Automatic Correct Scheduling of Control Flow Intensive Behavioral Descriptions in Formal Synthesis

Kai Kapp
Kai.Kapp@ira.uka.de

Viktor Sabelfeld
Viktor.Sabelfeld@ira.uka.de

University of Karlsruhe, Institute of Computer Design and Fault Tolerance (Prof. D. Schmid)
P. O. Box 6980, 76128 Karlsruhe, Germany

ABSTRACT

Formal synthesis ensures the correctness of hardware synthesis by automatically deriving the circuit implementation by behavior preserving transformations within a theorem prover.

In this paper, we present a new approach to formal synthesis that is able to handle control flow intensive descriptions. In particular, we consider here the scheduling phase, which is a central task in high-level synthesis. We describe a methodology employing a new control flow oriented representation which allows the fully automatic scheduling of control flow intensive descriptions in formal synthesis. To obtain scheduled circuits of high quality, the scheduling is guided by conventional scheduling algorithms.

Categories and Subject Descriptors:
B.5.2 Design Aids: Automatic synthesis, Hardware description languages, Optimization, Verification

General Terms: Design, Theory, Verification

Keywords: Formal Synthesis, Transformational Design, Scheduling

1. INTRODUCTION

With the increasing size and complexity of digital synthesis systems the probability of errors in the synthesis tools and, as a consequence, in their results increase as well. To ensure that the resulting design really meets the specified functionality, a verification phase is unavoidable. Yet the verification task is very expensive.

Transformational synthesis systems try to supersede the need of verifying the functional correctness of an implementation in regard to its specification by deriving the implementation from the specification exclusively through a sequence of behavior preserving transformations. Often, transformational derivation is also used for post-verification, e. g. , for the verification of the scheduling step [5].

However, the transformational synthesis approach succeeds only if the transformations actually do *not* change the behavior of the circuit. Thus, a formal proof of the correctness of the transformations is advisable. But, often, proofs are only performed in a paper & pencil style [10] which means they have to be examined by others to verify them. An actually secure way to ensure the correctness of transformations is to perform the proof within a theorem prover. For example, in [12] a transformation-based method for the verification of the scheduling of straight-line code is presented. The correctness of the used transformations has been proved [15] in the theorem prover PVS.

But even if all transformations are correct, this approach can still fail, if the transformations are not correctly implemented. The correct integration of the transformations into a synthesis tool is a critical task, since state-of-the-art synthesis tools are generally very big and complex. The proof of their correct integration is even more demanding, if not impossible.

A correct execution of the transformations can be guaranteed by embedding the whole derivation process into a theorem prover, as it is done in the formal synthesis approach: Circuits are specified by means of terms and formulae within a theorem prover and they are only modified by transformations, whose correctness has been proved in the theorem prover. Consequently, in addition to the resulting implementation each synthesis or optimization step yields automatically a proof of the functional correctness of the implementation in regard to its specification.

Yet, most formal synthesis approaches deal only with lower levels of abstraction (RT- and gate level) [11] or are restricted to acyclic data flow descriptions at the algorithmic level [8]. Also, they often cover only parts of the synthesis process, e. g. , the scheduling [12]. Furthermore, they are often not suitable for automation, but require the synthesis process to be guided interactively by the designer [9] and, thus, need the designer to be well grounded in formal methods.

By contrast, our formal synthesis system HASH (**H**igher order logic **A**pplied to **S**ynthesis of **H**ardware) covers the synthesis from the system level to the gate level. HASH is not restricted to the synthesis of pure data flow descriptions, but supports the synthesis of behavioral, control flow intensive descriptions[1] with cyclic control flows as well. Further, our goal is to fully automate the formal synthesis, so that the designer is not required to have any skills in formal methods.

We use the theorem prover HOL [6]. Circuit specifications and their implementations as well as all intermediate representations are specified as terms of higher-order logic in our formal hardware description language *Gropius* [14]. Since the constructs of Gropius are defined within HOL, the language has mathematically exact semantics. Also, the transformations used during the synthesis process are formulated in higher-order logic and their correctness is proved within the theorem prover.

In order to efficiently obtain implementations of high quality we

[1]Control flow intensive descriptions reflect a mix of arithmetical and logical operations, and control flow structures such as loops and conditional operations.

integrate existing optimization and synthesis algorithms into the formal synthesis process. To simplify their integration and to maintain their efficiency these algorithms are executed outside the theorem prover in the environments they have been originally designed for. For example, an object-oriented algorithm is implemented in an object-oriented environment. The results of these algorithms guide the selection of proper transformations to automatically reproduce their intentions within the theorem prover.

