Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Comparing Asynchronous and Synchronous Approaches to Knowledge Processing

by

Anders Skoglund

LIU-IDA/LITH-EX-G--11/009--SE

2011-06-03

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Final Thesis

Comparing Asynchronous and Synchronous Approaches to Knowledge Processing

by

Anders Skoglund

LIU-IDA/LITH-EX-G--11/009--SE

2011-06-03

Supervisor: Fredrik Heintz
Examiner: Fredrik Heintz

Abstract

This thesis presents a comparison between the synchronous and asynchronous model of computation in the area of knowledge processing. Focus lies on evaluating if a synchronous approach to knowledge processing is possible and practical. This has been done by implementing the reconfigurable fault diagnosis framework FlexDx using the synchronous programming language SIGNAL, a language designed to be used in embedded real-time systems. FlexDx have previously been implemented using the asynchronous knowledge processing middleware DyKnow, from which an example system with multiple failure scenarios consisting of input signals and results were available. Matlab code for many algorithms in FlexDx from the existing implementation could also be reused.

The SIGNAL implementation was tested using one of the available scenarios and the results matched the expected results from the DyKnow implementation almost perfectly.

The synchronous aspect of the new implementation was not a problem as the behavior of all parts of FlexDx that had to be reimplemented easily could be described synchronously. However, using SIGNAL for this purpose proved to be both complicated and cumbersome. This was partly because of the strict declarative coding style, but mostly because of limitations of SIGNAL and the POLYCHRONY compiler. Two such limitations caused most of the problems that were encountered. First, SIGNAL does not support dynamic arrays and all iteration constructs require that the number of iterations is determined at compile time. This could be overcome by using external types and processes, the method used in SIGNAL to import code written in other languages, to implement the needed functionality in C++ and Matlab. Second, the POLYCHRONY compiler provides very limited feedback that can be used to correct non-trivial coding errors, making the task of programming with SIGNAL far more complicated than necessary.

While it is clear that a synchronous approach to knowledge processing works well, it is not practical to write a working implementation of FlexDx using only SIGNAL. Because of the limitations of SIGNAL a large part of the system had to be implemented using other languages.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Purpose

The purpose of this thesis is to compare a synchronous and asynchronous approach to knowledge processing. This will be done by implementing the reconfigurable fault diagnosis framework FlexDx using the synchronous programming language SIGNAL, and comparing the result to an existing asynchronous implementation that uses the knowledge processing middleware DyKnow.

The primary goal of the thesis is to answer two questions.

- If SIGNAL can be used to implement a knowledge processing system as complex as FlexDx.
- If there are any limitations to SIGNAL or the synchronous model of computation that makes it unsuitable for this in practice.

A secondary goal is to evaluate the tools used to construct a system with SIGNAL and the practical aspects of using them.

1.2 Limitations

Instead of performing a thorough comparison of synchronous and asynchronous approaches to knowledge processing, which would require far too much time and effort, the thesis will focus on only one synchronous programming language, SIGNAL, and one example of knowledge processing, FlexDx, that has already been implemented asynchronously. This means that only one implementation needs to be constructed. It will, however, also mean that the comparison will not be complete and that no new code will be written that uses DyKnow.

Chapter 2

Background

In this chapter synchronous and asynchronous programming, DyKnow, FlexDx and SIGNAL will be described. It will focus on what is needed to understand the report and important areas that are of particular interest.

2.1 Synchronous and asynchronous programming

When writing code for real-time systems there are a number of computational models that describes, among other things, different ways of representing time. A programming language can support one or more model, and different models are suited for different situations. The asynchronous and synchronous models are two examples of such models [1][2][3, Ch. 1.2].

With the asynchronous model, tasks will be executed concurrently and the time needed to perform operations can be unpredictable. The temporal aspects of an asynchronous system will be influenced by the execution platform. Factors like the scheduling policy used by the operating system to switch between tasks and processor utilization will have a significant effect on the execution time of a process. Because these factors can vary it is not possible to know what the execution time of a process will be and it is therefore temporally non-deterministic [3, Ch. 1.3.1].

Programming languages like SIGNAL instead uses a synchronous model of computation. The most notable difference is that physical time has been abstracted away and replaced with abstract clocks that, basically, divides time into discrete time slots, starting with an input event and ending with an output event.

All computations following an input event are assumed to occur simultaneously and instantaneously, meaning that the computation of an output event conceptually takes no time. From the programmers perspective everything will happen instantaneously and at the same time. This assumption will not, of course, hold when actually executing the code, but the timing information of events in a time

slot is unimportant as long as the execution platform is fast enough to produce the output event before the next input event arrives. This does however mean that the execution platform must be validated to ensure that this is the case [3, Ch. 1.3.3].

2.2 DyKnow

DyKnow is a knowledge processing middleware for advanced robotic systems designed to bridge the gap between low-level sensor data and high-level knowledge and reasoning [4]. A system is built from a number of sources that provide streams of data from sensors and other data sources, as well as computational units and asynchronous streams that connects them.

A computational unit is similar to a process or function that uses streams for input and output. They also have internal state. A computational unit receives data from a number of input streams, performs some computations, and provides its results on an output stream that other computational units in turn can subscribe to. Streams in DyKnow carry samples that represent observations or estimates of a value at a particular point in time. In addition to the value a sample also have two time stamps. One is the valid time which represents the time when the observation or estimation was made. If a sample is used to create some refined knowledge about an observation the valid time will be preserved, indicating that the new knowledge was valid at the same time as the information it is based on.

If a knowledge process uses a sample from a source, for example a sensor that reports the location of a vehicle, that was produced at time t, and then used the sample to create some refined knowledge, for example the vehicle's average speed, then the sample it outputs will also have the valid time t.

The other time stamp, the available time, represents the time when the value is available in the stream. If for example a sample from a source had the valid time t, then a sample that is based on it will have the available time $t + d$ when it reaches a computational unit, where d is the time it took for the value to reach the computational unit. This delay depends on factors like communication delays and the computation time of intermediary computational units.

A computational unit can output multiple samples in the same stream with the same valid time. This can happen when a computational unit uses a fast algorithm to quickly compute an estimate of a value so that it can be sent with a low delay, but at the same time uses a slower algorithm to compute a more precise estimate. When this value has been computed it is sent in a sample with the same valid time as the first sample, but with a greater available time. However, two samples of the same stream can't have the same available time. The way samples are made available in a stream it is also possible to fetch the previous samples in a stream.

