A High-level Target-precise Model for Designing Reconfigurable HW Tasks

Maik Boden[1], Steffen Rülke[1], Jürgen Becker[2]

[1]Fraunhofer IIS / EAS Dresden
Zeunerstr. 38, 01069 Dresden, Germany
{boden|ruelke}@eas.iis.fraunhofer.de

[2]University of Karlsruhe / ITIV
Engesserstr. 5, 76131 Karlsruhe, Germany
becker@itiv.uni-karlsruhe.de

Abstract

The increasing complexity of embedded digital HW/SW systems, rising chip development and fabrication costs, and a shortened time-to-market require system-level design methods and the use of reconfigurable architectures. Our design method concerns the modelling of a system and its HW tasks at a high abstraction level. Using design patterns and macros, our library-based approach provides a consistent flow from an executable specification to its implementation. These templates ease the efficient application of partially run-time reconfigurable architectures. A case study depicts the high-level modelling of a HW task and its implementation in detail.

1. Introduction and motivation

Run-Time Reconfiguration (RTR) of HW enables the realization of System-on-Chip (SoC) solutions that provide higher performance and lower power consumption as well. Such requirements rise from innovative mobile applications for ubiquitous computing [5]. In recent years many RTR architectures have been proposed [6]. Especially coarse-grained architectures, capable of being reconfigured at run-time partially, provide a suitable platform for self-organizing, self-healing, and self-protecting systems [3].

A shortened time-to-market and the continuously rising complexity of embedded systems requires adequate design methods. Therefore, a high abstraction level and an executable system specification [8] have to be applied. Furthermore, novel architectures demand a new design paradigm and related SW-tools to enable Electronic System-Level Design (ESD) [7]. For example, High-Level Synthesis (HLS) transforms an algorithmic description into a target-specific mapping, immediately.

Regarding an efficient application of RTR architectures, optimization techniques have to reduce the overhead for reconfiguring a chip at run-time. But available approaches like the common sub-graph extraction [1] address the data path only. Online scheduling of block-partitioned devices, such as the Erlangen Slot Machine (ESM) [4], require cost functions to determine the correlation of HW tasks [10].

To close the gap, we propose a model-based design approach considering the Control and Data Flow Graph (CDFG) of a HW task. Our model specifies the behavior of a task using design patterns and macros provided by a design library. The operation sequence is represented by a binary graph which can be interpreted as a signature (pattern). Thus, pattern matching becomes a smart technique to determine the correlation of two tasks, and to calculate the system's reconfiguration costs.

This paper is structured as follows: Section 2 introduces our RTR design method including a system-level model. The design library and our RTR task model are the subject of Section 3. In Section 4 we apply our method to a case study and explain the implementation using an RTR target architecture. Section 5 summarizes the paper and examines some conclusions.

2. Design method and models

This section introduces our method for designing an adaptive embedded system. The starting point is a system-level model representing an executable specification. After HW/SW partitioning, we translate the dedicated HW tasks into our RTR task model. It applies macros and patterns of design libraries to ensure precise target mapping and design optimizations for an efficient use of RTR.

2.1. Design flow and system model

The system-level model of an embedded digital system is the starting point of our design flow (see Figure 1). It represents an executable system specification for emulating the behavior of the entire system.

The system S is specified by a set T of tasks, a set R of communication resources, and the task schedule t_s. Each task $t \in T$ implements a dedicated part of the entire system. Tasks interact via communication ports $p \in P$ connected to a system resource $r \in R$. All system components (such as task design elements, storage resources, or communication protocols) are provided by a system library.

1-4244-0054-6/06/$20.00

During HW/SW partitioning, each system task $t \in T$ is marked for implementation as a SW task or as a HW task. Regarding to our RTR design method, this paper does not consider the realization of SW tasks as well as aspects of HW/SW co-design. We apply the system-level model as test bench and for test pattern generation. In this case, resources bordering HW tasks act as input or output for the RTR design part.

