synchronous/reactive, among others. Due to the specialization of the existing models, each one can be better applied to a specific application domain. Nevertheless, when there is no solution for applications in a specific domain, heterogeneous models have been used. In this context, this paper discusses a heterogeneous model called Extended Dataflow. It is an extension of the dataflow model with support to event handling. This paper also addresses how software can be synthesized from extended dataflow specifications and discusses the development of a code generation tool prototype. This takes into account the possibility of component reuse for developing digital signal processing applications. A case study of adaptative applications using digital filters is used to illustrate our approach.

Key words: models of computation, embedded systems, dataflow, components, events.

RESUMO. Síntese de Software Adaptativo baseada em Especificações *Extended Dataflow*. As abordagens de desenvolvimento de software embutido têm feito o uso de modelos de computação, tais como fluxo de dados, eventos discretos, síncrono/reativo, dentre outros. A especialização desses modelos faz com que sejam apropriados a um domínio específico de aplicações. Entretanto, quando não existe uma solução adequada para determinada aplicação, os modelos heterogêneos têm sido utilizados. Neste contexto, este artigo discute um modelo heterogêneo, chamado Extended Dataflow, que é uma extensão do modelo de fluxo de dados com suporte ao tratamento de eventos. O artigo mostra ainda como um software pode ser obtido a partir de especificações usando Extended Dataflow e discute o desenvolvimento de um protótipo de ferramenta de geração de código. Isso leva em consideração a possibilidade de reuso de componentes em aplicações de processamento digital de sinais. Um estudo de caso sobre aplicações adaptativas envolvendo filtros digitais é utilizado para ilustrar o trabalho.

Palavras-chave: modelos computacionais, sistemas embutidos, fluxo de dados, componentes, eventos.

Introduction

Software development has been supported by several methodologies along the last decades. Most of these methodologies assume that computation is accomplished as a result of mathematical functions, expressed as procedures or methods. Henceforth, these functions basically transform input data into output data. However, it is not every software works like that. Consider, e.g., embedded software. Its main role is not data transformation, but the interaction with the physical world. Usually, it is not executed in traditional computers but within telephones, robots, cars, airplanes, and others.

During the development of embedded software, a special attention is given to their main characteristics. Concurrency and performance are important characteristics that should be appropriately taken into account during software design. Other constraints such as development time and resources required must also be satisfied. Development time should also be addressed to satisfy, e.g., time to market requirement.

According to Lee (2002), characteristics such as timeliness, concurrency, interfaces, heterogeneity, and reactivity are considered intrinsic to embedded software. These characteristics involve interaction with the physical world. Embedded software takes into account time, resources consumption and an endless life cycle. Another important factor being considered in embedded software is heterogeneity. Their interaction with real world systems may be accomplished in several ways, causing that software uses several models of computation and implementation technologies.

As a result of many constraints and characteristics that could be met, traditional methodologies have provided little support for

developing this kind of software. The existing models of computation have been trying to fill in this existing gap within embedded software development. They can be considered as a set of rules that govern the interactions between the model components. Most of these models of computation have been developed over the last two decades, each one giving support to specific application domain. When two or more domains are involved, a solution has been the creation of heterogeneous models.

Within this context, this paper presents a heterogeneous model of computation called Extended Dataflow (XDF) (Polato, 2004), which is an extension of the original dataflow model (Dennis, 1974; Davis and Keller, 1982; Ackerman, 1982) with additional support to event handling. This model has been applied to digital signal processing (DSP) applications. Specially, we have been using the XDF model to support the development of adaptative DSP applications involving digital filters. These applications can modify its initial configuration as long as events of either performance or quality degradations take place. This model of computation aims at satisfying system requirements when developing embedded software.

Given the preceding, a set of models of computation related to ours are discussed in the following section. The XDF model of computation and the use of components within the model are presented in Sections 3 and 4, respectively. Section 5 presents our code generation tool and explains how software can be synthesized from an XDF specification. A case study applying the XDF model to an adaptative DSP application is made in Section 6. Finally, concluding remarks are given in Section 7.

Models of computation

Models of computation have been used within embedded software design providing it with a set of rules that define how interactions between components can occur. It may also be viewed as a conceptual framework in which a design is made from the composition of components. Each model has its advantages and limitations, being better suitable to an application domain due to its specific requirements. The most prominent models of computation are discussed next.

Dataflow (DF)

In the dataflow (DF) model, actors are considered atomic entities, carrying out indivisible computations. These actors only start computing when all the input data needed for a computation is available. It is a powerful model to support parallel computation. This model appeared as one of the first attempts for exploring parallelism in programs (Dennis, 1974; Davis e Keller, 1982; Ackerman, 1982).

The DF model is widely used in digital signal processing applications. Usually, the DF model uses block diagrams as a mechanism to describe and explain graphically the algorithms of signal processing. The use of block diagrams is based on a metaphor of circuits as well it establishes a connection with the origins of digital signal processing. It also facilitates the visualization of complex systems. Such characteristics have made the DF model to be largely used in the community of digital signal processing.

Two important issues of the DF model are the support for concurrency and the lack of synchronism. These characteristics are present because the DF model depends upon the data availability. As a result, several components can be ready for execution simultaneously, giving support to concurrency. DF model can be mapped to software specifications and, specifically, embedded software. However, the original dataflow model is not suitable for mapping hardware specifications because of the lack of synchronism. This weakness was solved with special cases by using synchronous dataflow and dynamic dataflow models. In addition, control-oriented systems cannot be suitably supported by DF model.

