Behavioral Intervals in Embedded System Design and Verification

Fabian Wolf, Dirk Ziegenbein, Rolf Ernst
Technische Universität Braunschweig, Germany

Abstract— Embedded system timing and power consumption are state and input data dependent. Therefore, formal analysis of such data leads to behavioral intervals rather than single values. These intervals depend on system concurrency, execution paths and states of processes, as well as target architecture properties. The paper presents an approach to modeling and analysis of process behavior using behavioral intervals. It considers the execution context, i.e. the current state and input of a process. We show how the results can be used to model complex process networks with system modes. The example of an ATM packet handler demonstrates significant improvements in modeling precision.

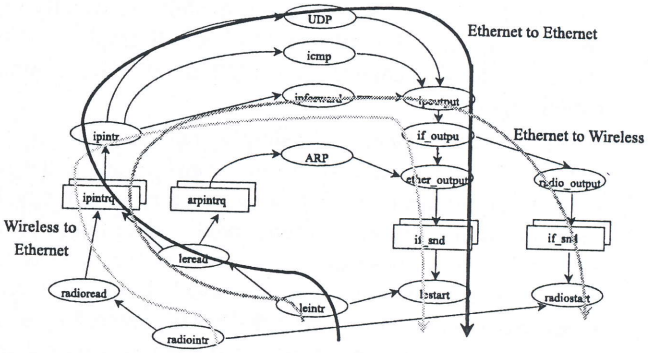

Figure 1: Packet dependent flow of execution in a base station

1 Introduction

Embedded systems typically include reactive and transformative functions, often described in different languages and semantics. A common representation called SPI (System Property Intervals) [12, 13] has been developed which permits the safe integration of different system parts and enables system optimization across language boundaries. SPI is based on the model of communicating processes, whose behaviors are described on a very high level by a set of parameters. The use of parameters like data rates, latency times, or an activation function enables the adaptation to different input models of computation.

The key point for the semantic flexibility of the model is the ability to specify the parameters not only in terms of an exact value but also as ranges of possible values, so called behavioral intervals. While the necessity for these intervals is straightforward for parameters like latency time, the extension of the behavioral interval concept to include also parameters like data rates, i. e. the amount of data consumed and produced at each process execution, enables a variety of different applications for the model. These include e. g. the representation of data-dependent communication or the integration of system parts in different design stages, ranging from possibly incomplete specification to legacy code.

The necessity to consider behavioral intervals for the design and verification of embedded systems becomes evident when looking at the limits of simulation. Profiling and simulation are the current practice in industry, but since exhaustive simulation is impractical, simulation results can only cover part of the system behavior, often with unknown coverage of worst and best cases due to partly unknown input data. Verification is a more complicated but attractive alternative. It provides lower and upper bounds reflecting data dependent control flow as well as data dependent statement execution. In the past, these bounds were very wide due to a lack of efficient control flow analysis and architecture modeling techniques. In the last few years, there has been significant progress in both areas such that formal behavioral interval analysis becomes practical.

Behavioral intervals depend to a certain extend on the process control flow which depends on the process input data. In other words, behavioral intervals of the processes and, hence, of the overall system are context dependent. Figure 1 gives an example. It shows a simplified set of processes of a pico cellular base station [5]. The solid lines represent the paths on which different data packets are routed through the process network. Important questions of the system architect can be the power consumption for sending a data package or the time to set up a connection in the base station. This should take the context into account, since for each packet type the processes react with a different control flow. Of course, simulation is always possible and statistical power and timing analysis are feasible, but the first approach is not reliable and the second is just an approximation of the complex hardware activities when executing the software of a base station. The analysis approach on source code level provides reliable and narrow behavioral intervals for context dependent process execution that is automatically evaluated by the analysis tool.

This paper is organized as follows: In section 2, the SPI representation model is introduced before we explain our path analysis approach for the determination of behavioral intervals in section 3. In section 4, intervals from data dependent instruction execution are explained. An example is presented in section 5 before we conclude in section 6.

2 The SPI Model

In this section, the basic concepts of the SPI (System Property Intervals) model are introduced, but the main goal of this section is to motivate the use of behavioral intervals. A more formal definition of SPI can be found in [12, 13].

In the SPI model, the system is represented as a set of concurrent processes which communicate via unidirectional channels that are either FIFO-ordered queues (destructive read) or registers (destructive write). Such models are usually represented as directed, bipartite graphs. A SPI graph consists of process nodes (P), channel nodes (C) and directed communication edges (E).

While each channel node simply transfers data from the sender to the receiver without any transformation, the functionality of process nodes can be of arbitrary complexity. But, the detailed internal process functionality does not have to be known for the purpose of optimization at the process level. Thus, processes and channels are modeled only by their abstract external behavior. This behavior is captured by a small set of parameters which are extracted from the original specification and associated with the graph nodes.