Certainly, the results of an external algorithm can be flawed, e. g. , because of a bug in that algorithm, and also there may exist errors in the code which controls the selection and application of the transformations in the theorem prover. Nevertheless, in our approach these errors lead at the worst to an abortion or an infinite running of the synthesis process, but *never* to an incorrect implementation. Our results are always proved correct.

In this paper, we present a new approach to the formal high-level synthesis (HLS). High-level synthesis maps a behavioral specification onto a structure at the register transfer level. In [3] we have already described a way to synthesize control flow oriented descriptions. There, the whole specification is transformed into a single loop with a basic block as its body. The resulting body is a pure data flow description and, so, the subsequent high-level synthesis is relatively easy. But since the explicit control flow of the original specification gets lost during the single loop transformation, it can hardly be used to optimize the quality of the scheduled circuit.

Our new approach to the synthesis of control flow intensive descriptions preserves the control flow of the circuit during all transformations at the algorithmic level. We describe how in preparation for the high-level synthesis, the original description is translated automatically into a new form in which the original control flow of the specification is represented by explicit state transitions (EST's). All further transformations at the algorithmic level operate on this representation, and, thus, the control flow is always maintained.

Then, we focus on the scheduling of control flow oriented specifications. Scheduling is one of the central tasks in high-level synthesis, in which the operations of the behavioral description are partitioned into states (or control steps).

The scheduling technique we present allows the integration of any scheduling algorithm, which is based on control flow graphs (CFG's), into the formal synthesis process. These scheduling algorithms maintain exactly the user-defined execution order of the operations in the specification [13]. Examples of corresponding algorithms are the path-based scheduling algorithm and the loop-directed scheduling algorithm [2].

Consequently, our scheduling methodology obtains results of the same quality as any CFG-based scheduling algorithm and additionally yields a proof of the functional correctness of the resulting implementation without requiring any user intervention.

In Section 2.1 we present the part of Gropius used for circuit descriptions at the algorithmic level. The circuit representation with explicit state transitions is introduced in Section 2.2 and in Section 2.3 we show how a behavioral Gropius specification is translated into this new representation. Section 3 outlines the execution of the high-level synthesis in our approach and describes in detail the scheduling step in formal synthesis followed by experimental results in Section 4. We finish with a conclusion in Section 5.

2. CIRCUIT DESCRIPTIONS IN GROPIUS

Gropius is a functional hardware description language ranging from the system to the gate level. It is strongly-typed, polymorphic and higher-order. Each construct of Gropius is defined in the theorem prover HOL. As a result Gropius has a mathematically exact semantics. A precise definition of Gropius can be found in [14].

2.1 Behavioral Circuit Description

In Gropius, the circuits at the algorithmic level are described with DFG-terms (**D**ata **F**low **G**raph-terms) and P-terms (**P**rogram-terms). *DFG-terms* represent non-recursive programs that always terminate. They are functions formalized using λ-expressions [4]. An example of a typical DFG-term is shown below:

```
λ(u,v,w). let a = u − v          in
          let b = a > w          in
          let c = MUX (b,a,w) in (b,c)
```

This DFG-term maps a triple of numerals to a Boolean paired with a numeral. The variable structure (u,v,w) following the symbol λ corresponds to the parameters of the function. The result of the function is defined by the finishing variable structure (b,c). The operation **MUX** represents a multiplexer: It selects the second argument if the first argument is true and, otherwise, the third argument. The operators used in DFG-terms correspond to elementary (mostly logical and mathematical) components, which always terminate.

Unlike DFG-terms *P-terms* represent arbitrary computable programs that may not terminate. The core syntax of P-terms is defined by the following EBNF[2] notation:

```
P-term = "WHILE"DFG-cond "DO" P-term |
         "DEF" DFG-term |
         P-term "SER" P-term |
         "IF" DFG-cond "THEN" P-term "ELSE" P-term
```

In a **WHILE**-loop the body (*P-term*) is executed as long as the condition (*DFG-cond*) evaluates to true (*DFG-conds* are DFG-terms with a Boolean result). **DEF** converts a DFG-term into a P-term. **SER** concatenates two P-terms to be executed consecutively and **IF** denotes the conditional execution of one of its branches dependent on its condition (*DFG-cond*). The Gropius example below shows an efficient algorithm for the computation of x^n:

```
DEF (λ(u,n).let v = 1 in (u,v,n)) SER
(WHILE (λ(u,v,n).n>0) DO
 (IF (λ(u,v,n).ODD n)
     THEN (DEF (λ(u,v,n).let v = u∗v  in
                          let n = n−1 in (u,v,n)))
     ELSE (DEF (λ(u,v,n).let u = u∗u  in
                          let n = n DIV 2 in (u,v,n)))))
SER DEF (λ(u,v,n).v)
```

In the following, we use lower-case letters (a, b, c, . . .) for variables of elementary types. Underscored letters ($\underline{\text{vsl}}$, $\underline{\text{vsr}}$, . . .) mark structures of such variables like, e. g. , ((a,b),(c,(d,e),f)) and lower-case italic letters (*fst*, *cond*, . . .) stand for whole DFG-terms. Upper-case italic letters (*L*, *R*, *T*, . . .) stand for P-terms.