The concurrency supported in the DF model is also kept in the XDF model. However, the lack of synchronism has been removed in XDF using the synchronism approach existing in the SDF model, where the data consumed and produced by each component are specified in advance.

Synchronous/Reactive (SR)

The SR model (Benveniste and Berry, 1991) deals efficiently with concurrent models using irregular events. In this model, connections between components contain values associated with system global time. Due to this characteristic, all components have the same notion of time within the system. Components represent the relationship between inputs and outputs of a system at each time unit. The SR model is appropriate to applications that have concurrent and complex logical control as, for instance, critical systems. Examples of languages using SR model are Esterel (Berry and Gonthier, 1992) and Signal (Gautier and Le Guernic, 1987).

Discrete events (DE)

The DE model provides a useful abstraction for real-time systems. Milner (1980) and Hoare (1985) proposed the first studies in this area. Later, Lee (1999) proposed a model for studying and handling discrete events. In this model, connections represent a group of atomic events along a timeline. Each event receives a pair (value, time) where time is used for determining the occurrence of an event. Events with the same time are ordered based on data precedence. Similarly to the SR model, a solid notion of global time exists. However, the main difference is the importance given to the time between events within a system.

The DE model is largely used for hardware specification and telecommunications systems simulation. The DE model has been applied in several environments, simulation languages, and hardware description languages, such as VHDL (Lipsett *et al.*, 1989) and Verilog (Thomas *et al.*, 2002). However, the DE model has a high implementation cost in terms of software due to the need to support a global time and an event manager.

Finite state machine (FSM)

The FSM (Gill, 1962) model differs from others because it is a strictly sequential model. In this model, a state can be considered a component. During the execution of the model only one state can be active at a time. Connections between components are represented as transitions. The execution of this model can be viewed as navigation between system states as transitions get fired.

The FSM model is appropriate to describe the control logic in embedded software and, more specifically, in critical systems. It can be used to predict the behavior of a system provided that a formal analysis can be made. It can also be easily mapped to hardware and software implementations.

However, the FSM model has limitations. It is not sufficiently complete to describe all the existing recursive functions of a system. Other limitation is the number of states can rapidly grow even when dealing with a minimum complexity. This happens because the number of states can become large as the system complexity increases, making difficult the management of the states. Despite that the FSM model is highly used to compose heterogeneous models because of the predictable behavior it provides to a system.

Heterogeneous models

Heterogeneous models have been used to overcome existing limitations in the previous models. In general, they combine two or more models of computation, involving different application domains. However, a problem when creating a heterogeneous model is the semantics resulting of the new model. Two options are commonly taken into consideration: the use of original semantics model or the creation of a new operational semantics model for the heterogeneous model.

For instance, consider the FSM model, it has been combined with different models in two different ways. The first one combines the FSM model with the SR model. As a result, Statecharts (Harel, 1987) has been obtained. In Statecharts, three elements are presented: hierarchy, concurrency and broadcasting. These three elements turn the behavior of complex systems into simplified diagrams. It can also be used to specify systems behavior.

In addition, FSM has been combined with three other models: DE, SR and DF models (Girault *et al.*, 1999), which has been called ⋆charts (pronounced "starcharts"). Note that a concurrent model can be chosen independently of the use of an FSM model. The main difference between these two ways of creating heterogeneous models lies on the semantics. While in Statecharts the semantics of the FSM model is strongly coupled with the concurrency model semantics, ⋆charts decouples the semantics of the concurrency model from the FSM.

DF model has also been combined with DE model resulting in other heterogeneous model (Chang *et al.*, 1997). This new model maintains the main characteristics of the DE model concerning the event handling.

Besides the heterogeneous models, there are other specific models. The DF model is widely used in DSP applications. However, this model does not support synchronism, essential for DSP applications. To overcome this limitation, models such as Synchronous Dataflow (SDF) (Lee e Messerschmitt, 1987) and the Dynamic Dataflow (DDF) (Buck, 1993) have been proposed. These models have new characteristics not supported by the DF model. For example, the SDF model solves the synchronism problem, by defining in advance the amount of data consumed and produced by each component. Henceforth, it is possible to generate schedules that run synchronously. Nevertheless, no support has been given to real-time event handling within dataflow models. To fill in this existing gap, the XDF model has been developed.

Extended dataflow

This section presents a heterogeneous model called Extended Dataflow (XDF) (Polato and Silva Filho, 2003, 2005) which has been applied to embedded software. The reader is referred to (Polato, 2004; Silva Filho, 2004) for further details. This model is an extension of the DF model (Dennis, 1974; Davis and Keller, 1982; Ackerman, 1982). The main characteristics of the DF model as well as a subset of characteristics of the SDF model have been kept. They include:

XDF can work synchronously or asynchronously, based on a set of parameters, defined at compile time, when creating a model specification to an application.

The firing conditions of XDF components are data dependent, i.e., its execution depends upon data availability.

The amounts of data being produced and consumed by each component are known in advance. Nevertheless, this is only possible when using the model in a synchronous manner.

In XDF, event handling is done in real-time. As well, events have a priority used to solve conflicts between two or more events occurring simultaneously. Chang proposed a similar model (Chang et al., 1997) by combining the DF model with the DE model. However, two characteristics make Chang´s model different from XDF:

The used dataflow model can be viewed as a generic model supporting a weak synchronism.

Event handling is done according to the time associated to the event, i.e., in agreement with a global timeline within the system.