A parameter does not have to be specified as a single value but may be formulated as a behavioral interval that constrains the possible values for the parameter by a lower and upper bound. A main reason for the necessity of behavioral intervals for the purpose of correct modeling is the possible uncertainty about input data in combination with input data dependent control flow inside a process that make it impossible to find exact values for the parameters. Another advantage of behavioral intervals is the possibility to integrate system parts in different design stages in a single coherent representation. The system parts may range from possibly incomplete specification to legacy code.

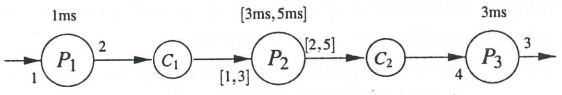

Figure 2: SPI Example

An example of a SPI model is depicted in Figure 2. At each execution, processes map input data to output data. However, since we are not interested in the function performed by a process, the communicated data is only represented by the amount of data which is important for communication scheduling or memory allocation. For example, process p_1 consumes 1 data token and produces 2 data tokens at each execution. The latency of p_1 (i. e. the difference between starting and completion time of p_1) is 1ms. Process p_2 is an example for a process that is specified using behavioral intervals, as it consumes at least 1 and at most 3 tokens from channel c_1 and produces at least 2 and at most 5 tokens on channel c_2, respectively. The execution latency is between 3ms and 5ms.

Mostly, the parameters of a process are not independent of each other but strongly correlated since they all depend on the executed program path inside the process. As we have discussed in the introduction, program paths are context dependent. To exploit context dependent behavior, we introduce process modes. Each mode thereby represents a subset of all possible program paths or external process behaviors. For example, process p_2 can be represented as having two alternative modes:

$$m_1 = ([3,3.8]\text{ms}, 1, 2)$$
$$m_2 = ([4.5,5]\text{ms}, 3, 5)$$

Then e. g. in mode m_1 process p_2's latency is between 3ms and 3.8ms, it consumes 1 token and produces 2 tokens, etc. Nevertheless, without specifying rules for the selection of a mode, the behavior of process p_2 is still uncertain since p_2 may execute in m_1 or in m_2.

Since the selected program path only depends on the process input data (and its state), a process can change its mode with every execution. Modes are local to a process and change as a result of process communication. While otherwise all data are abstracted to tokens with unknown values, the values of data which may change a process mode must be visible in SPI to be exploited. For this purpose, *virtual mode tags* may be associated to data tokens to represent data values. Thus, a process can select its mode depending on the presence of certain virtual mode tags. Therefore, an *activation function* is associated with each process that may be formulated as a set of rules. These rules map input token predicates to modes. A predicate in this case is either 'true' or 'false' depending on the number of tokens and the tag set of the first tokens on the input channels of the process. For process p_2 from the above example, these rules could be:

$$a_1 : (c_1.num \geq 1) \wedge (\text{'a'} \in c_1.tag) \mapsto m_1$$
$$a_2 : (c_1.num \geq 3) \wedge (\text{'b'} \in c_1.tag) \mapsto m_2$$

Assuming that process p_1 adds one of the tags 'a' or 'b' to the tag set of all produced tokens, the behavior of p_2 is completely determinate. If there is at least 1 available token on channel c_1 and if the tag 'a' is included in the tag set of this token, process p_2 is activated in mode m_1. Analogously, if there are at least 3 tokens available on c_1 and the first one has 'b' in its tag set, p_2 is activated in mode m_2. Thus, by adding knowledge about the input data, the behavioral intervals were substantially narrowed. This technique will be explained in more detail in the following section.

3 Behavioral Interval Determination

In this section, the determination of behavioral intervals of a process is described. A process implemented in software has a source code that can be analyzed. Input data can influence the control flow as well as the instruction execution on hardware level. When we investigate the control flow on the architecture independent source code level, analysis can be divided to path analysis and architecture modeling.

3.1 Path Analysis

For path analysis techniques [4] on source code level, a program is divided into basic blocks, where a basic block is a program segment which is only entered at the first statement and only left at the last statement [1].

Any program can be partitioned into disjoint basic blocks. Then, the program structure is represented on a directed program flow graph with basic blocks as nodes. Figure 3 shows an example. For each basic block a cost with respect to each behavioral interval is determined. Then, a longest or shortest path analysis on the program flow graph is used to identify a global behavioral interval. This procedure does not yet provide sufficient accuracy. For acceptable analysis accuracy one must identify feasible paths through a program. A feasible program path or trace is a path in this flow graph corresponding to a possible sequence of basic blocks when the program is executed, that means leading from the first to the last basic block of a program. A program segment PS is a segment of a program flow graph. This definition implies a hierarchy of program segments. A program path segment PPS is a path through a program segment. Not all paths in the program flow graph represent feasible program paths. A false program path is a path in the program flow graph which cannot be executed under any input condition.