2.2 Control Flow Oriented Representation

In this section, a new representation for circuits at the algorithmic level is introduced. Like a graph, the new representation contains states in which certain operations are executed. Transitions between these states explicitly express the control flow of the circuit. Because of the similarity of this representation to a graph it is much more suitable for the transformations generally performed during high-level synthesis than the behavioral description from Section 2.1.

An EST representation of the Gropius example in the last section is shown below. Since the first and last **DEF**-term in the behavioral description do not contain any operations they stay outside of the **LOOP**-term, which is the basis of the EST representation.

[2]Extended Backus-Naur Form

The **LOOP**-construct has two operands: a label (here 1) defining the initial state and a list representing the states of the program. The elements of the list are pairs (*s*, *P* **SER** *NXT*) consisting of a unique *state label* (*s*) and a P-term (*P* **SER** *NXT*) called the *body* of the state. To simplify matters, a state with a certain label *s* will be also referred to as *state* ⟨*s*⟩. The sub-term *P* of the body represents the part of the specification which is executed in the corresponding state. It is designated as the action of the state (*state action*). The *transition function NXT* determines the input of the next **LOOP**-iteration. This consists of the label of the succeeding state, which possibly depends on control values calculated by *P*, and the result of *P* (without the control values).

```
DEF (λ(u,n).let v = 1 in (u,v,n)) SER
LOOP 1
 [(1, DEF (λ(u,v,n).let gt = n>0 in ((u,v,n),gt))
      SER DEF (λ(x,gt).(x,MUX(gt,2,0))));
  (2, DEF (λ(u,v,n).let odd = ODD n in ((u,v,n),odd))
      SER DEF (λ(x,odd).(x,MUX(odd,3,5))));
  (3, DEF (λ(u,v,n).let mlt = u∗v in (u,mlt,n))
      SER DEF (λx.(x,4)));
  (4, DEF (λ(u,v,n).let sub = n−1 in (u,v,sub))
      SER DEF (λx.(x,1)));
  (5, DEF (λ(u,v,n).let sqr = u∗u in (sqr,v,n))
      SER DEF (λx.(x,6)));
  (6, DEF (λ(u,v,n).let div = n DIV 2 in (u,v,div))
      SER DEF (λx.(x,1)))]
SER DEF (λ(u,v,n).v)
```

The **LOOP**-construct is formally defined as follows:

```
LOOP start L ≝ DEF (λx.(x,start)) SER
               (WHILE (λ(x,s). StateExists L s) DO
                      (λ(x,s). StateCase L s x))
               SER DEF (λ(x,s).x)
```

The first **DEF**-construct introduces a variable, which holds the label of the current state, and initializes it with the label of the initial state (*start*). The last **DEF**-statement after the **WHILE**-loop finally drops this variable again. The function **StateExists** in the **WHILE**-condition checks whether a state with the label *s* exists in the list *L*. If not (as the label 0 in the example), it is an *exit state* and the loop terminates. Otherwise, the body of the **WHILE**-loop is executed. There, the function **StateCase** selects the body of the state ⟨*s*⟩ from the list *L*. This body is executed on x (in connection with this, x is also designated as the *input* of the state ⟨*s*⟩). The execution of the body yields the label and the input of the successor of state ⟨*s*⟩.

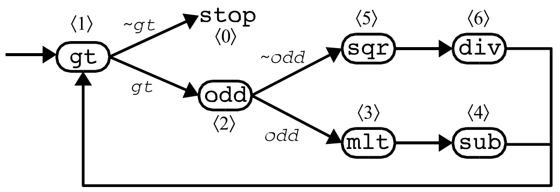

Figure 1: Control Flow Graph

The control flow graph corresponding to the EST representation of the Gropius example is shown in Figure 1. The names used for the states in this graph refer to states in the EST representation (EST states) containing the corresponding operations, e. g., the state “odd” refers to the EST state ⟨2⟩.