Streams in DyKnow also have policies that are used to specify what restrictions

and guarantees shall be enforced on the values and samples they carry, especially for communication in distributed systems. There are five different kinds of policy constraints in DyKnow.

- Change constraint
- Delay constraint
- Duration constraint
- Order constraint
- Approximation constraint

More information about policy constraints and how the different kinds of policies are used can be found in *DyKnow: A Stream-Based Knowledge Processing Middleware Framework* by Fredrik Heintz [4, Ch. 4.4.5].

One of the more interesting capabilities of DyKnow is that it is possible to reconfigure the networks of streams and computational units at runtime. It is for example possible to remove and add computational units and change the policies of streams while the system is running.

2.3 FlexDx

FlexDx is a reconfigurable diagnosis framework for detecting multiple faults from noisy data using a set of precompiled tests [5]. When using FlexDx the system can be divided into two parts, the first is the system that is monitored, and the second performs the diagnosis. The monitored system is assumed to be synchronous while in the existing DyKnow implementation the diagnosis part is asynchronous.

The diagnosis part receives sensor data from the monitored system and uses it to detect and diagnose faults. It executes independently of the monitored system so that the monitored system won't be affected when faults are detected and diagnoses computed. It is therefore possible that new sensor data is received before it is done computing fault diagnoses for the last set of sensor data, which means that sensor data must be stored until it can be processed. In DyKnow this is done by the streams, but with SIGNAL some sort of buffer will be needed.

The main purpose of FlexDx is to limit the computational cost of detecting and isolating faults. The basic idea is that only a subset of all the available tests are needed at any time, and by only running those tests that are needed at a particular time the computational cost will be reduced. FlexDx does this by initially selecting a set of tests that together can detect if there is a fault, but can't isolate the exact fault or faults. Every time one or more tests triggers an alarm, FlexDx will compute

a set of possible fault diagnoses based on the current set of possible diagnoses and the faults that the triggered tests can detect. In other words, it will refine the set of possible diagnoses as new tests are run and indications of faults are found. FlexDx then selects the next set of tests to run based on the refined set of diagnoses, estimates the last fault free time, and replays all sensor data starting from that point in time.

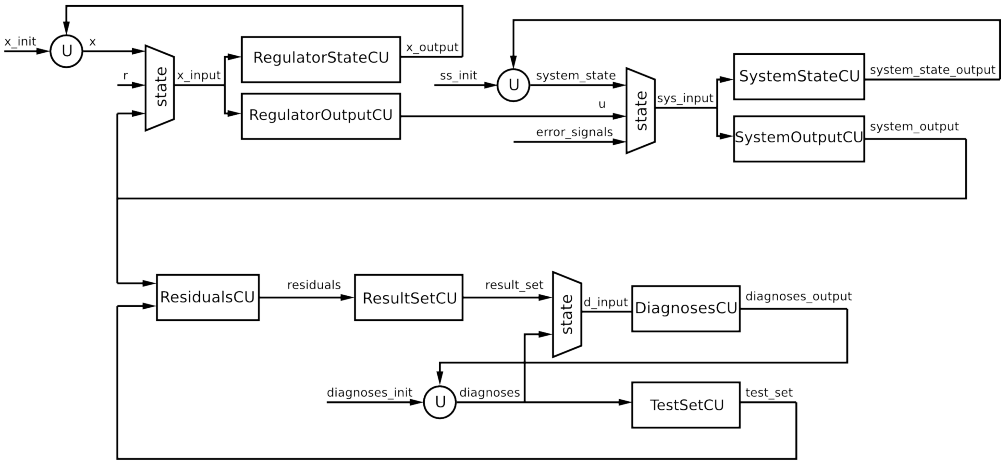

Figure 2.1: A graph representing the DyKnow implementation of FlexDx. Arrows represent streams. Rectangles are computational units that transform data; such as DiagnosesCU which takes the stream d_input, consisting of the current set of possible diagnoses and the new test results, and computes a new refined set of possible diagnoses. A trapezoid synchronizes multiple streams. A circle containing a U creates a stream that is the union of two input streams. It is used here to provide initial values for streams, which are given by the input streams x_init, ss_init, and diagnoses_init.

2.4 SIGNAL

SIGNAL is a synchronous programming language developed at Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA)[1] [6][7]. Like other synchronous languages SIGNAL have a strong mathematical foundation and strict coding style that makes it possible to perform formal code analysis and validation which makes it suitable for real-time systems.

IRISA has also developed a suite of free tools to use for SIGNAL development. Among these is the POLYCHRONY compiler that was used during the thesis. The

[1] `http://www.irisa.fr`

compiler validates SIGNAL code and generates low level code in one of three available target languages, C, C++ or Java, that then in turn can be compiled and executed.

This description of the SIGNAL language will only cover the areas used in the thesis and those that are needed to understand the example code. For a formal and complete description of the language see the SIGNAL reference manual [8].

The fundamental parts of SIGNAL are signals, clocks, relations and processes.

2.4.1 Signals and clocks

A signal is a totally ordered stream of typed values, similar to the streams in DyKnow but carrying only values instead of time stamped samples. Every signal has an abstract clock[2] that provides the set of instances where the signal is present, i.e. has a value. In the signal traces the symbol $\perp$ is used to represent signals that are not present. Unlike mono-clocked synchronous languages like LUSTRE [9] there is no global master clock in SIGNAL. All signals have their own clocks that are independent of each other as long as there are no constraining relations.

While the theoretical foundation of SIGNAL is not needed to understand this thesis, a few things should be said about the hierarchy of clocks. The way SIGNAL handles clocks is one of the most important aspects of the language. To create a meaningful program the clocks of signals are constrained using relations. Because signal clocks can be thought of as sets of instances where a signal is present, and the constraints defined by relations as set operations, all clocks will be arranged into hierarchies that can be represented as a collection of overlapping trees. The clocks are organized by set inclusion so that all clocks in a tree are subsets of the tree's root clock. If two trees overlap then there is subset of the two trees that are shared, and which also can be seen as a tree.

Figure 2.2 shows a simple example of such a hierarchy, consisting of four trees with the root clocks A, B ,C and D. The tree of D fills the overlapping area between B and C, while both B and C are subsets of A. This means that the set of instances where a signal with the clock D is present is equal to the intersection of the sets of B and C. In SIGNAL this would mean that a signal S_B with the clock B can be present, i.e. have a value, while a signal S_C is not and vice versa, and that both can be present at the same time. That S_B and S_C will only be present if and only if S_A is present, and that S_C will be present if and only if both S_A and S_B are present.