XDF has been used in embedded software development, mainly in adaptative DSP applications involving digital filters and speech compression (Polato and Silva Filho, 2003; Polato, 2004). To support specifications of software using the XDF model, a model specification has been developed. The XDF model spec can be viewed as a description language for the software being designed, where components, connections between them, inputs, outputs, and system events are specified. A generic XDF model spec is shown in Listing 1. Due to its flexibility and extensibility, XML language has been chosen to describe XDF model spec (Silva Filho, 2004). XML also provides means to support the XDF syntax through schemas. Although XML model spec has been proposed to meet requirements of a specific application domain (DSP), an extension of it can be thought of to meet new requirements in other application domain, but this issue is out of scope of this paper. Note that in Listing 1, only the main XML closing tags are shown.

```
<modelSpec>
  <systemName>
  <systemHeader>
    <componentName>
    ...
    <port>
      <portName>
      <portType>
      <portCapacity>
    ...
    <connector>
      <connectorName>
      <sourceComponentPort>
      <targetComponentPort>
    ...
    <event>
      <eventName>
      <eventPriority>
    ...
  </systemHeader>
  <systemBody>
    <component>
      <componentName>
      <interface>
        <in_port>
          <portName>
          <tokensConsumed>
          <minimumToFire>
          <connectorName>
        ...
        <out_port>
          <portName>
          <tokensProduced>
          <connectorName>
        ...
        <configurable_parameter>
          <paramName>
          <paramType>
          <paramSize>
          <paramValue>
        ...
        <accept_event>
          <eventName>
          <componentName>
          ...
        <raise_event>
          <eventName>
          <componentName>
          ...
      </interface>
    </component>
  </systemBody>
</modelSpec>
```

Listing 1. Generic XDF model specification

In a second abstraction level of the XDF model spec, components are individually specified. At this level, inputs and outputs of each component are specified. In addition, events being raised or accepted by components are also specified at this level. In this model, only registered events in a component specification can be raised or accepted by a component. Possible exceptions occurring during the execution of a system are handled as events with maximum priority.

XDF model supports the specification of systems that requires either synchronous or asynchronous mode. XDF model works synchronously by defining the parameters tokensConsumed, minimumToFire and tokensProduced. If these parameters are not

specified in a model specification, XDF model will work asynchronously. These parameters are located in the component interface and are child nodes of inPort and outPort elements. Besides the specification of system components, the XDF model spec also defines how event handling is carried out. For events raised by the system, there are specified conditions that should be satisfied to raise a specific event. Events accepted by a system have specified actions that should be carried out when an event occurs. Every specified event likely to be raised by a system should have an equivalent specification with actions to be carried out for handling that event.

The XDF approach goes beyond the proposition of the XDF model spec. To assure the correctness of the specification, both syntax and semantics of the model have been formally defined. XDF syntax uses XML. In addition, an operational semantics has been developed avoiding inconsistency during the execution of the model. Further details can be found in (Polato, 2004).

XDF components

Our approach used components to compose new applications. Thus, XDF can take advantage of component-based development (CBD), once XDF allows a designer to compose system by instantiating and combining existing components. CBD approaches have been widely studied and can benefit software development process. Even so, CBD faces limitations. First one is component retrieval. Components must be retrieved to match the desired functionality. Components must be chosen to correctly meet non-functional requirements, being a key issue when developing embedded software.

Note that research involving component selection from a repository which aims at matching non-functional requirements is currently in development (Yen *et al.*, 2002). Therein, an approach to cope with the problem of component retrieval is discussed. An integrated mechanism for component-based development of embedded software is further discussed in (Yen *et al.*, 2002). In this work, a repository has been created to provide mechanisms of component retrieval according to non-functional requirements. For this purpose, they assume that the components selected are the most suitable to compose an application.

Although CBD faces the limitations before mentioned, it also provides benefits to the software development. The major one is the reuse of software components. Reuse may both improve quality of software and reduce development time. By using XDF model, a reduction in terms of development effort is expected, provided that the developer can reuse software in two ways. First, the component itself can be reused in several applications. Second, the specification of this component within the model can be reused to create new applications using the same component. Note that quality improvement of an application may indirectly come from using components being already tested and used.

It is worth observing that the motivation to use components is not only related to software reuse, but also to time-to-market requirements where products must be released as soon as possible. Another benefit from using components when developing applications is the ease of maintenance and update of such applications. Components can be more easily upgraded or replaced.

Furthermore, to create an application using XDF, one could assume that a black box approach for component is used. In addition, we consider that a component must have a life cycle as defined in the operational semantics of the model, as well as the interfaces of a component must also be well known. This is discussed next.

Components life cycle

XDF component life cycle has been developed to provide a base to XDF operational semantics (Polato, 2004). Within this life cycle, components can initially assume two states: inactive or active. Components assume the active state when executing. The active state has four sub states: waiting, ready, running and suspended. The waiting state is reached by a component only when using the model synchronously. It makes a component to wait until all the input data needed to accomplish its computation is available. When this condition is satisfied, a system event called trigger will change the state of a component to ready. The ready state is assumed by a component when all conditions to start an execution of a component are satisfied. A component in the ready state waits to start its execution.

When using the model synchronously, a component with one or more input ports will be initially in the waiting state. Once its firing conditions are satisfied, it will go to the ready state. An exception occurs when a component does not need input data to accomplish its computation. These components without input ports, when selected, go directly from the inactive to the ready state. This transition occurs because these components do not have to meet any fire conditions and can be requested to execute at any time.