2.3 Translating Behavioral Descriptions into EST Representations

In this section we describe the translation of a behavioral description *SPEC* into a control flow oriented EST representation. The translation starts with the application of the following theorem:

```
⊢ (start ≠ exit) ⇒
   SPEC = LOOP start [(start, SPEC SER DEF (λx.(x,exit)))]
```

The resulting **LOOP**-list contains a single state with the label *start*, the initial state. The action of the state corresponds to the original behavioral description *SPEC*. The succeeding state determined by the transition function **DEF** (λx.(x,*exit*)) is unconditionally *exit*. Since there is no state with a label *exit* in the list, the loop terminates. So, this EST representation corresponds exactly to the original behavioral specification of the circuit.

In our approach each transformation is based on a universally valid theorem, which we have already proved in the theorem prover. To perform a transformation, the free variables in the theorem are instantiated with concrete values in order to match the current situation. For example, in the theorem above the variables *start* and *exit* are replaced by concrete labels and the variable *SPEC* is instantiated with the behavioral specification of the circuit.

Theorems often carry some preconditions with them, which are automatically proved by our synthesis system in the course of the transformation. Of course, this requires the planned transformation to be valid, e. g. , the theorem above is only valid, if the labels assigned to *start* and *exit* are really unequal.

Each transformation results in a concrete theorem, which states that the original circuit description is equal to the transformed description and, thus, the original description can be replaced by the new description.

The following transformations describe how the states in the EST representation are split up at the control constructs (**SER**, **IF** and **WHILE**) in the actions of their bodies. The affected control constructs are eliminated and their control flow is replaced with appropriate explicit state transitions (EST's) between the states resulting from the splits.

To simplify matters, the figures accompanying the corresponding transformations just show the involved original state(s) and the state(s) resulting from the transformation. Actually, most transformations operate on the whole **LOOP**-expression. The term *GO s* stands for a transition function that selects ⟨*s*⟩ unconditionally as its succeeding state. The function *NXT* represents the original transition function of the state under consideration.

Splitting SER

A state with an action *A* **SER** *B* is replaced by two states whose actions correspond to the left and right operand of the **SER**-construct, respectively. The new states are connected with an unconditional jump:

$$(s, A\ \textbf{SER}\ B\ \textbf{SER}\ NXT) \quad \overset{split}{\Longrightarrow} \quad \begin{array}{l}(s,\ \ A\ \textbf{SER}\ GO\ s_{new})\\ (s_{new}, B\ \textbf{SER}\ NXT)\end{array}$$

If the action of a state is a **DEF**-expression containing several operations, then the state can also be split (by the transformation above) after the split-up of the **DEF**-expression itself according to the following example:

```
DEF (λ(u,v,n).let v = u∗v in
              let n = n−1 in (u,v,n))
  ⟹
DEF (λ(u,v,n).let v = u∗v in (u,v,n)) SER
DEF (λ(u,v,n).let n = n−1 in (u,v,n))
```

More details about splitting **DEF**-terms can be found in [7].

Splitting IF

If the action of a state is an **IF**-expression, then, firstly, its transition function *NXT* is moved into both branches of the **IF**-expression by means of the following theorem:

$$\boxed{\begin{array}{l}\vdash (\textbf{IF}\ cnd\ \textbf{THEN}\ TB\ \textbf{ELSE}\ EB)\ \textbf{SER}\ NXT = \\ \quad \textbf{IF}\ cnd\ \textbf{THEN}\ (TB\ \textbf{SER}\ NXT)\ \textbf{ELSE}\ (EB\ \textbf{SER}\ NXT)\end{array}}$$

After that, the resulting **IF**-expression is split as follows:

$$\boxed{\begin{array}{l}(s, \textbf{IF}\ cnd\ \textbf{THEN}\ TBNXT\ \textbf{ELSE}\ EBNXT) \\ \quad \overset{split}{\Longrightarrow} \\ (s, \textbf{DEF}\ (\lambda \underline{x}.(\underline{x},cnd\ \underline{x}))\ \textbf{SER}\ \textbf{DEF}\ (\lambda(\underline{x},c).(\underline{x},\textbf{MUX}(c,s_T,s_E)))) \\ (s_T,\ TBNXT) \\ (s_E,\ EBNXT)\end{array}}$$

The resulting state replacing the original state $\langle s \rangle$ determines dependent on the condition *cnd* one of the two new states $\langle s_T \rangle$ or $\langle s_E \rangle$ as its successor and transmits its input unchanged to the selected state. The bodies of the new states correspond to the branches (*TBNXT* and *EBNXT*) of the original **IF**-expression.