This hierarchy could, for example, represent a system where S_A carries an integer value n, where S_B and S_C will be present when n is a multiple of 3 and 5 respectively, and that the output signal S_D will be present when both S_B and S_C are present, i.e. when n is a multiple of 15.

[2]Referred to simply as the clock of the signal from now on.

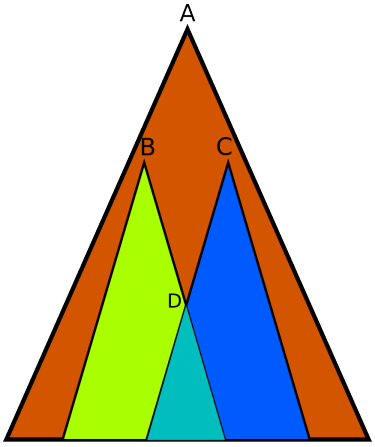

Figure 2.2: A simple clock hierarchy with four trees.

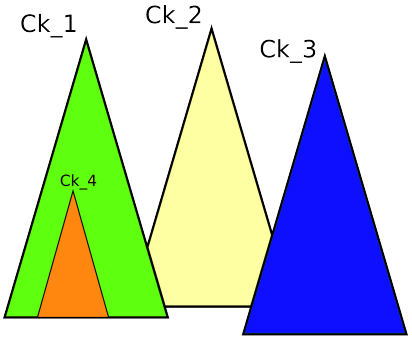

Figure 2.3: An exochronous clock hierarchy.

If the clocks can't be organized into a single tree it is called an exochronous hierarchy. Figure 2.3 shows an example of such a hierarchy. An exochronous clock hierarchy represents a system where multiple root clocks are independent of each other.

The other kind of clock hierarchy, an endochronous hierarchy, has a tree that contains all other trees, meaning that there is a clock that includes every other clock. This is the case with the master clock in LUSTRE. However, given the static and well defined nature of clocks in SIGNAL, any exochronous clock hierarchy can be transformed into an endochronous hierarchy [3, Ch. 9]. Even if clocks in SIGNAL can be independent of each other, and from the programmers point of view there is no master clock, a single hierarchy with a master clock can be created. Figure 2.4 shows the clock hierarchy of figure 2.3 transformed into an endochronous hierarchy. Here the clock Ck is the union of all other clocks, and a signal with clock Ck will

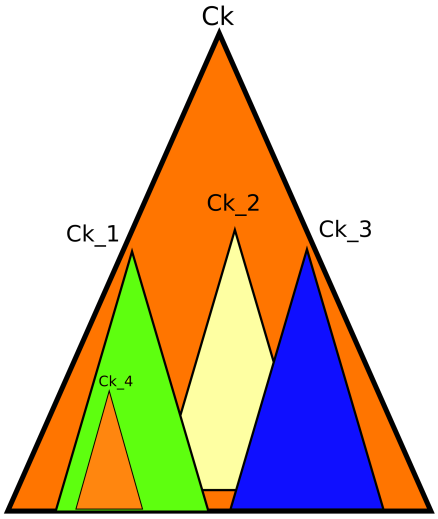

Figure 2.4: An endochronous clock hierarchy.

therefore always be present.

All clock relations are statically defined and will be known at compile time. The compiler will then analyze the clock hierarchy and check that it is valid. This way of handling clocks have two particularly interesting consequences. The first is that it is possible to say, before running the code, which signals will be present at any point of time. The second is that, since the compiler knows how every clock is related in the clock hierarchy, a SIGNAL program can "resynchronize" input data that is received asynchronously, as long the values for each signal is received in the correct order [3, Ch. 11]. SIGNAL can therefore be used to build components in a globally asynchronous, locally synchronous (GALS) systems, where synchronous components communicate over a asynchronous network.

2.4.2 Relations

SIGNAL code is declarative and built from equations where operators are relations between two or more signals. There are two kinds of operators, normal signal operators and clock operators. Normal operators can in turn be divided into two types. Monoclock operators and multiclock operators.

Signal operators A monoclock operator creates an implicit clock constraint so that all signals involved must, and will implicitly get, the same clock, while with a multiclock operator the signals can have different clocks. For example, the addition operator + is a monoclock operator that uses three signals. The expression

`x := y + z` states that the values of the signal x is the sum of the values of signal y and z, and that x, y, and z have the same clock. This means that all three signals will be present at the same time, as can be seen in the following signal trace.

x:	5	11	17	⊥	23	29	...
y:	1	3	5	⊥	7	9	...
z:	4	8	12	⊥	16	20	...

All operators "return" a signal so it is possible to combine operators to create nested expressions. It is for example possible to replace z in the previous example with `(z + w)` to form the expression `x := y + (z + w)`. This can be done anywhere a signal is expected.

The `when` operator is an example of a multiclock operator. The operator is used for undersampling, that is, creating a signal with a clock that is a subset of another clock. For example, the expression `x := y when z` states that x is equal to y when z is true. x and y must have the same type, and z be of boolean type. It does not add any restrictions on the clocks of y and z which means that all three signals can have different clocks, as the following signal trace shows.

x:	1	⊥	⊥	4	⊥	⊥	7	...
y:	1	2	3	4	⊥	6	7	...
z:	T	F	⊥	T	T	F	T	...

Five particularly important operators that are used frequently in the thesis is the delay operator `$`, the `window` operator, the memorization operator `var`, the merging operator `default`, and the `count` operator.

`$` is used to "delay" a signal, meaning that it will give an earlier value of the signal. For example `x := y $ 1` states that x shall have the previous value of y and the same clock as y. The argument to `$` is a positive constant integer and can be left out, in which case it will use the default value 1.

x:	1	2	3	4	5	6	...
x $:	0	1	2	3	4	5	...
x $ 2:	0	0	1	2	3	4	...

`init` is used to define initial values of a signal when using delays. When using an argument to `$` that is greater then 1 it is also possible to provide multiple initial values using an array of the form $[x_1, x_2, \ldots, x_n]$. If the `init` part is left out or too few initial values are provided a type dependent default value, 0 in the case of integers, will be used as the initial value.

x:	1	2	3	4	5	6	...
x $ 1 init 1:	1	1	2	3	4	5	...
x $ 2 init [-2, -1]:	-2	-1	1	2	3	4	...

`window` is another way to get previous values of a signal. It is used in the same way as $ but instead gives an array of previous values, including the current value.

x:	1	2	3	4	5	6	...
x window 2:	[0, 1]	[1, 2]	[2, 3]	[3, 4]	[4, 5]	[5, 6]	...