The target library provides a set of design elements for implementing a HW task on the RTR architecture. It is a refinement of the system library, which considers specific constraints and attributes of the target architecture. For an efficient use of the new technology, several RTR design optimizations are necessary. Details of modelling and implementing a task are the subject of the next section.

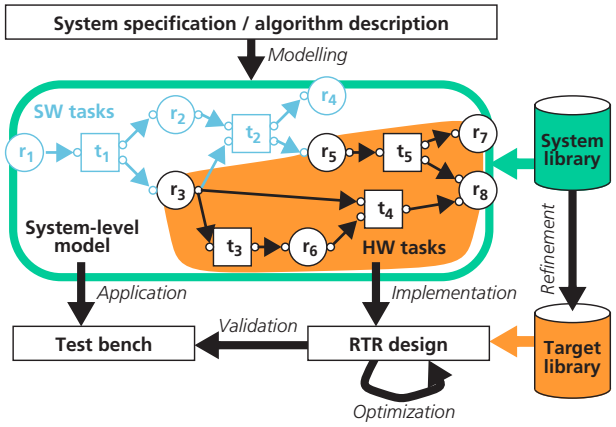

Figure 1. Design flow and system model

A task t_i represents the smallest unit for HW/SW as well as RTR partitioning. To enhance the granularity, a task t_i needs to be split up into sub-tasks $t_{i1}..t_{in}$, if possible. This requires a remodelling of the executable specification, where $t_{i1}..t_{in}$ replace t_i in the system. Additional resources for the communication of the sub-tasks become part of the system, too. Especially the partitioning of HW tasks for RTR requires a finer-grained system model.

2.2. Modelling and implementing of HW tasks

Commonly, a High-Level Language (HLL), such as C++ or SystemC, is used to develop the system-level model. Its high abstraction level for algorithm specification eases modelling of the entire system. But in respect to a HW implementation and to RTR design optimizations, a Register-Transfer Level (RTL) model is required for target mapping.

The target library provides design elements that meet the RTR design requirements (high abstraction level and RTL realization) to model and implement a HW task t_k:

- A set M of design macros implements logic, arithmetic, and communication functions. Each operation $o \in O_k$ of task t_k is specified by a design macro $m \in M$.
- A set Y defines all operand types. The operands, called variables, are stored in the set V_k. Each variable $v \in V_k$ of task t_k is defined by an operand type $y \in Y$.
- A set I of interfaces specifies communication ports. Task t_k communicates using a set P_k of ports. Each port $p \in P_k$ is specified by an interface $i \in I$ and bound to a system resource $r \in R$.
- A set D of design patterns provides templates for the control flow. The task signature s_k represents the flow of all operations $o \in O_k$ of task t_k using the pattern. Each design pattern $d \in D$ concatenates two operations, two patterns, or one operation and one pattern.

Figure 2 shows the application of our system-level design method using an architecture-specific target library for precise modelling of a HW task t_k.

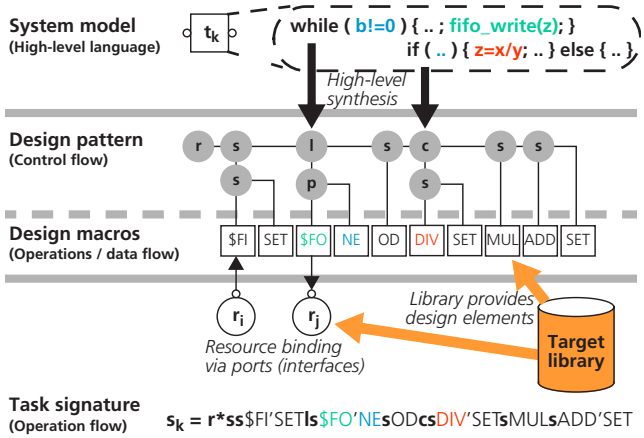

Figure 2. Task modelling and target mapping

After generating the task signature via HLS or by hand, an architecture template is used for target mapping and to realize the HW implementation.