The execution of a system is directed by an entity called system coordinator. This entity is responsible for

holding the schedules for the system, to select which components will be running using the XDF life cycle, and to solve possible problems of memory usage. The system coordinator is also responsible for handling the events within a system. By dealing with system events, the system coordinator controls the state transitions of a component. By dealing with component events, it can control the events generated by components during the system execution.

When requested by the system coordinator, a component in the ready state can execute. This takes place through a system event called start, which makes the component enter the running state, as shown in Figure 1. When in the running state, there are two possibilities of state change for a component:

The suspend event changes the state of a component from the running to the suspended state.

The stop event changes the component state to two target states: waiting or ready. The transition is made based upon the number of input ports. If the component does not have input ports, it returns to the ready state, where it will be able to be invoked again. If there is any input port, components return to the waiting state, where should wait the fire conditions to be satisfied again.

Finally, when a component is in the suspended state, it awaits the resolution of events or related actions that caused its entrance in this state as, for instance, exceptions. When these are solved, the component returns to the running state. This state change is caused by the resume event. The complete XDF components life cycle is shown in Figure 1.

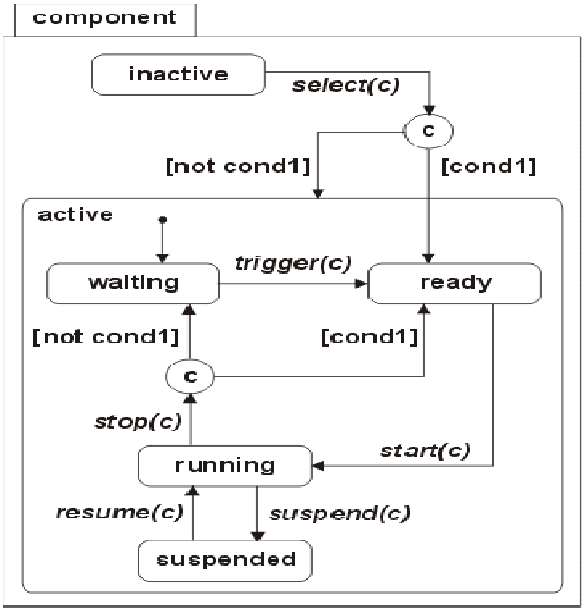


Figure 1. XDF Components life cycle.

Code generation tool

This paper also addresses the development of a prototype of code generation tool. The main purpose of this tool is to help the development of applications that used the XDF model. Its main functionality is the partial code generation from XDF specifications.

Using this tool, C code is generated from the XDF model spec. The tool also allows the edition of a model spec for an application. The C language has been chosen in part because it is widely used within the DSP area. Other reason for using C language is the performance it can provide.

The prototype can be divided into two major parts: code generation module and GUI module. Figure 2 shows the architecture of the prototype. The code generation module is responsible to generate the C code, having an XDF model spec as input. The specification is read and stored in memory. Each part of a specification has a table to hold the related data located in the model spec. For example, there are tables to keep information about components, connectors, events, and ports of a system. The reader is referred to Figure 3.

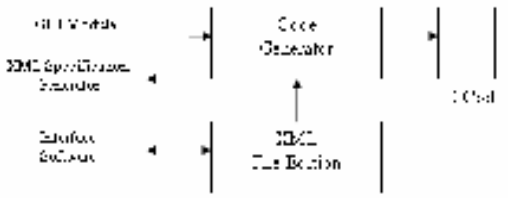


Figure 2. Code generation tool prototype architecture.

To generate the code from the XDF specification, a mapping procedure is used. Components, ports, connectors, and other entities of the model specification have been mapped. Once all the information has been read from the specification and stored in the memory, the mapping algorithm is called to generate the code. Firstly, a default header for C programs is generated. This header contains the basic libraries used in a C program. Once the header is generated, prototypes of the components are generated. The component table is read and the prototype of each entry from this table is generated.

In addition, the declaration of the ports of a system is generated. In this case, the semantics considers that all communication is made through buffers. So, at this point, declarations of the connectors are generated. This generation is illustrated in Figure 3. Note that the code generator accesses both connector and port tables to create a buffer declaration in the C code.

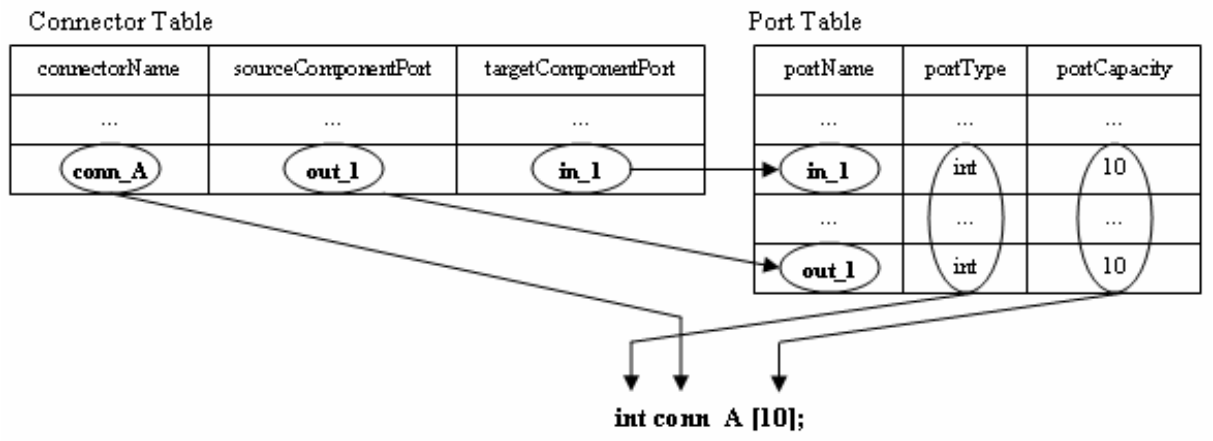


Figure 3. Example of buffer code generation.