`var` is a multiclock operator used to get the latest value of a signal, independent of the clock of the signal. `x := var y` states that x shall have the latest value of y, but the clock of x is independent of the clock of y and must be specified elsewhere. This is very useful when you need the value of a signal but do not want to be constrained by its clock.

x:	⊥	1	⊥	⊥	3	3	⊥	⊥	⊥	5	6	6	...
y:	1	⊥	2	⊥	3	⊥	4	⊥	5	⊥	6	⊥	...

`default` is used to merge two signals. For example, `x := y default z` states that the clock of x is the union of the clocks of y and z, and that x shall have the value of y if y is present, otherwise it will have the value of z.

x:	-1	1	⊥	2	-3	3	⊥	4	...
y:	⊥	1	⊥	2	⊥	3	⊥	4	...
z:	-1	-2	⊥	⊥	-3	-4	⊥	⊥	...

`count` is used to count the number of occurrences of a signal modulo N, where N is a constant argument. For example, `x := i count 3` will count the number of signals in i from 0 up to 2, and then start again at 0. This is a monoclock operator so x and i will have the same clock.

i:	1	1	⊥	2	3	⊥	5	8	...
x:	0	1	⊥	2	0	⊥	1	2	...

Clock operators The clock operators are used to explicitly specify relations between signal clocks. These consist of the synchronization operator `^=` and the set operators `^+`, `^*` and `^-`.

`^=` states that the left hand signal and the right hand signal have the same clock, i.e. are synchronous.

`^+` creates a union of two clocks. For example, `x ^= y ^+ z` defines the clock of x to be the union of the clocks of y and z.

`^*` creates an intersection between two clocks. For example, `x ^= y ^* z` defines the clock of x to be the intersection of the clocks of y and z.

`^-` creates a relative complement of two clocks. For example, `x ^= y ^- z` defines the clock of x to be the set difference of the clocks of y and z.

2.4.3 Processes

SIGNAL code is organized using processes. The following code shows an example of a process consisting of two parts; the interface and the body. The process takes an input signal i and, at every n:th instance of i, outputs the sum of the last n values of i.

```
1  process sum_n =
2  {integer n;}
3  ( ? integer i;
4    ! integer s; )
5  (| old_i := i $ n
6   | zsum := sum $ 1
7   | sum := zsum + i - old_i
8   | cnt := i count n
9   | s := sum when cnt = n - 1
10  |)
11 where
12   integer zsum, sum, old_i, cnt;
13 end
```

The following signal trace shows input values and output values of the process with n set to 3.

i:	8	5	6	5	7	5	4	4	4	...
s:	⊥	⊥	19	⊥	⊥	17	⊥	⊥	12	...

The interface includes the name of the process (line 1), static parameters that are provided when compiling (line 2), input signals (line 3), and output signals (line 4). The body includes the code that describes its behavior (line 5 to 10) and local signal declarations (line 12).

There are additional details regarding processes that are important to know when writing SIGNAL programs, but those are not needed to understand this report. See the SIGNAL reference manual for a more complete definition of processes [8, Ch. 4].

2.4.4 Types

SIGNAL supports a rather typical set of data types. There are integers, reals of both single and double precision, complex numbers, booleans, characters, strings and static arrays. There is no support for dynamic arrays, i.e. resizable arrays. The size of an array must either be hard coded or set with a static parameter when compiling and cannot be changed at runtime. There is, in addition, an implementation dependent largest supported string length.[3] This means that all SIGNAL types can be statically allocated, with the exception of external types which are handled differently.

A less typical data type is the event type. This type is used solely to represent signal clocks. The clock of a signal can be extracted using the `^` operator, as in `x := ^y`, where the signal x is of the event type. A signal of this type will always have the value true.

y:	1	⊥	2	3	⊥	4	...
x:	T	⊥	T	T	⊥	T	...

There are also two types of tuples, the monoclocked structure and the multiclocked bundle.[4] A structure is a collection of synchronous elements that can be of different types, very much like the `struct` type in C, while a bundle is a generalization of a structure where the elements have independent clocks and will at any instance, like normal signals, either hold a value or be absent.

Finally, it is possible to define external types using `type NAME = external`, where NAME is the name of the new external type. When defining a signal to be of this type the compiler will generate code that uses the signal but leaves the type undefined. It is then up to the programmer to define the type in the chosen target language. SIGNAL can't do any operations on signals with an external type other

[3] There is no check for this in POLYCHRONY version 4.16 so it is up to the programmer to make sure that it is not exceeded.

[4] The bundle type is not implemented in POLYCHRONY version 4.16.

than clock operations. The only way signals of external type can be used is to pass them to external processes.

2.4.5 External processes

An external process is a method used in SIGNAL to, among other things, import code written in other languages. It is an abstraction of a process where the interface, including the clocks of input and output signals and any dependencies between them, are specified. An example of such a dependency is when an output signal of an external process is computed from an input signal, and where both signals have the same clock. Because the external code is not declarative, external processes must often be called in a certain order. The signal dependencies are needed by the compiler when it generates low level code so that it knows in what order computations and calls to external code can be made, and what computations can be made concurrently.

External processes are used in the exact same way as normal processes and make it very easy to use code written in the target language. The compiler will generate code that declares and uses these external processes, but it leaves their definitions blank. The programmer then has to implement the processes in the target language and make sure that the code complies with the specified interfaces.

External types and processes are particularly useful when types or capabilities that are not supported by SIGNAL are needed. A good example of this are dynamic arrays. Since SIGNAL does not support dynamic arrays the programmer can instead add an external dynamic array type implemented in the target language and then have external processes to add, remove, or in other ways manipulate data in the arrays.

2.4.6 Oversampling

Just as it is possible to define a signal whose clock is a subset of the clock of another signal using undersampling, it is also possible to use oversampling to define a signal whose clock is faster than, i.e. a superset of, that of another signal. It is for example possible to define the clock A of a signal S_A such that the signal produces a fixed number of values for every instance of another signal S_B. This can be used to create a procedure that for every instance of its input signal produces five instances of its output signal.