Splitting WHILE

A state whose action is a **WHILE**-expression is at first transformed as follows:

$$\boxed{\begin{array}{l}(s, (\textbf{WHILE}\ cnd\ \textbf{DO}\ LBODY)\ \textbf{SER}\ NXT) \quad \Longrightarrow \\ (s, \textbf{IF}\ cnd\ \textbf{THEN}\ (LBODY\ \textbf{SER}\ GO\ s)\ \textbf{ELSE}\ NXT\ )\end{array}}$$

The resulting **IF**-expression tests the same condition *cnd* as the **WHILE**-expression. If it is true, the original body *LBODY* of the **WHILE**-loop is executed and the next state is unconditionally the same state $\langle s \rangle$ again. If the condition is false the original transition function *NXT* is executed.

After this transformation the action of the state is an **IF**-expression and the state can be split as described above.

The transformations presented in this section require any newly created state to have a unique label, i. e., it has to be unequal to all labels of the states in the **LOOP**-list as well as to all labels used for exit states. In our system all these preconditions are proved automatically in the course of the application of the corresponding transformations.

As a result of the iterative application of the split transformations the actions of all states are finally **DEF**-constructs. This is a premise for the following transformations.

Sometimes, so-called *transit states* can arise from the application of the split transformations. These states transmit their input unchanged to their successors. For instance, the state $\langle s \rangle$ resulting from the split of the **IF**-expression above transmits its input ($\underline{x}$) unchanged to its successor $\langle s_T \rangle$ or $\langle s_E \rangle$.

Subsequent to the split transformations existing transit states are eliminated by merging them with their predecessors. In case of the initial state it is merged with its successors. The resulting EST representation complies exactly with the control flow graph of the original description of the circuit. This representation builds the starting point for the scheduling step described in Section 3.1.

Merging States

The merge transformation of two states $\langle s_A \rangle$ and $\langle s_B \rangle$ integrates the body of state $\langle s_B \rangle$, i. e., its action **DEF** *actb* and transition function **DEF** *nxtb*, into $\langle s_A \rangle$. If state $\langle s_B \rangle$ has only state $\langle s_A \rangle$ as potential predecessor, it is removed during this transformation.

In the body of the resulting state $\langle s_A \rangle$, at first the body of the original state $\langle s_A \rangle$ is executed. Its result is stored in the succeeding **DEF**-term in the variable structure ($\underline{xa}$,*suca*). Thereupon, the body of state $\langle s_B \rangle$ is executed on $\underline{xa}$ and the result is assigned to the variable structure ($\underline{xb}$,*sucb*). At that point, ($\underline{xa}$,*suca*) holds the result of the execution of the original state $\langle s_A \rangle$ and ($\underline{xb}$,*sucb*) the result corresponding to the *successive* execution of the original states $\langle s_A \rangle$ *and* $\langle s_B \rangle$. Finally, a **MUX** selects one of these results dependent on whether the state $\langle s_B \rangle$ turns out to be the successor of state $\langle s_A \rangle$, i. e., whether ($suca = s_B$) holds.

$$\boxed{\begin{array}{l}(s_A, \textbf{DEF}\ acta\ \textbf{SER}\ \textbf{DEF}\ nxta) \\ (s_B, \textbf{DEF}\ actb\ \textbf{SER}\ \textbf{DEF}\ nxtb) \\ \overset{merge}{\Longrightarrow} \\ (s_A, \textbf{DEF}\ acta\ \textbf{SER}\ \textbf{DEF}\ nxta\ \textbf{SER} \\ \quad \textbf{DEF}\ (\lambda(\underline{xa},suca). \\ \qquad \textbf{let}\ (\underline{xb},sucb) = nxtb(actb\ \underline{xa})\ \textbf{in} \\ \qquad \textbf{MUX}(suca = s_B,(\underline{xb},sucb),(\underline{xa},suca)))) \\ (s_B, \textbf{DEF}\ actb\ \textbf{SER}\ \textbf{DEF}\ nxtb)\end{array}}$$

Each merge transformation is followed by a conversion which transforms the body of the resulting state $\langle s_A \rangle$ into the common form **DEF** *action* **SER** **DEF** *transition* again.

3. FORMAL HIGH-LEVEL SYNTHESIS

Prior to performing the high-level synthesis, in our approach, some source code optimizations are carried out as they are known from the software domain [1] (e. g., common subexpression elimination and dead code elimination [7]). Then, the high-level synthesis starts with an analysis of the control and data flow of the behavioral Gropius specification of the circuit. The flow information is passed to the external environment and is transformed there into data structures (generally graphs) suitable for the external synthesis algorithms. The external algorithms are executed and, thus, all design decisions regarding the high-level synthesis tasks scheduling, binding and allocation are made dependent on constraints given by the user. The results are returned to the formal environment, where the description of the circuit is modified by proved-correct transformations in such a way that it exactly reflects the design decisions made by the external algorithms.