False path identification is mandatory for programs with loops since loops correspond to cycles in the flow graph which lead to an infinite number of potential paths. The approaches by Mok [6], Puschner and Koza [9], Park and Shaw [8] require iteration bounds for all loops in the program which the user must provide by loop annotation. The approach by Gong and Gajski [3] can partially consider false paths because the user can specify the branching probabilities. While making formal analysis feasible, loop bounding alone is not sufficient for accurate path analysis. Nested loops are often interdependent and conditions depend on each other. These dependencies can be rather complex. Therefore, as a second step in [4] and in [8], the user is asked to annotate false paths. The number of false paths can be very large. Instead of enumerating false paths or, conversely, feasible paths, a language for user annotation with regular expressions is introduced in [8]. Still, the number of required path annotations can be extremely large in practice, as demonstrated with even small examples in [4]. A major step forward was the introduction of implicit path enumeration [4]. Here, the user provides linear (in)equations to define false paths. To evaluate these (in)equations, Li and Malik map the upper and lower bound identification to two ILP optimization problems, the one optimizing for the lower, the other one for the upper interval bound.

3.2 Local Path Cost

The execution time model in [4] is established as a standard model for static approaches. Here, the general term *cost c* is used since this analysis holds for many cost measures, such as timing, power consumption or data rates. The execution cost model is the sum–of–basic–blocks model. Let a program consist of N basic blocks with x_i execution count of basic block BB_i and c_i execution cost. Then, the sum–of–basic–blocks model defines for the total program execution cost C:

$$C = \sum_{i}^{N} c_i \times x_i$$

This model assumes that all executions of a basic block cost the same, which is true for data rates. However, data dependent instruction execution and superscalar or superpipelined architectures with overlapped basic block execution lead to widely varying local path cost with respect to latency time and power consumption. They have a substantial effect on the overall behavioral interval. For these common architectures, the sum–of–basic–blocks model cannot provide close bounds, but must be pessimistic to be correct. For higher accuracy, basic block sequences must be considered. This shall be called the sequence–of–basic–blocks model.

3.3 Path Classification

It is possible to exploit program properties to simplify path analysis for the determination of the sequence-of-basic-blocks. Large parts of typical embedded system programs have a single program path only. An FIR filter is a simple example and a fast fourier transform is a more complex one. In other words, there is only one path executed for any input pattern, even though this path may wrap around many loops, conditional statements and even function calls which are used for program structuring and compacting. A program has a **Single Feasible Path SFP**, when paths through the program are not depending on input data. A program segment with the SFP property is called an SFP–segment. Previous analysis approaches give more than one execution path for SFP programs because they do not distinguish between input data dependent control flow and program structuring aids. In the best case, they may be accurate but require much user interaction for SFP programs such as [4] and still do not deliver the sequence–of–basic–blocks. In case of SFP, simulation would choose the one correct path for any input pattern without further user interaction, but most practical systems also contain non–SFP parts. These have the multiple feasible path property MFP. A program segment has **Multiple Feasible Paths MFP**, when paths through the program are depending on input data. A program segment with the MFP property is called an MFP–segment. Isolation of SFP and MFP parts can help to exploit the SFP property and the resulting sequence in all programs. To apply different techniques to MFP and SFP parts, disjoint program partitioning is needed.

3.4 SFP Identification and Clustering

Hierarchical Flow Graph The input program is mapped to a hierarchical control flow graph like the bubble sort example in figure 3. In this figure, every control construct, such as if, case, loop, is represented by a shaded area. The nodes in these areas are the basic blocks of the program. Each of the control constructs has an associated condition that decides which of

the paths of the construct is executed. Conditions of the control structures can be nodes as well because an execution leading to a basic block and therefore a node may be necessary for the evaluation of the condition. In this graph, a Program Control Segment PCS is a Program Segment with an associated control structure in the control flow graph. Each control structure as well as the nodes are classified as being either SFP or MFP.

```
for(    i=0;
        i<14;
        i++ )
{
   for( j=i+1;
        j<15;
        j++ )
   {
        if(a[i]<a[j])
           swap(a[i],a[j]);

   }
}
```

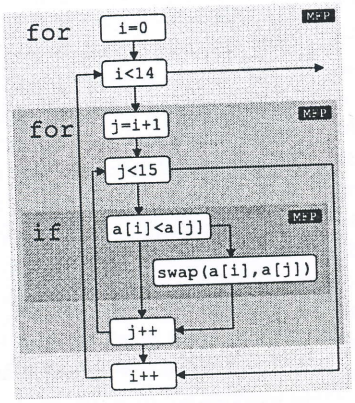

Figure 3: Flow graph with a control structure hierarchy

Feasible Paths in the Control Flow Graph Figure 3 shows a bubble sort algorithm and its control flow graph. A conservative analysis assumes that the program paths branch at the *for* and the *if* statements [4] such that all the corresponding program segments have the MFP property. In figure 4a two possible paths for every iteration of the loop can be seen, one of which is being taken for every iteration. If the condition in the *if* statement is evaluated, it can be recognized that values in $a[]$ are not known, meaning two potential paths for every loop iteration leading to $2^{loop\ iterations}$ potential paths. The first major step is to split the program in two segments, the *if* construct and the rest. The two paths through PS_2 are now considered to be merged into a single MFP segment. As a consequence, the paths of figure 4a are merged into a single path through the segments PS_1 and PS_3 in figure 4b which winds around the two fixed, and hence input data independent loops. In other terms, PS_1 and PS_3 become an SFP segment which includes the program segment PS_2 with separately analyzed cost bounds.