It is also possible to let the clock of a signal depend on the value of another signal. The following example shows a procedure that uses this to create a counter.

```
1 process count_from_n =
2 ( ? integer n;
```

```
3   ! integer i;
4 )
5 (| cnt := n default (zcnt - 1)
6  | zcnt := cnt $ 1 init 0
7  | n ^= when (zcnt = 0)
8  | i := zcnt when zcnt > 0
9  |)
10 where
11     integer cnt, zcnt;
12 end;
```

`count_from_n` is a simple process that functions as a counter. The process takes as input an integer signal that holds the number to count from, and has an integer signal as output that counts from this value down to zero. As can be seen in the following signal trace the output signal `i` has more instances than the input signal `n`.

n:	2	⊥	⊥	3	⊥	⊥	⊥	2	⊥	⊥	...
cnt:	2	1	0	3	2	1	0	2	1	0	...
zcnt:	0	2	1	0	3	2	1	0	2	1	...
i:	⊥	2	1	⊥	3	2	1	⊥	2	1	...

The ability to do this is very important as there is no proper iteration construct in SIGNAL. All existing iteration constructs must have a predefined number of iterations that is set at compile time. They are useful when one need to iterate over static arrays, but they are of little use when implementing iterative algorithms. The alternative that SIGNAL provides is to instead use oversampling. The next example shows a simple process that uses this to compute factorials.

```
1  process factorial =
2  ( ? integer n;
3    ! integer f;
4  )
5  (| zi := i $ 1 init 1
6   | i := n default zi - 1
7   | f := acum when i <= 1
8   | acum := 1 when n = 0
9                 default n
10                default acum $ 1 * i
11  | n ^= when zi <= 1
12  |)
```

```
13 where
14    integer i, zi, acum;
15 end;
```

n:	5	⊥	⊥	⊥	⊥	2	⊥	...
i:	5	4	3	2	1	2	1	...
zi:	1	5	4	3	2	1	2	...
acum:	5	20	60	120	120	2	2	...
f:	⊥	⊥	⊥	⊥	120	⊥	2	...

Chapter 3

The SIGNAL implementation

This chapter explains how the SIGNAL implementation of FlexDx was constructed, tested, and how the more interesting parts works. Because of the limitations of SIGNAL, such as the lack of dynamic arrays, a significant part of the final system had to be written using other languages. The chapter is therefore divided into three parts. The first covers the core of the system written in SIGNAL, the second external code written in C++ and Matlab, and the last the method used to test the system.

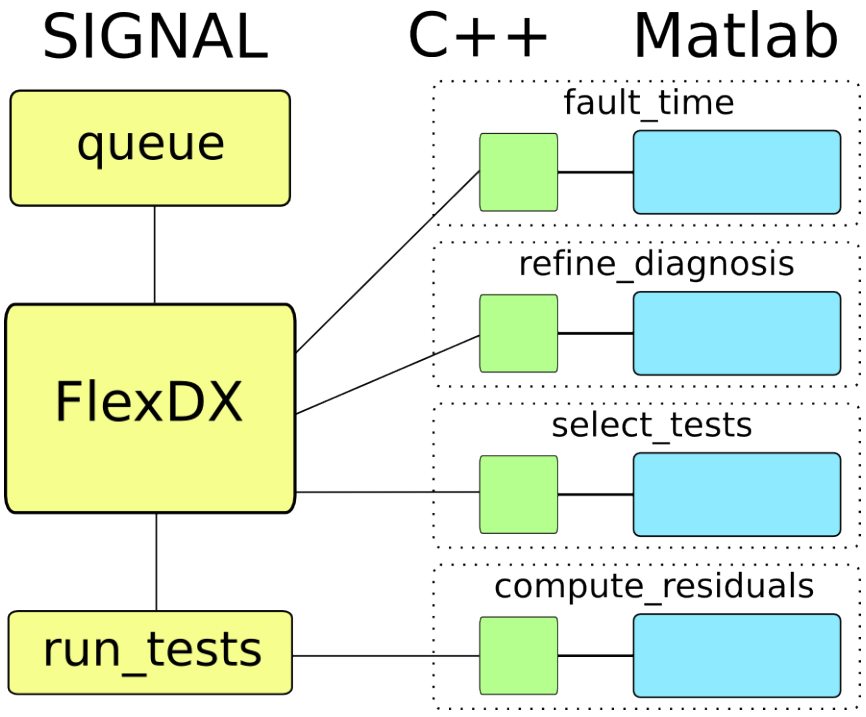

Figure 3.1: A simplified representation of the SIGNAL implementation.

3.1 SIGNAL code

The most important part for this thesis is the core of the system written in SIGNAL. This part connects the external processes that performs computations and is responsible for receiving and storing sensor signals. It forms a static system of external processes and signals quite similar to the network of computational units and streams used in DyKnow.

One particularly important part of the system, and a good example of how SIGNAL can be used, is the sensor queue. As was mentioned in section 2.3 it must be possible to store and use previous sensor signals when new tests are selected, and how far back it must go depends on how long it takes to detect a fault.

This is where the signal queue is used. Its task is to store sensor signals, go back and replay previous signals when needed, and to overwrite stored signals when they are no longer needed.

Action	Argument	Result	Stored signals				
IN	2	-	-	-	-	[2]	
OUT	-	2	-	-	-	2	[]
IN	7	-	-	-	2	[7]	
OUT	-	7	-	-	2	7	[]
IN	1	-	-	2	7	[1]	
OUT	-	1	-	2	7	1	[]
DEC	3	-	-	[2]	7	1	
OUT	-	2	-	2	[7]	1	
OUT	-	7	-	2	7	[1]	
OUT	-	1	-	2	7	1	[]
IN	8	-	2	7	1	[8]	

Table 3.1: An example of how the queue is used. The signal that the read index currently points to is surrounded by square brackets. The queue will output stored signals until the index no longer points to a stored signal, as can be seen when new signals are added to the queue and when the read index is decremented.

```
process queue =
{ integer SIZE; }
(? data sensor;
   integer dec_time;
 ! data read_s;
   integer read_time, write_time;)
(| last_actual := actual_time $ 1 init 0
 | last_current := next_current_time $ 1 init 0

 | read := when last_current < last_actual
 | write := when last_current >= last_actual

 | read_time := current_time when read
 | write_time := actual_time when write

 | actual_time := 1 + last_actual when
                  write default last_actual
 | current_time := last_current - dec_jump when
                  last_current >= dec_jump default 0
 | next_current_time := current_time + inc_read

 | dec_jump := dec_time default 0
 | inc_read := 1 when read default 0

 | buffer := sensor window SIZE when write
 | var_arr := var buffer
 | var_actual := var actual_time

 | read_s := var_arr[SIZE - (var_actual - read_time)]

 | sensor ^= write
 | var_arr ^= var_arr ^= read ^= read_s
 |)
where
  integer actual_time, current_time, last_current,
          next_current_time, last_actual, dec_jump,
          inc_read, var_actual;
  [SIZE] data buffer, var_arr;
  event read, write;
end;
```

Unfortunately SIGNAL has few ways of storing signal values. One way is to use delayed signals to get the N:th previous value, where N is a constant integer greater than zero, but this would clearly not work in this case as N must be known at compile time. Another method is to use the `window` operator to get an array with the last N values of a signal, where N, as before, is a constant integer greater than zero. This is still far from perfect as the array must be of constant size, but the elements of the array can be accessed with a non-constant index. This makes it possible to iterate over the array.