Figure 3 depicts the implementation of task t_k starting from the task signature and the operation flow:

- The data path (respectively the operation sequence) is partitioned according to the task signature. In this step, the task signature s_k is pruned by removing redundant design patterns. The so called control signature s_k' is used for the controller and data path generation.
- The data path comprises of the operations O_k, the ports P_k, and the variables V_k. The implementation of each operation, port, and variable follows the template of the

related macro, interface, and operand type specified in the target library.

- The control signature s_k' contains all design patterns necessary to generate the control path. The template of each pattern represents a piece of the entire controller. Composing all templates generates the Finite State Machine (FSM).

To validate the signature of task t_k using the test bench of the system-level model, an identical SW model can be generated easily: The operation sequence is substituted by the corresponding library elements of the system library.

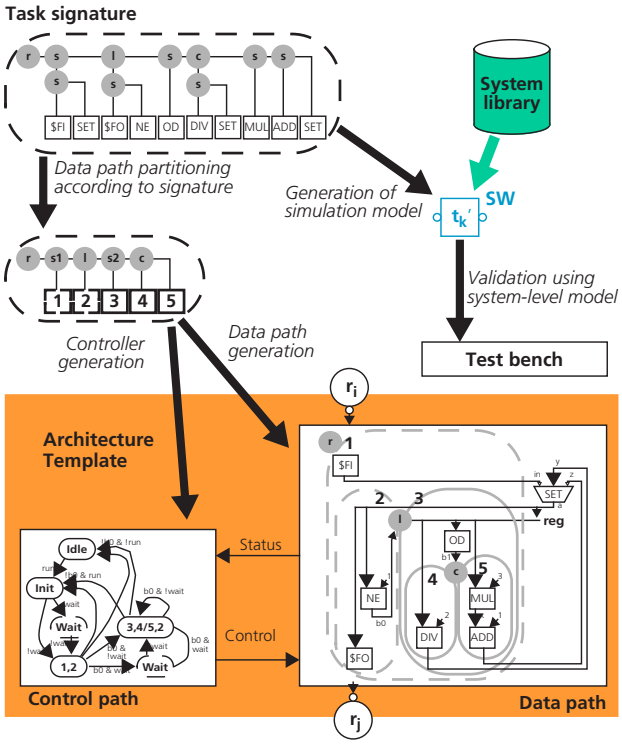

Figure 3. HW task implementation

Section 4 applies our design method for modelling and implementing HW tasks on an example and describes the data path and control path generation in detail.

2.3. Design optimization to reduce RTR overhead

The RTR implementation of HW tasks creates both benefits and drawbacks:

On the one hand RTR increases the flexibility of chip applications: Two or more designs share the same chip area. An optimized implementation of a design (for example achieving lower power consumption) can be loaded depending on the system state at run-time.

On the other hand, RTR produces some overheads: Additional memory is necessary to store the configuration data. Furthermore, during RTR, the reconfigured part stays idle until the new configuration is completely available. The consequential delay decreases the performance, if the application needs to pause until RTR has finished.

To use RTR efficiently, the RTR overhead must be reduced. Our model provides cost functions that enable the development of optimization strategies and algorithms.

A first set of cost functions determines the operation time, the operation performance, and the required chip area of a task. It is implemented by one or more configurations. A second set of cost functions determines the size, the corresponding chip area, and the reconfiguration time of the set of HW configurations. Thus, both performance and implementation costs of a sub-system can be calculated.

In general, an optimization algorithm that compares all possible variants of implementations and configuration sets is not applicable. For this reason, the strategy must start on a higher abstraction level.

Using our RTR design model, the similarity of two HW tasks is represented by the similarity of the corresponding task signatures. Because a signature is a string, pattern matching techniques can be applied for calculating the similarity. The common macro set $f_{cm}(t_1,t_2)$ of two HW tasks (t_1 and t_2) represents function units that are used by both tasks. Furthermore, the set of common signatures $f_{cs}(t_1,t_2)$ represents operations that are related to each other. This means, that the function units of the implementation have local interconnections. Thus, a common signature is a good candidate for partitioning the configuration.