SFP Identification As we see in the example, the flow graph nodes need to be partitioned into SFP and MFP nodes. Every program control segment which does not contain an input data dependent control construct must be SFP. Nodes of the CFG, i.e. basic blocks, are SFP by definition. A control structure is SFP, if it only contains SFP nodes and its associated condition is independent of input data. The input data dependency of conditions can be determined by an algorithm. It requires a global data flow analysis [1] which forms a transitive closure over all data dependencies of variables in control statements. A global data flow analysis does typically not cover dependencies across array elements and pointer operations. Therefore, the global data flow analysis is complemented with symbolic simulation of basic blocks [11]. There is a simple recursive clustering approach to flow graph partitioning. It automatically cuts the program into SFP and MFP segments. This is shown in the following:

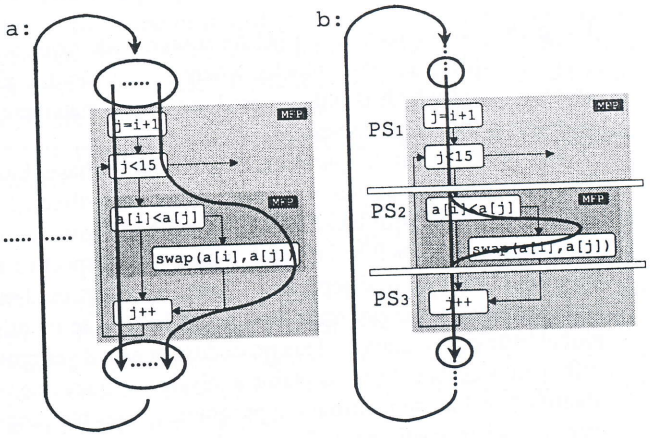

Figure 4: a: Program paths of bubble sort code shown above, b: Separation of the *if* construct

```
SFP Clustering Algorithm:

for all PCS on top level
    if PCS contains condition with symbolic expressions
        if condition is independent from data: PCS is SFP
        else: PCS is MFP, cut out PCS
    for all basic blocks nodes of the PCS
        basic block node follows PCS classification
        recursively check all sub PCS and sub BB nodes
```

SFP clustering is not sufficient when MFP path segments are embedded. The bubble sort in figure 3 is an example. The clustering algorithm is extended to merge adjacent SFP blocks: If the associated condition of a hierarchical node depends on input data, this PCS has the MFP property. Cut points are set at the beginning and the end of the MFP nodes and clustering is repeated ignoring the MFP nodes but regarding the cut points. In this case we remove the MFP from the graph, analyze it separately regarding SFP segments on lower levels and MFP cost, and add its cost to the SFP assuming worst case intervals at the cut points. For the SFP, MFP cost c is set to 0. The total cost C is composed by the cost for the SFP c_{SFP}, the cost for the transitions at the beginning and the end $c_{transition,wc}$ and the cost for the MFP c_{MFP}.

$$C = c_{SFP} + 2c_{transition,wc} + c_{MFP}$$

The new clusters found shall also be defined as SFP, since there is only one path after separation of the embedded MFP blocks. To be conservative for correct analysis, it is sufficient to analyze the MFP node separately and assume worst case analysis behavior at the remaining cut points. This is guaranteed by leaving the cut points inside the SFP clusters. Then, no

false SFP paths leading to incorrect bounds can be introduced in the next steps. The result is still correct, but SFP segment lengths are maximized. For most higher performance architectures with pipelines as well as for architectures with caches, analysis precision increases with path length. Figure 5 shows the result for the example in figure 3. Only the condition basic block, the comparison and the swap() are in MFP path segments.