The example system used to evaluate the SIGNAL implementation was simple enough that the `window` method was sufficient, and it was used to build a queue with a read-index that can be decremented to "replay" previous signal values. Every time a value is pushed onto the queue the read index is decremented, every time a value is read from the queue the index is incremented, and when a new test selection is made the index is decremented by the number of sensor values that shall be replayed. This is not hard to do with SIGNAL, but it gets more complicated when oversampling is added as the clock relations then becomes rather complex.

Oversampling makes it, as was explained in section 2.4.6, possible to define a signal with a higher clock rate than that of the input signals to the queue process. The queue process uses this to give the output signal a higher clock rate than the input signals when stored values are requested. The queue will normally output stored signal values as soon as they are added to the queue, but when old values are requested the queue will use oversampling to output them before the next instance of the input signals. The stored values will be returned, one at a time, until the read index has reached the end of the queue. Only when this is done will it again start receiving new signal values to add to the queue.

Because oversampling is used this can be done without any special input signals that tells the process when to read from, or write to, the queue. However, as was mentioned earlier this is not a perfect solution as the queue can only hold a constant number of signal values, but as long as the chosen constant is large enough it works well.

A number of alternative methods of doing this was also tested. The most obvious method, and which was also used until the clocks needed for oversampling could be figured out, was to set a ratio between reads and writes, for example one write for every ten reads. It was a very simple solution, but it created some problems that are avoided with oversampling. Most important is that the ratio would have to be carefully decided on as it is essential that no sensor values are lost while they are still needed. If the ratio is set too low stored values could be removed from the queue to early, while a too high ratio would incur performance costs as every unnecessary read creates some overhead.

3.2 External code

The algorithms in FlexDx require dynamic arrays and more flexible iteration constructs than are available in SIGNAL. Some parts of FlexDx therefore had to be implemented with other languages using external processes and types. It would have been possible to avoid this problem by not using the actual algorithms and instead write simple functions that return hard coded results. This would still show if it is technically possible to implement FlexDx in SIGNAL, but it would not have said anything about how hard it would be. It would have simplified the task greatly, but because a DyKnow implementation already existed and its Matlab code was available it seemed a better idea to instead write an implementation that uses the existing code for the algorithms. This approach makes it possible to perform a better comparison between the implementations as they use the exact same algorithms and computes results the same way, and it provides an example of the possible flexibility of SIGNAL and how easily it can be extended using external code.

Two problems had to be resolved for this to work. The first, and easiest, problem was to actually be able to call Matlab code. This was done by integrating a GNU Octave[1] interpreter into the C++ part of the system. GNU Octave is mostly compatible with Matlab and the existing code could therefore be used with almost no changes. The second problem was that the functions written in Matlab uses matrices and arrays of variable size that must be stored and used by other parts of the system. Since SIGNAL does not support such data types they had to be implemented using C++ and external processes and types. This is the purpose of the C++ part of the system, to implement the data types needed, act as an interface to the Matlab code, and perform type conversions between the types used by the different languages.

3.3 Testing the implementation

To test that the SIGNAL implementation is working correctly an existing example scenario was used that included several sets of input signals containing various errors, parameter values for the algorithms, and expected results [5, Sec. 6].

Because the purpose of the test was to see if SIGNAL could be used for a task of this nature the actual results, and their correctness, were not very important. What mattered was that the implementation could be constructed using SIGNAL, not that it provides perfectly correct results. As long as it can be shown that all parts of the implementation are called and produce some results, it will be clear that SIGNAL is capable of performing this task. Therefore only one set of input signals was used. Enough to see that all parts of the system are working.

[1] `http://www.gnu.org/software/octave/`

	DyKnow implementation				SIGNAL implementation			
	t_f	t_a	Diagnoses	Active tests	t_f	t_a	Diagnoses	Active tests
1	0	101.9	NF	1,2,**5**	0	101.7	NF	1 2 **5**
2	98.9	101.9	1,3,5,6	1,3,10,**13**	99.2	104.5	1,3,5,6	1,3,10,**13**
3	98.6	102.2	1,3,25,26,45,46	1,2,6,7,**8**,11,**12**	99.6	106.1	1,3,25,26,45,46	1,2,6,7,**8**, 11, **12**
4	98.4	101.4	1,25,26,35,36,45	1,2,**6**,7,**9**,10,11	99.7	108.3	1,25,26,35,36,45	1,2,**6**,7,**9**,10,11
5	98.5	102.1	1,26,36,45	1,2,7,10,**11**	99.8	112.9	1,26,36,45	1,2,7,10,**11**
6	98.8	-	1,26,36	1,2,7,10	101.2	-	1,26,36	1,2,7,10

Table 3.2: The results from running the SIGNAL and DyKnow implementation of FlexDX with the same sensor data.

Table 3.2 shows the results from the SIGNAL and DyKnow implementations. t_f is the last fault free time. t_a is the time when one or more tests triggered an alarm. `Diagnoses` is the current set of possible diagnoses. `Active tests` is the set of tests that were used; tests that triggered an alarm are shown with bold face. A fault diagnosis consists of one or more faults that might have occurred. Each fault is represented with a single digit. The fault diagnosis 1 indicates that fault number 1 can explain the failed tests, and 23 that fault number 2 and 3 together could be the cause.

As can be seen in the table the results from the SIGNAL implementation matches the expected results perfectly in all but fault times and when alarms are triggered. It is still clear from these results that all parts of the implementation must be working as intended. The discrepancies in timing is likely caused by small differences, or possibly bugs, in the implementation of the fault time algorithm. Because these differences does not affect any results relevant to the purpose of the thesis, and because time was limited, the code was not corrected.

Chapter 4

Discussion

This thesis shows that FlexDx can be implemented both asynchronously using DyKnow, and synchronously using SIGNAL with external processes. However, there are several differences between SIGNAL and DyKnow that affect these implementations and the task of constructing them. This chapter discusses those differences, what the practical implications are, and the suitability of SIGNAL for this task.