Figure 4 shows the signature and the data flow of two tasks. The common signatures represent reusable modules that become part of a configuration used by both tasks.

After determining the similarity of two tasks, methods like [1] can be applied for correlating the data paths.

3. Design patterns and design macros

In Section 3, we explain the task execution model and its representation - the task signature. Following this, we clarify the different abstraction levels of design patterns in the design libraries. Finally, constraints for target mapping and classifications are discussed.

3.1. Execution model and its representation

As explained in Section 2, our design method is based on design libraries. They provide design patterns D, and design macros M to specify the function of task t_k. Operations O_k represent the instantiated design macros, and the task signature s_k represents the operation flow using design patterns.

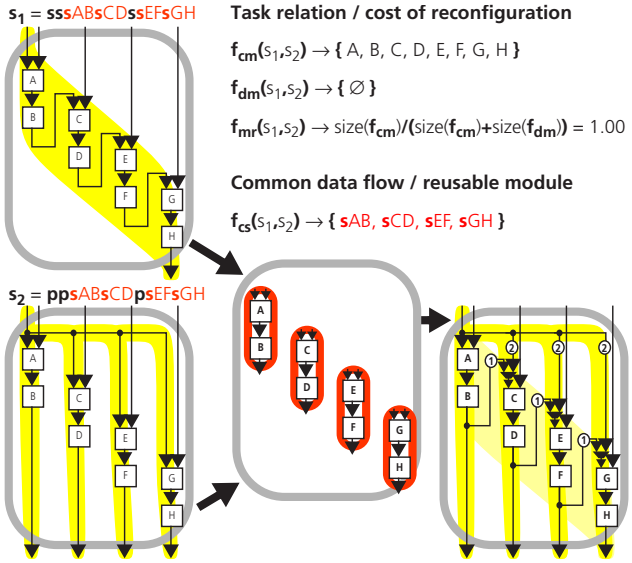

Figure 4. RTR design optimization

The execution model behind the task signature is the Binary Macro Tree (BMT). The model is based on a tree consisting of branches and leaves. The leaves represent the operations O_k of the corresponding task t_k. The set B of branches represents the applied design pattern. Each $b \in B$ specifies the relation between two sub nodes (leaves or branches as well) of this branch according to the definition of the corresponding design pattern d ∈ D.

Figure 5 depicts that a HW task t_k is - formally correct - specified by a 4-tuple (B_k, O_k, V_k, P_k).

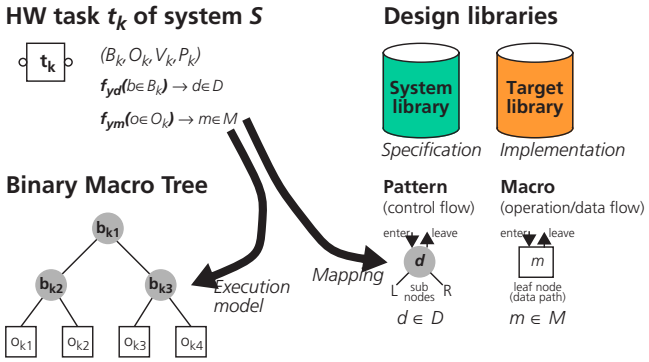

Figure 5. Binary Macro Tree (BMT)

The task signature is a preorder (or prefix) traversal of the BMT as shown in Figure 6. Thus, a string (pattern) is sufficient to define the operation flow.

3.2. Design patterns and their abstraction

Design patterns describe how to interprete a BMT. Only macros represented by operations (the BMT leaves) can be executed. The order of processing operations (the operation

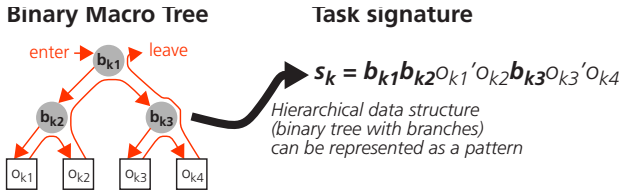

Figure 6. Task signature

flow) is specified by design pattern (the branches).