After the generation of the headers of an application, involving libraries and buffers, the main program is generated. In this case, a single schedule is generated, according to the specification. The schedule generated contains at least one entry of each component in a system.

It is worth highlighting that we can use either our components or third party components in the XDF model specification and generate C code from this specification. Note that a component in this case can be seen under the 'black box' approach, where by knowing the functionalities and interfaces, we can use a component to compose an application using the XDF model. A benefit of this approach is that components can be reused, which results in less development effort. Figure 4 shows a screenshot of our code generation tool.

Figure 4. Code generation tool screenshot

The GUI module allows the user to develop applications using component diagrams. Components and connectors can be instantiated and specified graphically. Once a system has been completely specified using components diagrams, the XDF model spec can be automatically generated by the tool. With the model spec at hand, the code generator module is called and generates the partial code for the system. A complete example is given next.

A case study

To illustrate the use of the XDF model, a case study of an application involving digital filters is discussed. Digital filtering is one of the most important functions within the DSP area, being widely used. Applications including speech, image, and video processing are just a few examples which digital filters can be applied to.

Generally, digital filters are used with two main purposes: signal restoration and signal separation. The first case is used when the signal has been distorted some way. The second one is applied when the signal has been contaminated with interference, noise, or even other signals.

A digital filter works as follow: the analog input signal must first be sampled and digitized using an ADC (analog-to-digital converter). The resulting binary numbers, representing successive sampled values of the input signal, are transferred to the processor, which carries out numerical calculations on them. These calculations typically involve multiplying the input values by constants (these constants are called coefficients) and adding the products altogether. In addition, the results of these calculations, which now represent sampled values of the filtered signal, can be output to a DAC (digital-to-analog converter) to convert the signal back to analog form. Note that in a digital filter the signal is represented by a sequence of numbers rather than a voltage or current, generally, used in an analog filter. The whole process is shown in Figure 5.

Figure 5. Signal conversion and filtering.

The process of recording a song using instruments (bass, guitar, drums, piano, etc) could illustrate the transformation of the analog signal through an ADC to a digital signal. To play the recorded song on a CD player could represent the opposite process where the digital signal is converted back to an analog signal through a DAC.

In addition, digital filters can be classified as being recursive or non-recursive filters. A recursive filter is one, which, in addition to input values, also uses previous output values. These, like the previous input values, are stored in the processor memory. A non-recursive filter is also known as an FIR (Finite Impulse Response) filter while a recursive filter as an IIR (Infinite Impulse Response) filter. An FIR filter is one whose impulse response is of finite duration. An IIR filter is one whose impulse response theoretically continues forever because the recursive (previous output) terms feed back energy into the filter input and keeps it working.

Besides an impulse response, digital filters also have a step response and a frequency response. Each of these three responses provides complete information about the filter, but in different forms. If one out of three is specified, the other two can be directly obtained. All these three representations are important because they describe how a filter will react under different circumstances.

Another important characteristic of digital filters is the filter length. The length of a recursive filter is the largest number of previous input or output values required to compute the current output. Since the non-recursive (FIR) filters uses only the current and previous inputs to compute the current output, the order of a FIR filter is the number of previous inputs (stored in the processor memory) used to calculate the current output. Moreover, filters may be of any order from zero upwards.

Other characteristic of digital filters comprises the coefficients. The values of these coefficients determine the characteristics of a particular filter. Both FIR and IIR filters need these coefficients in order to do their job. There are several ways to calculate the coefficients of a filter. However, this is out of scope of this paper.

The example system illustrated here is one designed to improve the signal-to-noise ratio (SNR) of an input signal. A noise often causes degradation in terms of speech quality and intelligibility. So noise reduction is an important feature to improve the quality of a signal in DSP applications.

In the example shown in Figure 6, a system receives an input signal and one out of three filters is selected according to the needs of the system. These filters comprise two FIR lowpass filters and one LMS adaptative filter. A fourth component appears in the model, and it is called SNR. This component is in charge of selecting one out of three digital filters. The system is presented in terms of its components. Note that the system can adapt itself to the environment, according to the system needs. This is made by the component SNR, which selects the appropriate filter to the input signal.

SNR is the component that selects one filter out of three that will be applied to the input signal. A filter will be used according to the conditions detected by the SNR component. Herein the need for improving the ratio signal-to-noise is considered. These filters are usually used to remove Gaussian white noise present in the signal. In this case, noise causes degradation of the desired signal quality. This noise source is also represented in Figure 6. Note that, even, one out of three digital filters is in use. Another can become active if the SNR component detects degradation in signal quality, which depends on signal-to-noise ratio among other parameters.