4.1 Differences between SIGNAL and DyKnow

The first thing that should be noted is the different purposes of SIGNAL and DyKnow. DyKnow is a middleware designed to be used for advanced knowledge processing, while SIGNAL is a synchronous programming language designed for real-time systems where timing properties and reliability are most important. Both uses streams; DyKnow uses streams for communication between computational units, and SIGNAL, which uses the dataflow paradigm, essentially uses streams instead of variables. There are some differences in how the streams are used and in what they can do. Streams in DyKnow are, for example, more flexible than signals in SIGNAL. However, FlexDx does not need to use much of this added functionality and there are therefore few practical differences between streams and signals in this particular case.

SIGNAL, with its background in real-time systems, is very restrictive and does not directly support many things that can usually be done in typical imperative programming languages. DyKnow is far more flexible. First of all; DyKnow is used together with a supported programming language, for example powerful languages like C++ and Java. Secondly; the structure of a system built using DyKnow is far more flexible than that of one written in SIGNAL, since it with DyKnow is possible to add and remove computational units at runtime, change the connections between them, and change the policies of the streams. With SIGNAL everything

must be known at compile time. Sizes of arrays, the number of iterations in loops, procedures, signals and their clocks; all must be specified at compile time and can't be changed later. It is not possible to change the structure of a system built with SIGNAL in the same extent that is possible with DyKnow. This means that, while SIGNAL is well suited for time critical applications because of the predictable and safe code it produces, it is also far more restrictive than DyKnow and tasks that can be performed with DyKnow can be hard or impractical to perform using SIGNAL.

An example of this could be seen while implementing FlexDx. FlexDx need the ability to add and remove tests when a new test selection is made. A natural way to do this would have been to represent tests as processes that are added and removed at runtime, the same way as computational units can be added and removed in DyKnow. This is not possible in SIGNAL. Fortunately there is a simpler method that can be used. Since the tests only consists of matrices, and results and states are all computed with matrix multiplications, simple arrays of matrices is enough to represent the tests.This is the method actually used in the DyKnow implementation, and even if it had been possible to use removable processes, this simpler method is more appropriate. However, due to the lack of dynamic arrays, even this can be somewhat difficult to implement.

The lack of dynamic arrays is a serious limitation of SIGNAL, and was the most problematic obstacle when implementing FlexDx. The algorithms that are used in the DyKnow implementation relies heavily on such arrays[1], both internally and to hold the resulting data which the rest of the system must be able to use. One possible solution to this problem is to instead use static arrays that are large enough to hold all data that is expected to be needed. This was used in early prototypes of the system.

Such a solution might work when using hard coded functions instead of the actual algorithms, or if a very simple system is monitored; where only a handful of tests are needed and faults are detected quickly so that few sensor signals and results must be stored. It is not, however, a practical solution for more complex systems. It would be necessary to estimate how large the arrays must be to hold all data that will be needed during execution, which depends on the monitored system and its output. Even if that is possible it would not be compatible with the purpose of FlexDx, which is to decrease the computational cost of finding faults by only running a subset of all tests at any time. It will only run the tests that are needed at the moment, and the amount of resources used will vary depending on the tests. This would not be possible using static arrays as the amount of memory used would be constant, and because the iteration constructs that are needed to traverse the arrays must have a predefined number of iterations, i.e. the size of the

[1]To be more precise they use matrices of arbitrary size, but in SIGNAL there is no distinction between a matrix and an array.

arrays, it would also create a computational overhead.

SIGNAL does however allow the programmer to easily add additional functionality using external processes and types that does not need to adhere to most of the restrictions of SIGNAL. As long as the code used with external processes respects the synchronous nature of SIGNAL and the clocks of their input and output signals, this will allow the system to use the far more powerful capabilities of C, C++ or Java. This makes it possible to add dynamic arrays, proper iteration constructs, and other things that are not supported in SIGNAL. However, by doing this the programmer is making a compromise between the flexibility and power of these languages, and the analyzable and safe code that SIGNAL produces.

4.2 Using SIGNAL

The strict coding style using relations that makes SIGNAL suitable for real-time systems also makes it cumbersome to use. Writing the SIGNAL code of the FlexDX implementation required far more effort than the parts written in C++. Part of this was undoubtedly caused by a lack of experience in using SIGNAL, but it was mostly due to how the POLYCHRONY compiler handles coding mistakes and how easy they are to make.

Possibly the most complicated part of writing SIGNAL code was the specifications of the clocks. For a small program this was not hard, a handful of signals and their clocks are easy to keep track of, but when more advanced uses of signal clocks are needed, for example when using oversampling, the limitations of the compiler starts to become a problem.

The POLYCHRONY compiler does not report errors well. Only syntax errors, and to a lesser degree cyclic clock dependencies, are reported in a clear way that makes them easy to find and fix. Syntax errors are reported directly by the compiler with both the type of error that has been found and a line number that is usually correct. Cyclic clock dependencies are reported almost as clearly with a list of signals that are part of cyclic dependencies. This does not say much about where the problem lies or how to fix it, but given this information it was never hard to find the causes of such errors.

Inconsistent relations between signals and clocks were far harder to debug, and more common. When the compiler detects such errors it will skip the code generation phase and add warnings that explains why the compilation failed to the intermediary output files that the compiler produces. These files have all the information needed to find the errors, but they are not easily parsed. They contain a more detailed representation of the code where all clocks are represented as signals, and implicit clock restrictions are made visible as relations between such signals. These files are very helpful when you know how to read them and how to spot common

errors. Unfortunately the warnings give very little information about the errors that were found and will not say why an error occurred. This must be deduced by the programmer using his knowledge of SIGNAL and the different warnings that the compiler might add.

To make matters worse, signals, and especially their clocks, are very sensitive to inconsistent relations. Almost every line of code in SIGNAL creates relations between signals and clocks, and all such relations must be consistent with all other relations or the code will not compile. This makes such errors very easy to make, and because multiple signals, clocks, and relations are involved they are often hard to find and fix. This means that the programmer must be very careful not to make any mistakes when writing SIGNAL code.

It can of course be argued that this actually makes programming easier in the long run as most errors are found early by the compiler. This is probably true, at least when the programmer is experienced enough to fully understand the intermediary files. It also means that the programmer must know, almost completely, how a program shall function and how all signals and clocks are related, before any code is written. This too can be a good thing since code must be well thought through to be accepted by the compiler and it therefore forces the programmer to plan the code well, but it also makes it harder to experiment with solutions to problems or explore the capabilities of the language. Design be trial and error is not practical with SIGNAL; code that has not been well planned will only work by accident.