Regarding to our design method, design patterns are available in different abstraction levels. The system library provides a HLL specification that is used for an abstract specification of the algorithm realized by a task, whereas the target library provides templates - the counterparts for implementation (see Figure 7).

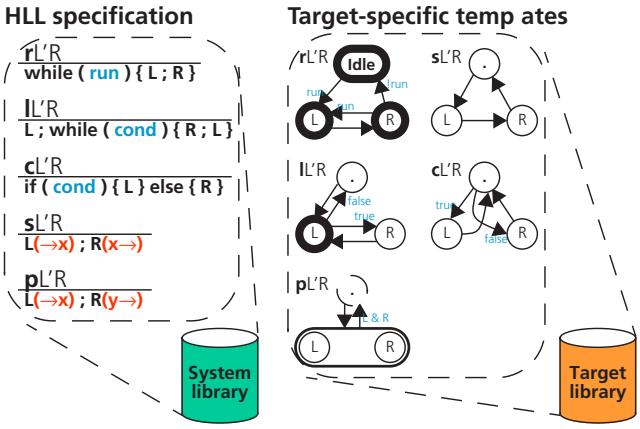

Figure 7. Abstraction of design pattern

Different abstraction levels of the same design pattern enable the generation of a target-precise task model as well as a system-level task with an identical behavior model using the same task signature. For design validation, the original executable task specification in the system model is replaced by the generated one. Thus, design consistency can be proved by validating the modified system model using the original test bench.

3.3. Target mapping: node types and macro classes

The target library provides templates of design patterns and design macros for a target architecture. But different architectures support different levels of abstraction, and require different schemes for synchronizing both the data and the control path. A programmable logic device (e.g., a Xilinx Virtex FPGA [11]) implements designs on a low level of abstraction using logic function generators, clock-triggered flip-flops, and a routing matrix as oposite to a coarse-grained computing architecture (e.g., PACT XPP [2]) provides self-synchronizing arithmetic/logic function

units, an adaptive inter-communication network, and an intelligent RTR controller hierarchy. Many other RTR architectures have been developed in recent years [6].

To enable the reuse of HW tasks using different architectures and to unify the interpretation of a task's signature, we defined node types and macro classes.

A node type specifies the behavior of the controller's FSM regarding an operation execution (see Figure 8):

- A regular node represents an operation. Thus, each state in a template executes the related operation. After its execution, the next state of the FSM is entered.
- A post-triggered node requires a trigger to leave, which means that the controller's FSM stays in current state until the trigger is fired. Afterwards, the next operation is executed.
- A pre-triggered node requires a trigger to enter, which means that the controller enters a wait state right before this FSM state. When the trigger is fired, the controller enters this state and executes the operation.

A node can be regular, post-triggered, pre-triggered, or post- and pre-triggered.

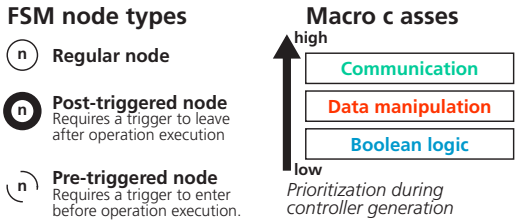

Figure 8. Node types and macro classes

The operation sequence is represented by a node graph. Macro classes characterize the node type regarding to the operation executed by this node. We distinguish between three macro classes:

- Communication macros handle interfaces, which means the corresponding operation receives local data from or sends local data to a port. Depending on the kind of interface, communication macros can cause wait cycles when required data for processing is not available or computed data can not be transmitted.
- Data manipulation macros apply arithmetic operations or complex functions on local data. These macros are characterized by a fixed processing time and require a regular node.
- Boolean logic macros represent the interface between control and data path. They provide status information of data manipulations or communication functions. Usually, the macros require a regular node.