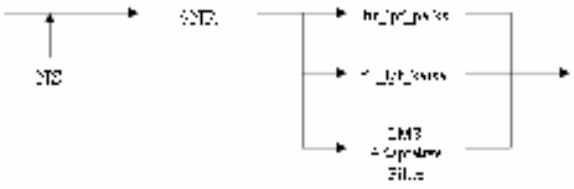
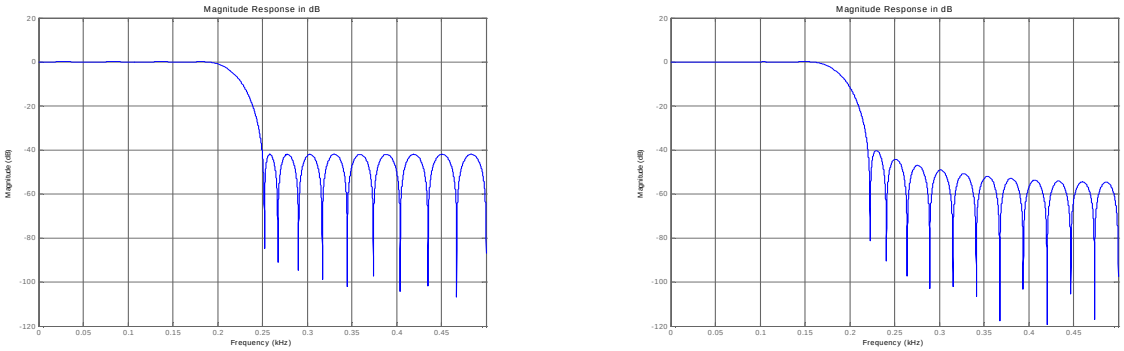


Figure 6. Digital filtering system.

The FIR filter components in this example are all lowpass filters. The fir_lpf_parks component is a filter that uses coefficients generated by the Parks-McClellan method. Figure 7.a shows the frequency response of this component when using 35 coefficients. Consider the following analysis: by increasing the length of the filter a better result can be achieved, according to the 70-point and the 128-

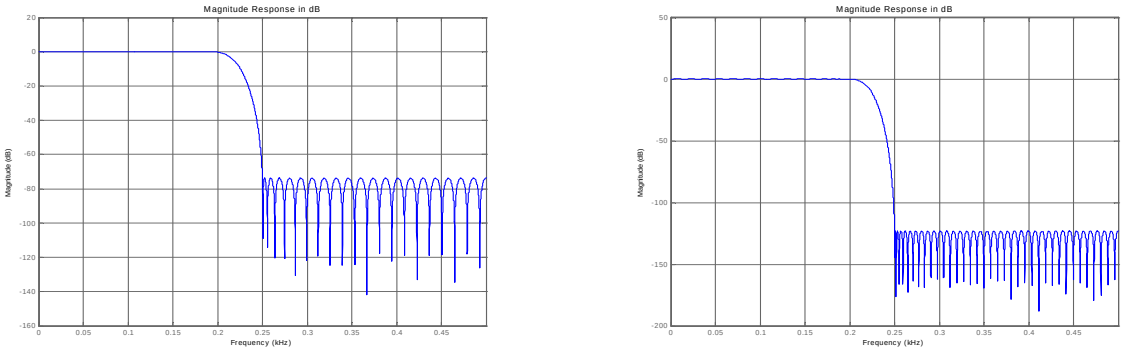
point filter frequency responses shown in Figure 8a and 8b, respectively. Note the larger, the filter, the better, and the results achieved.

The second filter, the fir_lpf_kaiser, uses coefficients developed using the Kaiser Window method. It has a 0.19 cutoff. Figure 7.b shows the frequency response of the fir_lpf_kaiser component when using 37 coefficients. The same analysis carried out with the previous component is made again. The larger, the filter, the better, and the results achieved. Figure 9a and 9b shows the frequency responses for the 70-point and the 128-point filters using coefficients generated by Kaiser Window method.



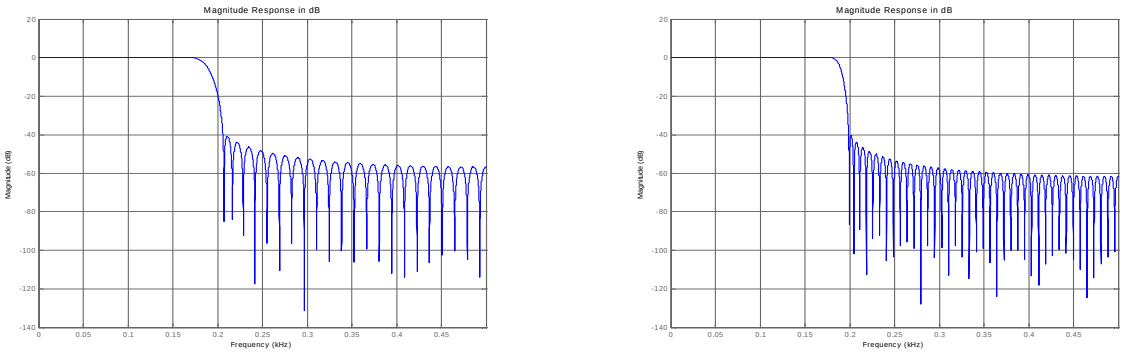
(a) (b)

Figure 7. (a). Frequency response of fir_lpf_parks component using 35 coefficients. **(b).** Frequency response of fir_lpf_kaiser component using 37 coefficients.



(a) (b)

Figure 8. (a). Frequency response of fir_lpf_parks component using 70 coefficients. **(b).** Frequency response of fir_lpf_parks component using 128 coefficients.



(a) (b)

Figure 9. (a). Frequency response of fir_lpf_kaiser component using 70 coefficients. **(b).** Frequency response of fir_lpf_kaiser component using 128 coefficients.