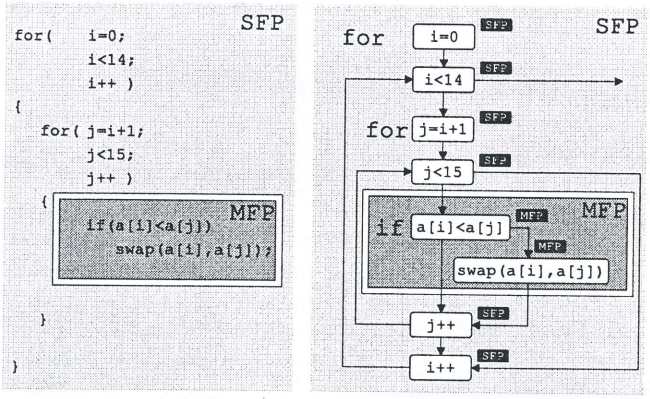

Figure 5: Single Feasible Path property with isolated Multiple Feasible Path among the distinct paths

3.5 Context Dependency and SPI

The path analysis approach presented above is based on the identification of input data independent control flow defining single feasible paths. This improves the accuracy compared to the approach by Malik in [4]. On the other hand, it does not exploit context dependent execution patterns as they are utilized in SPI for the concept of process modes [13]. Figure 6 gives an example. System simulation of an ATM switch identifies some of the cells in the data cell stream as so called operation and maintenance cells OAM which control the ATM connection [10]. They do not carry user data so they are irrelevant for data transmission. Figure 6 shows a typical code segment to handle the operation and maintenance component of the switch. The control flow graph is shown in figure 7. When processing OAM cells, the shaded *else* program segment in figure 6 cannot be reached. It should not be included in further analysis. The context "VCI = 3" selects a subset of all program paths rather than a single path. It corresponds to a mode at the level of a SPI process network. More precisely, every mode can be annotated by a corresponding context. This is, then, used for a context dependent analysis of the SPI process which provides one cost interval per context. The cost interval can be back annotated to the SPI process as behavioral interval of the respective mode for a given target architecture. This way, SPI process network analysis and individual process analysis have been tightly integrated. The next sections shown how the context dependent analysis is implemented.

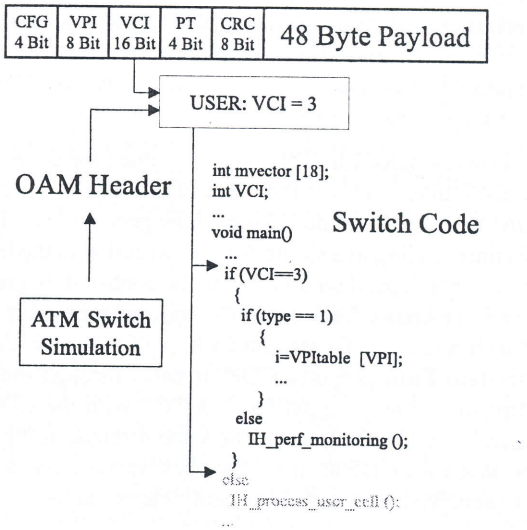

Figure 6: Path selecting property of the known VCI of a cell

3.6 Classification of Input Data

Process analysis needs to identify the context dependency in the source code. It is given by input data for a mode or context in the control structures. Therefore, all input data in control structures leading to MFP segments in the previous approach has to be classified. Constants in control structures leading to SFP segments are not regarded to be input data. Input data is given to the process at execution time and belongs to one of the following categories.

Constant Data

```
for(i = 0;i < const_fourteen;i++)
    for ...
```

Constant input data may have a predefined value that is known by the user like the value of 14 for the loop bound. This may be the case for unused header fields that are put to a defined default value. This case is obvious and not expected to have great impact because few control structures will depend on constant data that are not found by SFP analysis. Control flow gets predictable for this kind of control structures according to the SFP definition like in figure 5.

Unknown Data

```
for(i = 0;i < const_fourteen;i++)
  for(j = i;j < const_fifteen;j++)
    if(a[i] < a[j]) { ... } /* a is unknown */
```

While the loop bounds belong to the previous class of data, the contents of the array $a[]$ are not known. This class of input data is variable, meaning that it is not known in any case. This is the default case used in the state of the art [4] leading to MFP that are isolated.

Context Dependent Input Data

```
if{VCI == 3} /* Designer: VCI = 3, known from context */
  if{type == 1}
    {...}
else
    {...}
```

Process modes in SPI can give values for some input data in the context. This is the value of the VCI in figure 6 for the OAM execution mode. Control flow gets predictable at analysis time leading to a single flow of execution in the hierarchical nodes that depend on this data in the control flow graph. In figure 6, the known VCI selects the upper part of the *if* statement. This is a Context Dependent Path. A PCS has the **Context Dependent Path** property **CDP**, if paths through the PCS only depend on known input data. A PCS with the CDP property is called a CDP–segment. The same discussion for the gain in accuracy for CDP as in [11] applies because longer sequences are achieved than with SFP identification alone.

3.7 Extended Clustering

CDP segments are only found within MFP segments. Without a modification to the clustering approach they stay isolated improving only estimation accuracy of the MFP parts. At the transitions between SFP and CDP segments, no cut points are needed and the clusters can be extended. CDP segments and SFP segments are merged for a given context by extending the clustering algorithm. The longer clusters result in higher analysis precision.