4. Case study - an application of our method

This section describes the application of our method using an example: the Collatz conjecture. Starting from an algorithm description, we model the task using the HLL source, create the signature, and implement the task on our architecture template.

4.1. Example: Collatz conjecture

The Collatz conjecture is one of the long-standing open problems of mathematics. Apparently first posed by Lothar Collatz in the 1930s, it has since withstood every attempt at proof. The problem is related to a wide range of topics in mathematics, including number theory, computability theory, and the analysis of dynamic systems [9].

The statement of the Collatz conjecture involves the mapping M: $\mathrm{N} \rightarrow \mathrm{N}$ where:

$$M(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

Let a_i denote the result of i iterations of M on n:

$$a_i = \begin{cases} n & \text{for } i=0 \\ M(a_{i-1}) & \text{for } i>0 \end{cases}$$

The Collatz conjecture states that for all n there is some i such that $a_i=1$:

$$\forall n \in \mathrm{N} > 0: \exists i \in \mathrm{N}: a_i = 1$$

Figure 9 shows an example, where we start with $n=11$:

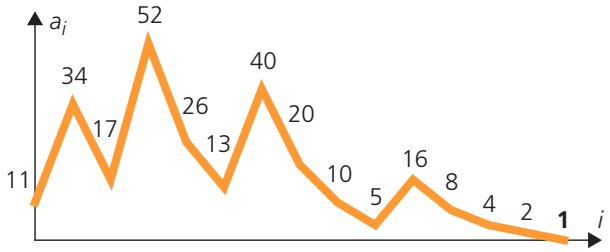

Figure 9. Example: Collatz conjecture

4.2. Deducing a task signature from a HLL source

For modelling a HW implementation of the related task function, we use the HLL source (ANSI-C) of a small command line program that computes a sequence for the Collatz conjecture (see Figure 10). After typing in an integer n, the program prints out all iterations a_i starting with a_0 until $a_i=1$.

```
main( void )
{
   readf( "%d\n", a );
   printf( "%d\n", a );
   while ( a != 1 )
   {
      if ( (a % 2) == 0 )
      {
         a = a / 2;
      } else {
         a = (3 * a) + 1;
      }
      printf( "%d\n", a );
   }
}
```

Figure 10. HLL source (ANSI-C)

At this point, the designer has to model the behavior of the given algorithm using the design pattern and design macros provided by the design libraries. We developed a Macro Sequence Language (MSL) to represent the BMT in textual form. Macros are instantiated like function calls in a HLL (such as C/C++). Consequentially, each function call represents an operation. The operations are executed in sequence according to the *sL'R* pattern. To apply a different design pattern (branch in a BMT), a pair of braces prefixed by the pattern's name is used. The first brace represents the left BMT tree, the second brace the right BMT tree (see Figure 11).

```
r( run ) // Program body (main)
{
}{
   $FI( in, f_in ); // Input port
   SET( a, in );
   l( b0 ) // WHILE statement
   {
      p()
      {
         $FO( f_out, a ); // Output port
      }{
         NE( b0, a, 1 ); // a!=1
      }
   }{
      OD( b1, a ); // (a%2)==0
      c( b1 ) // IF statement
      {
         DIV( x, a, 2 ); // a=a/2
         SET( a, x );
      }{
         MUL( y, a, 3 ); // a=3*a+1
         ADD( z, y, 1 );
         SET( a, z );
      }
   }
}
```

Figure 11. Macro sequence / MSL source

While modelling, the designer has to pay attention to several restrictions being conform to our model:

- To input or output data requires the application of dedicated communication macros (such as *$FI* to read from a FIFO or *$FO* to write to a FIFO) on defined I/O ports (such as *f_in or f_out*).
- Variables modified by different operations have to be unified using the *SET* macro (such as in case of *a*).
- Design patterns use only boolean operands. Thus, boolean logic macros have to be applied for performing data analysis operations, such as *NE* (not equal to).