3.1 Scheduling in Formal Synthesis

In this paper, we focus on the scheduling step, which in our approach is the first of the high-level synthesis steps to be reproduced in the formal environment. Please note that a fixed reproduction order of the synthesis steps in the formal environment does not affect the quality of the design. The external algorithms that actually make the design decisions can still be executed in any desired order.

For the external scheduling algorithm, the control and data flow information of the original Gropius specification is presently transformed into a CFG (see Figure 2). The scheduling algorithm we have implemented is based on the *loop-directed scheduling algorithm* (LDS). This algorithm tries to minimize the average execution time of the schedule through the use of loop unrolling.

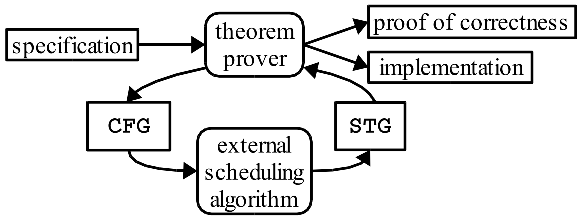

Figure 2: Scheduling in Formal Synthesis

After the execution of the scheduling algorithm its result is returned to the formal environment in form of a *state transition graph*

(STG). An STG is a directed graph whose nodes represent states and whose edges represent transitions between states. Nodes in the STG hold information about the operations executed in the corresponding state, and edges capture the conditions under which a state transition takes place.

Section 2.3 described how the specification of a circuit is translated into an EST representation corresponding to the control flow graph of the behavioral specification. This representation is now converted through a sequence of behavior preserving transformations into another EST representation which complies with the STG provided by the external scheduling algorithm. That way, the scheduling decisions made by the external algorithm are exactly reproduced in the formal environment.

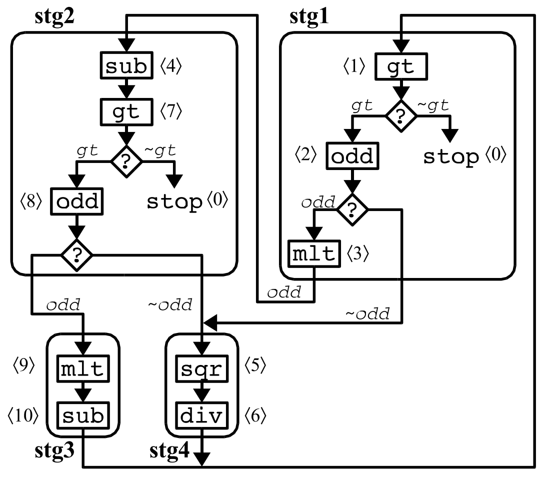

Figure 3: A State Transition Graph

Figure 3 shows an STG resulting from the execution of the external LDS algorithm on the Gropius program from Section 2.1. Its initial state is **stg1**. The names of the operations in the STG states refer to certain states in the EST representation containing these operations. For example, "`sqr`" is a name referring to the operation performed in state ⟨5⟩.

The first step in the reproduction of the STG is the copying of EST states. For each operation which occurs in more than one STG state (such as, e. g. , the operation `odd`) a proper number of copies of its corresponding EST state is made.

The *copy transformation* adds a new state with a unique label and the same body as its original state to the **`LOOP`**-list. Further, the *similarity* of the new state to its origin is memorized. Two states are *similar* , if (i) their bodies are identical, or (ii) for any input their bodies yield the same result except for their succeeding states, which just have to be similar.

In the example, the states ⟨1⟩, ⟨2⟩, ⟨3⟩, and ⟨4⟩ are copied once, because each of the operations `gt`, `odd`, `mult` and `sub` occur twice in the STG. The copies get the labels `7`, `8`, `9`, and `10`, respectively (see Figure 3).

Now, each operation in the STG is assigned to a certain instance of a corresponding EST state. For example, the EST states ⟨3⟩ and ⟨9⟩, the copy of ⟨3⟩, correspond to the operation `mult` in the STG. Here, the operation `mult` in **stg1** is assigned to EST state ⟨3⟩ and its occurence in **stg3** is assigned to EST state ⟨9⟩.

Athat, the transition functions of the EST states are modified in order to reproduce the structure of the STG. All EST states are redirected to point to the correct instances of their successing states.