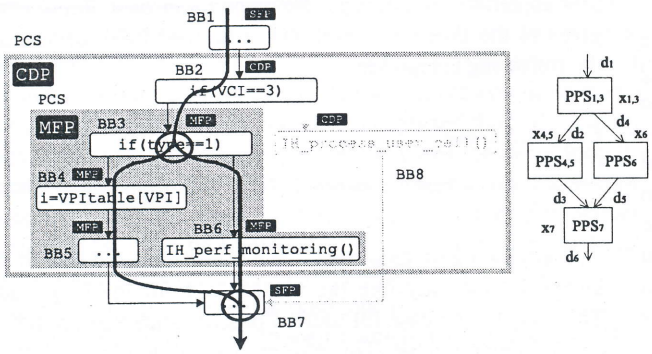

Figure 7: SFP, CDP and BB collapse to program path segments

After clustering, the graph consists of SFP/CDP clusters with a single flow of control and isolated MFP. The SFP/CDP clusters can be treated like basic blocks in the following giving the possibility to reduce the control flow graph to a graph of program path segments PPS. This is shown in figure 7, where the control flow graph for our example is collapsed. Program path segments like $PPS_{1,3}$ from CDP segments and $PPS_{3,5}$ from SFP segments on lower levels as well as program path segments PPS_6 from single basic blocks are reduced to single nodes in the resulting control flow graph that needs to be analyzed using the methodology in [4] for MFP segments.

Before we get to the analysis of MFP segments, the cost of the program segments is needed. The path analysis approach that has been described up to this point is shown in the upper part of figure 8. Local analysis of the encapsulated program segments is shown on the left side. These results are known in the global path analysis on the right side of figure 8 that we explain in the following. Their determination is described in section 4.

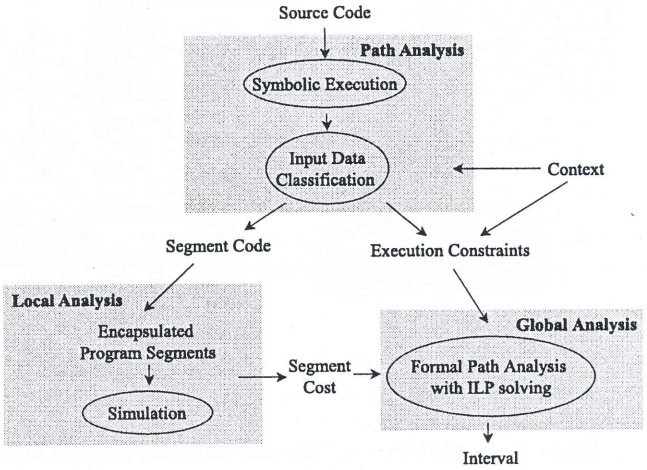

Figure 8: The toolflow with analysis steps and interfaces

3.8 Global Analysis

For the global analysis, the cost for the program path segments, as e.g. in figure 7 is assumed to be known. For the MFP program segments the methodology proposed in [4] and used by the first clustering approach in [11] is adopted. The execution cost C is assumed to be the sum of all basic block or program path segment execution costs c_i multiplied with their execution counts x_i when basic block and path cost is the same for every execution.

$$C = \sum_{i=0}^{N} c_i \times x_i$$

In figure 7, this means that the overall execution time T as an example for the behavioral interval of the MFP is

$$T = x_{1,3} \times t_{1,3} + x_{4,6} \times t_{4,6} + x_6 \times t_6 + x_7 \times t_7$$

with t being the execution times and x the execution counts of the nodes. Intervals for the MFP cost I_c are needed as the execution count of the program path segments in an MFP can be an interval with a minimum execution count $x_{i,min}$ and a maximum execution count $x_{i,max}$. For the program path segment execution count interval $[x_{i,min}, x_{i,max}]$, the user provides an implicit description of the path by means of linear equations for execution counts. For example, this could be $x_{4,5} > 2x_6$ meaning that `i=VPItable[VPI]`,... is executed more than twice

as often as `IH_perf_monitoring()` in figure 7. Annotations may be given for the MFP part. This kind of user annotation requires a deeper understanding of the software under investigation than the annotation of known input data available from process modes. It requires the execution constraints shown in figure 8. The execution count inflow d of a program path segment PPS in figure 7 equals its execution count x and its execution count outflow d. It defines another set of equations [4].

$$\sum_{PPS} d_{inflow} = x_{i,PPS} = \sum_{PPS} d_{outflow}$$

In figure 7 the equations are $d_1 = x_{1,3}$; $x_{1,3} = d_2 + d_4$.. These equations and inequations for the upper and the lower execution count bound are mapped to two ILP problems which can be solved to derive the widest execution count interval $[x_{i,min}, x_{i,max}]$ for each MFP. The execution cost c_i of the basic block is determined by architecture modeling assuming a constant execution cost c_i in the first approach. So the cost interval I_c for an MFP segment can be calculated. When c_i is not constant for every execution because of data dependent instruction execution or pipeline hazards and register allocation due to unpredictable execution sequences in MFP parts, an interval $[c_{i,min}, c_{i,max}]$ for the cost is needed, redefining the interval to

$$I_c = \sum_{i}^{N} [c_{i,min}, c_{i,max}][x_{i,min}, x_{i,max}]$$

Just assuming the worst case for c_i which is common practice invalidates the best case for the complete interval delivered by the ILP solver because of the overhead that is always added.