There were also some issues with using external processes. The problem is that external processes must follow the clocks specified in their interfaces for input and output signals, but the code that is generated does not have any means of providing the imperative functions that the programmer must implement with information about these clocks. This is not a problem when using simple processes with fully synchronous input and output signals, i.e. all signals have the same clock and are always present, as was the case with all external processes used in the SIGNAL implementation, but when they have different clocks it becomes more complicated.

The imperative function that implements an external process will have a parameter for every input and output signal of the SIGNAL process, and is called every time one of these input and output signals is present, but the function is not told which of these input and output signals are present when it is called. It has no way of knowing which input signals hold values, or when to compute output values. The external code must therefore get this information some other way. One simple method would be to use separate boolean signals that represent the clocks of each input and output signal. If a signal is present its boolean signal holds the value true, and if it is not present it holds the value false. This is not hard to do and would give the external process all clock information it needs, but it is cumbersome and could easily have been done by the compiler.

4.3 The limitations of SIGNAL

While the usability of the compiler and easy debugging is important, the limitations of SIGNAL caused problems that are more important for the thesis. It quickly became clear that the limitations of SIGNAL makes it to difficult to create a fully functioning implementation of FlexDX using only SIGNAL in the time available. The most interesting parts of the system could be written in SIGNAL, such as the signal queue, but the rest needed dynamic arrays, and methods to iterate through them. It is possible that more parts could have been written in SIGNAL by using static arrays and oversampling instead of loops, but I could not find a practical way to do so. Fortunately SIGNAL's external processes and types could be used to easily overcome these limitations.

By using another language, in this case C++, it was possible to add both dynamic arrays and loops to the system with very little effort. There are however limitations to what can be done, or at least to what is practical to do, using external processes. SIGNAL is not designed to be used for the kind of dynamic systems that can be built with DyKnow. Even if external code is used the SIGNAL code that makes up the foundation of a program will always be static and limit the flexibility of the system. There is also a cost to using external code. While external processes can add functionality that is not supported in SIGNAL, the more external code that is used the less of a reason there will be for using SIGNAL at all.

External code can easily be used to implement dynamic arrays, loops, and external processes that could use them, but if FlexDX had needed more of the capabilities of DyKnow, such as adding and removing computational units at runtime, it would have been better to use another language than. If one wants a synchronous language that is flexible and allows a program to change at runtime then SIGNAL is not a good choice.

Chapter 5

Conclusions

As this thesis shows it was not possible to write a working implementation of FlexDX using only SIGNAL in the available time. The lack of dynamic arrays and proper iteration constructs made that far to difficult. This problem could however be overcome using external processes to add the needed functionality from other programming languages. With the ability to use external code and types SIGNAL became far more powerful and usable. While this worked well it was not a perfect solution as signals of an external type could not be used directly by SIGNAL code, with the consequence that a large parts of the system had to be implemented in C++.

While working on the new FlexDX implementation no real problems were encountered related to using a synchronous approach to knowledge processing. The practical differences in writing synchronous code with SIGNAL instead of a typical asynchronous code using, for example, C++, was relatively small. The declarative nature of SIGNAL and its relations had a far greater impact on the task than it being synchronous. The high level behavior of FlexDX could easily be described synchronously and all existing code could be called and used without any significant problems.

Writing SIGNAL code without making any mistakes with the relations between signals was difficult, but mostly because of how hard it was to debug code with the POLYCHRONY compiler. In those cases when the compiler worked well and there was no need for loops, dynamic arrays, or complex clock relations, there were usually very few problems.

SIGNAL might not be ideal for this task, but it is clearly powerful enough to be used instead of DyKnow in this particular case. SIGNAL would not be able to replicate all of the capabilities of DyKnow, and can definitely not replace DyKnow for more advanced systems. It can work well in a system that does not need the runtime reconfigurations that DyKnow provides, but it is not likely that SIGNAL would be sufficient when such capabilities are needed.

The lack of dynamic arrays and proper iteration constructs is a problem, and if SIGNAL had had support for such things it would have been possible, probably even practical, to implement FlexDX, including all algorithms, using only SIGNAL. Even without them, given enough time, it should still be possible to implement all of FlexDx using only SIGNAL, but it would be very difficult and the lack of dynamic arrays could still cause some problems.

5.1 Future work

It was not tested during the thesis, but it is likely that DyKnow and SIGNAL could work well together. SIGNAL is made for writing safe and predictable code based on a sound mathematical foundation that can be analyzed formally, but produces static systems and does not provide any means of communication in a distributed system. DyKnow on the other hand is used to build distributed systems where the structure of the system can be changed at runtime, and handles the communication between its components. It is very powerful and flexible, but it must be used together with a programming language that performs the work of the computational units. It is possible that SIGNAL could be used for this.

As was mentioned in section 2.4.1, SIGNAL can be used in GALS-systems where synchronous components communicate over a asynchronous network. This is exactly what we have. DyKnow creates a distributed system for knowledge processing that uses asynchronous streams to connect the components, while SIGNAL creates synchronous code that perhaps could be used in such components.

It would be very interesting to see if, instead of replacing DyKnow with SIGNAL, it would be possible to use them together and combine the many advantages and strengths that both offer.

Bibliography

[1] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *IEE Proceedings - Computers and Digital Techniques*, 152(2), 2004.

[2] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. *Proceedings of the IEEE*, 91:64 – 83, 2003.

[3] Abdoulaye Gamatié. *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification.* Springer, 2010.

[4] Fredrik Heintz. *DyKnow: A Stream-Based Knowledge Processing Middleware Framework.* PhD thesis, Linköpings universitet, 2009.

[5] Mattias Krysander, Fredrik Heintz, Jacob Roll, and Erik Frisk. FlexDx: A Reconfigurable Diagnosis Framework. *Engineering Applications of Artificial Intelligence*, 2010.

[6] Paul Le Guernic, Jean-Pierre Talpin, , and Jean-Christophe Le Lann. Polychrony for system design. *Journal of Circuits, Systems, and Computers*, 12(3), 2003.

[7] Abdoulaye Gamatié, Thierry Gautier, Paul Le Guernic, and Jean-Pierre Talpin. Polychronous design of embedded real-time applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(9), 2007.

[8] Loïc Besnard, Thierry Gautier, and Paul Le Guernic. *SIGNAL V4 – INRIA version: Reference Manual*, 2010. (working version).

[9] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The Synchronous Dataflow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305 – 1320, 1991.