Using the MSL source, the task signature as well as the BMT can be easily generated (see Figure 12):

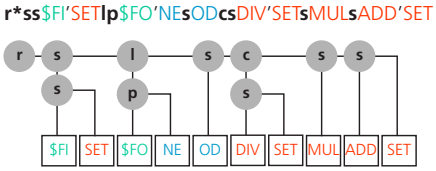

Figure 12. Task signature and BMT

Beside design optimizations regarding RTR, the task signature is used to generate the controller and to partition the data path accordingly. To wire the data path, the MSL source (comprising all variable mappings) is required.

4.3. RTL implementation of HW task

The implementation of a HW task consists of three steps: deducing the control signature, partitioning and wiring the data path, and generating the control path and optimizing the controller.

The control signature s_k' of task t_k is deduced from the task signature s_k by removing redundant patterns. These are all pattern that do not affect the control flow (such as *sL'R* or *pL'R*) excepting those which comprise non-redundant pattern. Figure 13 depicts the task signature and the deduced control signature of the example.

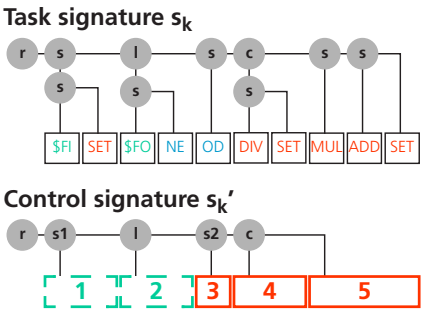

Figure 13. Deduction of the control signature

The control signature is the basis for partitioning the data path as well as generating the control path. The leaves of s_k' represent the control states of task t_k.

Partitioning the data path assigns each operation to a dedicated control state. Two operations belong to the same partition, if they are in the same BMT sub-tree according to the control state. All operations assigned to the same control state operate concurrently. To execute operations, the control path provides an execution signal (e_1,e_2,..,e_5) for each control state (*1,2,..,5*). The signal e_i is used to enable an operation as well as to switch the data flow if *SET* is applied to modify a variable. After partitioning, the operations are wired using the operation-to-variable mapping of the MSL source (see Figure 14).

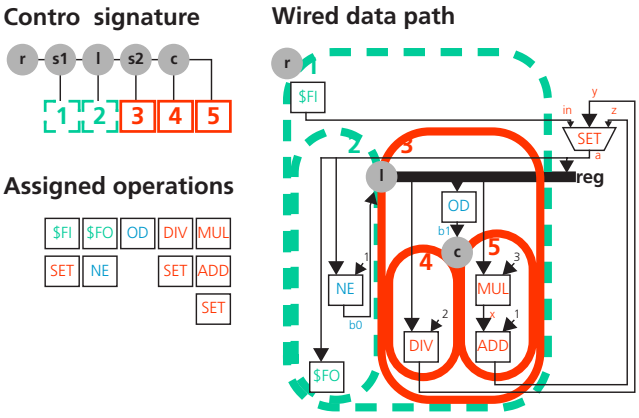

Figure 14. Data path partitioning and wiring

The generation of the control path starts with the application of FSM templates on the control signature. After that, each applied template represents a part of the entire control flow (see Figure 15).

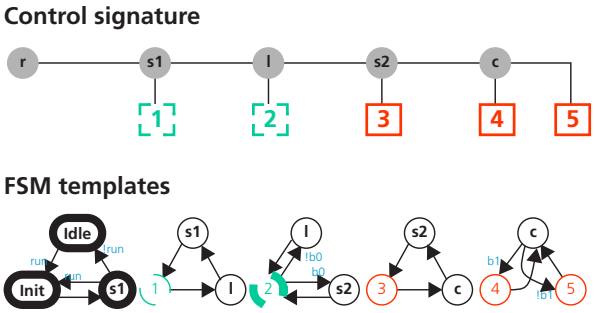

Figure 15. Application of FSM templates

The control path is produced by joining all templates. To join two templates, a leaf node representing a pattern is substituted by the related template (where the pattern is on the top of the template). After substituting, the controller's FSM consists of control states and system states (such as *Idle* or *Init*), see Figure 16 (left) only.