4 Architecture Modeling

Now, the local analysis from figure 8 for the program path segments is explained. The cost is determined by simulation using one of the following two techniques when appropriate.

Instruction Cost Addition ICA The instruction or statement execution costs in a basic block or along a path segment are added. These execution costs are taken from a table. This is a very computation time efficient approach. Minimum and maximum instruction execution cost can be considered.

Program Segment Simulation PSS The basic block or program segment is simulated using a cycle true processor model which can exactly model hardware cost.

4.1 Instruction Execution Interval

Instruction execution cost can be dependent on input data. A popular example is a shift-and-add implementation of a multiplication in a processor delivering an interval for c_i. Before the program path segment functions can be executed, the used variables must be initialized to legal values to avoid for example divide-by-zero effects. A standard data flow analysis [1] provides the variables to be declared and initialized including the user annotations for CDP parts. Where values at PPS starts are not available, because the segment is not reached in simulation or delivered by data flow analysis, ICA must be used. The worst case cost from the table is needed in this case. As an alternative, a set of data for the paths and basic blocks can be assumed where not available and PSS is used. After that, the cost for the data dependent instructions are compared with the worst cases in the instruction tables. The difference between the values for the assumed data and the worst case is added afterwards.

4.2 Simulation for Timing and Power

As an example for PSS that delivers timing and power consumption of the program path segments, a StrongARM simulator core is combined with the DINERO III cache simulator delivering both instruction and data cache behavior. Both source codes have been recompiled to one simulator to achieve better performance. Architecture modeling regarding timing is derived from [2] while the energy dissipation model is taken from [7]. Data rates are directly derived from the amount of data produced or consumed on a path and its execution count interval. Simulation results for timing and power consumption of program path segments are used in the global analysis to calculate the global interval. The results for a PPS regarding timing and power are already intervals because data dependent instruction execution for divisions can be present and cache simulation can start from both first hit as the best case and first miss as the worst case if applied.

For data rates, the communicated amount of data is directly given by the number of executions of a send or receive statement and the size of the data block it is communicating.

5 Experiment

The behavioral interval determination has been applied to a single process that reads a packet and loads a picture. If the picture is addressed to the system component, it performs an "unlikely dot" filter on the picture data and sends it to another buffer. Loop bounds for the case that no mode is selected have been annotated. The StrongARM frequency is set to 80 MHz, the bus frequency to 40 MHz and the memory access time to 25 ns in the PSS. A pseudo code description is given as follows.

```
Experiment:

header = receive(INPUT, HEADER_SIZE);          /* line 89 */
for all pixels
    picture[y][x] = receive(INPUT, 1);

if(address == MY_ADDRESS) {  /* Ann. address, line 122 */
    for all pixels {                           /* line 124 */
        for a 3*3 pixel window {
            if(without_center)/* Ann. center, line 143 */
                average = sum/8;
            else average = sum/9;
                                               /* line 151 */
            if(abs(picture[y][x]-average)>threshold)
                send(OUTPUT, average, 1);
            else send(OUTPUT, picture[y][x], 1);
        }
    }
}
```

Table 1: Behavioral Intervals without mode or annotation

Line+	Type	Latency ms	Power mWs	Sent bytes	Received
89	SFP	[4.92,38.0]	[2.0,8.5]	[0,0]	[6197,25045]
122	MFP	[413ns,2475ns]	[50nWs,178nWs]	[0,0]	[0,0]
124	CDP	[39.5,329]	[17.5,72.6]	[0,0]	[0,0]
143	MFP	[1.54,131]	[0.65,14.7]	[0,0]	[0,0]
151	MFP	[16.7,182]	[2.85,20.4]	[0,24393]	[0,0]
Best	–	4.955	2.099	0	6197
Worst	–	680.847	116.211	24393	25045

In table 1, behavioral intervals without mode or address annotation are given. Due to the loop bounds given above, we know the minimum and maximum number of pixels leading to a CDP in line 124. The intervals for latency time, power consumption and data rates as well as the path classification are given for every program segment that is referenced by the line number it is starting with. They can be very wide because worst cases assume a cache flush for the beginning of the segment while best cases assume hits. SFP segments may be parts of CDP segments, so they may not be visible in the results. Sent numbers of data bytes do not equal the received numbers of data bytes because headers have to be received and the outer pixels are not sent to the output buffer.