For instance, EST state ⟨9⟩ is a copy of state ⟨3⟩. Both states (still) have identical bodies. Their successor is unconditionally state ⟨4⟩. Now, according to the structure of the STG, state ⟨9⟩ has to be redirected to another copy of state ⟨4⟩, namely state ⟨10⟩.

The *redirection transformation* allows the substitution of a transition function **`DEF`** *nxt* of a state (*s*, *P* **`SER DEF`** *nxt*) by any other transition function **`DEF`** *nxt'*, if for any x the results of *nxt* x and *nxt'* x are equal except for the resulting succeeding states, which just have to be similar.

In practice, the redirection transformation generates a new transition function *nxt'* from the original transition function *nxt* by replacing some of the labels of the successors with the labels of proper similar states. As a result, it is very easy to prove that for any x the results of *nxt* x and *nxt'* x are equal except for the successors determined, which are similar.

The scheduling step is completed by merging all EST states (using the transformation described in Section 2.3) that belong to the same STG state. The resulting EST representation of the circuit complies exactly with the STG given by the external scheduling algorithm.

Starting from the example in Section 2.2, a reproduction of the STG in figure 3 results in the following EST representation. EST state ⟨1⟩ corresponds to STG state **stg1**, state ⟨4⟩ to **stg2**, state ⟨9⟩ to **stg3** and state ⟨5⟩ to **stg4**:

```
DEF (λ(u,n).let v = 1 in (u,v,n)) SER
LOOP 1
 [(1, DEF (λ(u,v,n).let gt = n>0 in
                    let odd = ODD n in
                    let mlt = u∗v in
                    let v' = MUX(gt∧odd,mlt,v)
                        in ((u,v',n),gt,odd))
      SER DEF (λ(x,gt,odd).(x,MUX(gt,MUX(odd,4,5),0))));
  (4, DEF (λ(u,v,n).let sub = n−1 in
                    let gt = sub>0 in
                    let odd = ODD sub in ((u,v,sub),gt,odd))
      SER DEF (λ(x,gt,odd).(x,MUX(gt,MUX(odd,9,5),0))));
  (5, DEF (λ(u,v,n).let sqr = u∗u in
                    let div = n DIV 2 in (sqr,v,div))
      SER DEF (λx.(x,1)));
  (9, DEF (λ(u,v,n).let mlt = u∗v in
                    let sub = n−1 in (u,mlt,sub))
      SER DEF (λx.(x,1)))]
SER DEF (λ(u,v,n).v)
```

The copy and redirection transformations allow the reproduction of any part of the original CFG in the EST representation. This includes, e. g. , the possibility of loop unrolling: All states of a whole loop are copied and, then, the control flow is redirected at the end of the original loop to flow over the newly created copies.

Thus, with the presented methodology the result of any CFG-based scheduling algorithm, which maintains the user-defined execution order of the single operations, can be reproduced starting from an EST representation of the CFG of the circuit.

In the final EST representation each state corresponds to the operations performed in a cycle on the RT-level. The input type of the states combined with the type of the state labels correspond to the required registers.

4. EXPERIMENTAL RESULTS

The formal scheduling technique presented in this paper has been applied to the several well-known examples, which we have translated into our hardware description language Gropius. A selection of them is presented in Table 1: The programs `diffeq` and

Program		Translation into EST representation				Reproduction of external STG				
name	operations	splits	merges	EST states	time	copies	redirections	merges	STG states	time
diffeq	11	5	7	3	1,3 s	2	3	2	2	0,9 s
bin-GCD	16	20	20	13	5,5 s	18	23	22	9	13,9 s
bubble	19	15	15	10	4,4 s	7	9	8	8	4,4 s
fibonacci	22	19	8	18	4,0 s	12	12	17	13	15,9 s
kalman	53	44	45	28	51,1 s	43	58	57	14	194,4 s
fp_add	61	39	28	28	21,1 s	24	32	24	28	40,2 s

Table 1: Experimental Results

`kalman` are taken from the HLS Benchmark Set collected by Dutt and Panda. `bin-GCD` is a variant of the binary Euclidean GCD algorithm, `fp_add` adds two floating point numbers (it corresponds to the function `__addsf3` in `floatlib.c` of the gcc-3.3 with all macros expanded) and `bubble` is the bubble sort algorithm. `fibonacci` is a fast algorithm for the calculation of fibonacci numbers. The second column of the table lists the number of operations in each program.

The second part of the table shows the numbers of split and merge transformations that were required to transform the corresponding program into an EST representation with the listed number of states as described in Section 2.3. In contrast with the example of Section 2.2 the number of EST states is always less than the number of operations, since operations within a basic block that only appear combined in the STG states are actually not unnecessarily separated. The run-times refer to the duration of the formal transformations in HOL. We have used HOL98-Taupo6 running on an Athlon XP1700+ under Linux 2.4-20.