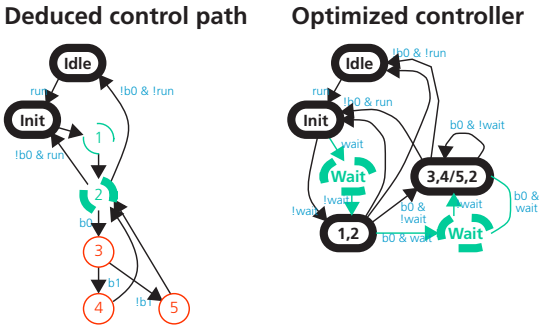

Figure 16. Control path and controller's FSM

Figure 16 (right) shows the optimized controller for our example. To optimize the controller, the control states are joined in a similar manner as applied to the templates. In this case, all control states passed between a pre- or post-triggered node form a state of the optimized controller.

5. Summary and conclusions

We introduced a method to design applications for RTR architectures on a high level of abstraction. Our approach includes a system model which enables the specification of an executable system-level model.

To use RTR architectures efficiently, we apply the BMT: an abstract execution model for HW tasks. It is based on design patterns and macros provided by a design library. For this, exploitation of a system library is used for modelling, and a target library is used for an optimized target mapping and implementation.

The case study verifies that the application of an abstract task signature (comprising design patterns and macros) eases the implementation of a abstract high-level task model using a precise architecture template provided by the target library.

For future work, we plan to use the task signature for optimizing RTR designs regarding reconfiguration costs (such as configuration size, reconfiguration time, and chip area). Furthermore, we intend to realize applications of embedded or multimedia devices, such as Forward Error Correction (FEC) or Digital Signal Processing (DSP).

6. References

[1] D. Aravind, A. Sudarsanam, "High Level - Application Analysis Techniques and Architectures - to Explore Design Possibilities for Reduced Reconfiguration Area Overheads in FPGAs Executing Compute Intensive Applications", 19th IEEE IPDPS, RAW, Denver, CO, USA, April 2005

[2] V. Baumgarte, F. May, A. Nuckel, M. Vorbach, M. Weinhardt., PACT XPP - A Self-Reconfigurable Data Processing Architecture, ERSA, Las Vegas, Nevada, USA, June, 2001

[3] J. Becker, K. Brändle, M. Ullmann, "Reconfigurable HW and Intelligent Run-time Systems for Adaptive Computing", it 4/2005, Oldenbourg Verlag, Munich, Germany, 2005

[4] C. Bobda, J. Teich, "ESM: The Erlangen Slot Machine - Architecture and Development Tools", The Future of Reconfigurable Computing, DATE 05 Friday Workshop, Munich, Germany, March 2005

[5] J. Brakensiek, B. Oelkrug, M. Bücker, D. Uffman, A. Dröge, "Re-configurable Multi-standard Terminal for Heterogenous Networks", IEEE RAWCON, Boston, MA, USA, August 2002

[6] R. Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective", DATE, Munich, Germany, March 2001

[7] R. Hartenstein, "Reconfigurable Computing: the Roadmap to a New Business Model - and its Impact on SoC Design", 15th SBCCI, Brasilia, DF, Brazil, September 2001

[8] S. Klaus, S.A. Huss, "Concepts for the Control of the Complexity of Embedded System Design", it 2/2004, Oldenbourg Verlag, Munich, Germany, 2004

[9] J. Lesieutre, Z. Wang, On a Generalization of the Collatz Conjecture, 2004 RSI research report, Massachusetts Institute of Technology (MIT), Cambridge, CA, USA, 2004

[10] H. Waldner, M. Platzner, "Online Scheduling for Block-partitioned Reconfigurable Devices", DATE, Munich, Germany, March 2003

[11] Virtex 2.5V Field Programmable Gate Arrays, Product Specification, DS003-1, Xilinx Inc., April 2, 2001