Table 2: Behavioral Intervals: Large mode, MY_ADDRESS

Line+	Type	Latency ms	Power mWs	Sent bytes	Received
89	SFP	[19.2,38.0]	[8.4,8.5]	[0,0]	[25045,25045]
122	CDP	[164,329]	[72.6,72.6]	[0,0]	[0,0]
143	CDP	[15.7,22.6]	[4.4,5.00]	[0,0]	[0,0]
151	MFP	[64.9,182]	[11.8,20.3]	[24393,24393]	[0,0]
Best	–	264.604	97.308	24393	25045
Worst	–	572.012	106.514	24393	25045

In table 2, the address has been annotated to a match which leads to a CDP instead of an MFP in line 122 which is clustered with the CDP in line 124. The calculation of the luminance including the center pixel has been annotated as well affecting line 143 which is not clustered with other segments. The process mode has been annotated as LARGE, meaning that the big version of the picture is computed. This leads to tighter intervals because the execution path through the filter is known as well as the loop bounds for the picture. The only MFP is caused by the control structure depending on picture data. The communicated data rate is exactly known due to the known picture size for the mode.

Worst cases for timing and power consumption get tighter for a worst case annotation because the known sequences for the context lead to higher analysis precision.

Table 3: Behavioral Intervals for different annotation scenarios

Annotation	Latency ms	Power mWs	Sent bytes	Received
Mode Small	[4.955,66.71]	[2.099,24.61]	[0,5865]	[6197,6197]
Mode Large	[19.24,680.8]	[8.474,116.2]	[0,24393]	[25045,25045]
No Mode	[4.771,680.8]	[2.099,116.2]	[0,24393]	[6197,25045]
Small+Address	[38.49,63.62]	[21.03,23.61]	[5865,5865]	[6197,6197]
Large+Address	[264.6,572.0]	[97.3,106.5]	[24393,24393]	[25045,25045]
No Mode+Address	[38.49,572.0]	[21.03,106.5]	[5865,24393]	[6197,25045]

In table 3, different scenarios for process modes and data annotation have been explored. In the first three lines, just modes or no modes have been annotated, while the address and the luminance calculation have been annotated in the last three lines.

Using the results from the annotations of picture size and address, the mode set of the process in SPI is as follows:

$$M = \{m_{Small}, m_{Large}\}$$

$$m_{Small} = < [38.49, 63.62]ms, 6197, 5865 >$$

$$m_{Large} = < [264.6, 572.0]ms, 25045, 24393 >$$

Each mode is a tuple of latency, input data rate and output data rate. In comparison with the description with a single behavior (last line in table 3), the behavioral intervals have been substantially narrowed.

6 Conclusion

Process timing and power consumption can be highly context dependent. Process modes are introduced to distinguish contexts with significantly different timing and power consumption. An existing symbolic analysis approach is extended to capture context dependent behavioral intervals of single processes which are then used to model concurrent process networks in the SPI representation. A wireless IP base station is given as a motivational example. The results demonstrate a significant improvement in process modeling accuracy.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compiler Principles, Techniques and Tools*. Bell Labs, 1987.

[2] S. Furber. *ARM System Architecture*. Addison Wesley, 1996.

[3] J. Gong, D. Gajski, and S. Narayan. Software execution from executable specification. *The Journal of Computer and Software Engineering*, 1994.

[4] Y. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer, 1999.

[5] J. Liu, G. Maguire, M. Mateescu, A. Schmidt, and R. Ruppelt. Document of network architecture strategies and tradeoffs. ESPRIT MEDIA report, KTH Stockholm, 1999.

[6] A. Mok. Evaluating tight execution time bounds of programs by annotations. In *Proceedings of the Workshop on Real Time Operating Systems and Software*, 1989.

[7] J. Montanaro. A 160-MHz,32-b,0.5W CMOS RISC microprocessor. *IEEE Journal of Solid State Circuits*, 1996.

[8] C.Y. Park and A.C. Shaw. Experiments with a program timing tool based on source level timing shema. In *Proceedings of the Real-Time Systems Symposium (RTSS '90)*, 1990.

[9] P. Puschner and C. Koza. Calculating the maximum execution time of real time programs. *The Journal of Real Time Systems*, 1989.

[10] F. Wolf and R. Ernst. Software timing and power estimation of telecom systems. ESPRIT MEDIA report, Technical University of Braunschweig, 1999.

[11] W. Ye and R. Ernst. Embedded program timing analysis based on clustering and architecture classification. In *International Conference on Computer-Aided Design (ICCAD '97)*, 1997.

[12] D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L. Thiele. Combining multiple models of computation for scheduling and allocation. In *Proceedings of Codes/CASHE*, 1998.

[13] D. Ziegenbein, K. Richter, R. Ernst, J. Teich, and L. Thiele. Representation of process mode correlation for scheduling. In *International Conference on Computer-Aided Design (ICCAD '98)*, 1998.