Experiments were made using both FIR filters mentioned above to set standards that are used by the SNR component. The first simulation has analyzed the filter attenuation against the filter length. The filters were applied to an input signal, and the results are shown on Figure 10. Note that the filter using coefficients generated with the Parks-McClellan method has a better attenuation as the filter length increases. The other filter has attenuation almost constant, even when using a larger filter length.

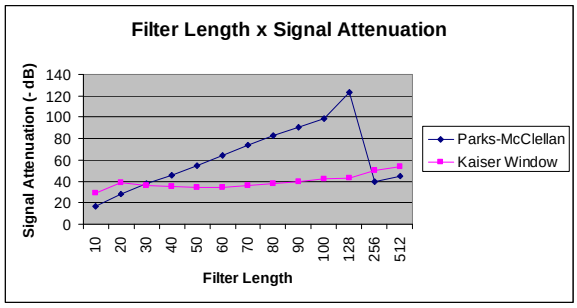


Figure 10. Filter length x signal Attenuation graph.

Other simulations have been made to test the performance of both filters. The results have shown that regardless the method used to generate coefficients – in this case Parks-McClellan and Kaiser Window – the performance of both has shown to be almost the same. The results are illustrated in Figure 11.

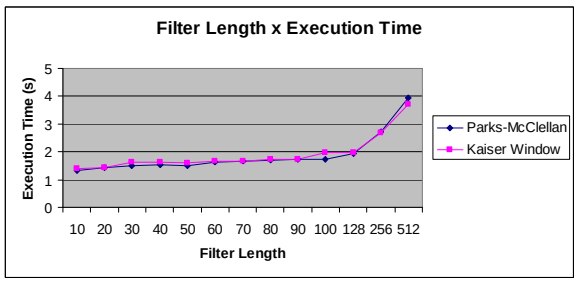


Figure 11. Filter Length x execution time graph.

A third set of simulations has been carried out to check the improvement on the signal-to-noise ratio caused by each filter. The results can be used by the SNR component to select an appropriate filter according to the system needs, as shown in Figure 12.

Although the Figure 12 shows that both filters have almost the same improvement with same amount of coefficients, the difference can be set when selecting a filter that have, for example, a better attenuation. If we select an FIR filter using 128 coefficients generated by the Parks-McClellan method, it will have the same signal-to-noise improvement that a FIR filter using 128 Kaiser Window coefficients. But, the first one will have a better attenuation than the second one.

A third filter appears in Figure 6. It is an adaptative filter that uses the LMS algorithm. The main difference between this filter and the two previously analyzed is that it does not need a method to generate coefficients. An adaptative filter has, initially, all coefficients equal to zero. Using the input signal and the desired response, the filter adapts itself and alters its coefficients to achieve the expected result.

The only parameter that must be set to this particular component is the ***mu*** parameter. This is the parameter that determines the convergence of the filter to the designed response. Although studies have been made trying to achieve an optimal value for ***mu***, this could not be determined.

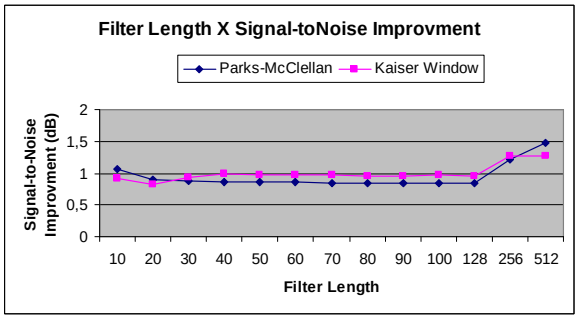


Figure 12. Filter length x signal-to-noise improvement.

Simulations have also been carried out with the LMS Adaptative Filter component. In this case, the ***mu*** parameter and the number of coefficients used by the filter were changed. The values of ***mu*** used were: 0.0001, 0.001, 0.01, 0.1, 0.2 and 0.3. The filter used 10, 20, 30 and 40 coefficients.

When using 10 coefficients and ranging over the ***mu*** values, the filter could not adapt to the desired response and remove the noise from the input signal. The output signal almost matched the input signal, showing that the filter could not remove the noise. The filter using 30 and 40 coefficients could not also adapt to the response signal. But in this case, the output signal was corrupted, distorting the signal.

The filter using 20 coefficients had the best results. Using different values of ***mu***, the filter had different improvements in the signal-to-noise ratio. Table 1 shows the results of these simulations.

Table 1. Signal-to-noise improvement.

	mu	Signal-to-noise Improvement (dB)
20 coefficient LMS adaptative filter	0.0001	4.14
	0.001	2.00
	0.01	0.96
	0.1	0.51
	0.2	0.36
	0.3	- 4.82

The SNR component is in charge of using all these information to provide support for system adaptation. To do so, events are used. The main functionality of the SNR is to choose the most suitable filter according to a specific condition. The SNR component could select a filter according to the parameters set to a system during the development.

The system illustrated in Figure 6 was developed using the XDF model. The model specification was generated and is shown in Listing 2.

```
<modelSpec>
  <systemName>digital_filters</systemName>
  <systemHeader>
    <componentName>snr
    <componentName>lpf_parks
    <componentName>lpf_kaiser
    <componentName>lms_adaptative
    <port>
      <portName>in_snr
      <portType>int
      <portCapacity/>
    </port>
    …
    <connector>
      <connectorName>connector_1
      <sourceComponentPort>out_snr_1
      <targetComponentPort>in_lpf_parks
    </connector>
    …
    <event>
    <eventName>select_lpf_parks
    <eventPriority>0
    </event>
    …
  </systemHeader>
  <systemBody>
    …
    <component>
      <componentName>lpf_parks
      <interface>
        <inPort>
          <portName>in_lpf_parks
          <tokensConsumed>1
          <minimumToFire>1
          <connectorName>connector_1
        </inPort>
        <outPort>
          <portName>out_lpf_parks
          <tokensProduced>1
          <connectorName>connector_3
        </outPort>
        <acceptEvent>
          <eventName>select_lpf_parks
          <componentName>snr
        </acceptEvent>
      </interface>
      <acceptableEvents>
        <on>select_lpf_parks
        <doAccept>start(lpf_parks)
      </acceptableEvents>
    </component>
    …
  </systemBody>
</modelSpec>
```

Listing 2. XDF model spec for the example system.

To compose this example, third party components implemented in C language have been used. The prototype of code generation was used to generate code of the DSP application and successfully generated partial code for the application.

Final Remarks

This paper presents an approach of embedded software synthesis from Extended Dataflow specifications. It has been applied to DSP applications, such as those using digital filtering, which may be able to adapt itself during execution, by choosing the filter that better adapts to specific conditions. XDF model has shown to be suitable to the examples it has been applied to. At present, examples involving digital filters and speech compression have been applied to this model. The code generation tool has successfully generated partial C code for the applications which it has been applied to.

It is worth highlighting that we can use third party components and generate C code from XDF specification provided that the interfaces of these components and their functionality are known in advance. A further step in our research work is currently focused on exploring other niche of applications for this approach.

References

ACKERMAN, W.B. Data Flow Languages. *Computer*, Long Beach, v. 15, n. 2, p. 15-25, 1982.

BENVENISTE, A.; BERRY, G. The synchronous approach to reactive and real-time systems. *Proc. IEEE*, New York, v. 79, n. 9, p. 1270-1282, 1991.

BERRY, G.; GONTHIER, G. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, Amsterdam, v. 19, n. 2, p. 87-152, 1992.

BUCK, J.T. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the token Flow Model.* 1993. Dissertation (Ph.D)-Dept. of EECS, University of California, Berkeley, CA, 1993.

CHANG, W. *et al*. Heterogeneous Simulation - Mixing Discrete Event Models with Dataflow. *J. VLSI Signal Proc.*, v. 13, n. 1, 1997.

DAVIS, A.L.; KELLER, R.M. Data flow program graphs. *Computer*, Long Beach, v. 15, n. 2, p. 26-41, 1982.

DENNIS, J.B. First version of a data-flow procedure language. *Proceedings of the Colloque sur la Programmation, Lecture Notes in Computer Science*, Paris, v. 19, p. 362-376, 1974.

GAUTIER, T.; LE GUERNIC, P. Signal: A declarative language for synchronous programming of real-time systems. Functional Programming Languages and

Portland: Computer Architecture, 1987.

GILL, A. *Introduction to the theory of Finite-State Machines.* Mc Graw-Hill Book Company, Inc, 1962.

GIRAULT, A. *et al*. Hierarchical Finite State Machines with Multiple Concurrency Models. *IEEE Transactions On Comput.-sided Des. Integr. Circuits Syst.*, New York, v. 18, n. 6, 1999.

HAREL, D. Statecharts: A Visual Formalism for Complex Systems. *Science Computer Programming*, v. 8, p.231-274, 1987.

HOARE, C.A.R. Communicating Sequential Processes. *International Series in Computer science.* Prentice Hall, 1985.

LEE, E.A. Modeling Concurrent Real-Time Processes Using Discrete Events. *In:* ANNALS OF SOFTWARE ENGINEERING, Special Volume on Real-Time Software Engineering, v. 7, 1999.

LEE, E.A. Embedded Software. *Advances in Computers*, London: Academic Press, v. 56, 2002.

LEE, E.A.; MESSERSCHMITT, D.G. Synchronous Data Flow. *Proc. IEEE*, New York, v. 75, n. 9, 1987.

LIPSETT, R. *et al*. *Vhdl: Hardware Description and Design*. Kluwer Academic Publishers, 1989.

MILNER, R.A. Calculus for Communicating Systems. *Lecture Notes in Computer Science*, n. 92, Springer Verlag, 1980.

POLATO, I. *XDF–Extendend Dataflow: Uma Extensão do Modelo de Fluxo de Dados com Suporte a Tratamento de Eventos*. 2004. Dissertação (Mestrado)-Departamento de Informática, Universidade Estadual de Maringá, Maringá, 2004.

POLATO, I.; SILVA FILHO, A.M. A Component-based Approach to Embedded Software Design using Extended Dataflow Specifications. *In*: 1. WET – Workshop on Engineering and Technology. 2005.

POLATO, I.; SILVA FILHO, A.M. XDF–Extended Dataflow. *In:* COMPONENT-BASED DEVELOPMENT WORKSHOP*,* 3., 2003, São Carlos. *Proceedings…* São Carlos, 2003.

SILVA FILHO, A.M. *Programando com XML.* Editora Elsevier/Campus, 2004.

THOMAS, D.E. *et al. The Verilog Hardware Description Language.* 5. ed. Kluwer: Academic Publishers, 2002.

YEN, I.L. *et al*. Component-based Approach for Embedded Software Development. *In:* IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING. 5., 2002. Washington, D.C. *Proceedings…* Washington, D.C. p. 402-412. 2002.

Received on December 08, 2004.
Accepted on October 25, 2005.