The third part of the table shows the results concerning the reproduction of the STG. The numbers of the required copy, redirection and merge transformations are given together with the number of states in the STG as well as their total run-time (see Section 3.1).

The program `kalman` takes the longest time, since it reaches the highest number of states during the transformations (28 EST-states plus 43 copies). This affects primarily the copy and merge transformations.

The synthesis methodology we have presented in this paper yields guaranteed correct results. A proof is given with the implementation, which states that the implementation fulfills the specification. Therefore, the run-times in the table have to be compared with a conventional synthesis process followed by an exhaustive simulation, which is for complexity reasons often impossible.

5. CONCLUSION

In this paper, we have presented a new approach to the formal synthesis of control flow intensive descriptions. Through the introduction of a control flow oriented circuit representation, the control flow of the original specification is maintained during all transformations at the algorithmic level. As a result, the optimization potential of the synthesis has been considerably increased compared with other formal synthesis approaches. The similarity of the new representation to graphs simplifies the integration of conventional high-level synthesis algorithms into the formal synthesis process.

Further, we have described a methodology to automatically reproduce the design decisions made by conventional CFG-based scheduling algorithms in our formal synthesis system. Since we exactly reproduce the results of these external algorithms, we obtain implementations of the same quality. In addition, our system produces fully automatically a formal proof of the functional correctness of the scheduled circuit in regard to its specification. So, the resulting implementation is guaranteed correct.

Future work includes the reproduction of the results of the remaining HLS tasks. Just as in the case of the scheduling step, the synthesis decisions will also be made outside the formal environment and subsequently guide the selection and application of behavior preserving transformations within the theorem prover to securely obtain an implementation of high quality.

6. ACKNOWLEDGMENTS

The authors thank the reviewers who gave valuable advice on how to improve this paper.

7. REFERENCES

[1] A. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques and tools*. Addison Wesley, 1986.

[2] S. Bhattacharya, S. Dey, and F. Brglez. Performance analysis and optimization of schedules for conditional and loop-intensive specifications. In M. Lorenzetti, editor, *Proceedings of the 31st Conference on Design Automation*, pages 491–496. ACM Press, June 1994.

[3] C. Blumenröhr and V. Sabelfeld. Formal synthesis at the algorithmic level. In *Proceedings of Correct Hardware Design and Verification Methods: CHARME '99*, volume 1703 of *LNCS*, pages 187–201. Springer, Jan. 1999.

[4] R. E. Davis. *Truth, deduction and computation: logic and semantics for computer science*. Computer Science Press, New York, 1 edition, 1989.

[5] H. Eveking, H. Hinrichsen, and G. Ritter. Automatic verification of scheduling results in high-level synthesis. In *Proceedings of the conference on Design, automation and test in Europe*, pages 59–64. ACM Press, Mar. 1999.

[6] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[7] K. Kapp and V. Sabelfeld. Dead code elimination in formal synthesis. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 121–130. Rolf Drechsler, Feb. 2003.

[8] M. Larsson. An engineering approach to formal digital system design. *The Computer Journal*, 38(2):101–110, 1995.

[9] M. Larsson. Improving the result of high-level synthesis using interactive transformational design. In *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, pages 299–314. Springer-Verlag, 1996.

[10] J. M. Mendas, R. Hermida, M. C. Molina, and O. Pealba. Efficient verification of scheduling, allocation and binding in high-level synthesis. In *Proceedings of the Euromicro Symposium on Digital Systems Design*, pages 308–315. IEEE Computer Society, 2002.

[11] R. Kumar, C. Blumenröhr, D. Eisenbiegler, and D. Schmid. Formal synthesis in circuit design - A classification and survey. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design. First International Conference, FMCAD'96*, number 1166 in Lecture Notes in Computer Science, pages 294–309, Palo Alto, CA, USA, Nov. 1996. Springer.

[12] R. Radhakrishnan, E. Teica, and R. Vemuri. Verification of basic block schedules using RTL transformations. In *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 173–178. Springer-Verlag, Sept. 2001.

[13] M. Rahmouni and A. A. Jerraya. Formulation and evaluation of scheduling techniques for control flow graphs. In *Proceedings of the Conference on European Design Automation*, pages 386–391. IEEE, 1995.

[14] V. Sabelfeld and K. Kapp. Numeric types in formal synthesis. In M. Broy and A. Zamulin, editors, *5th International Andrei Ershov Memorial Conference, "Perspectives of System Informatics"*, volume 2890 of *LNCS*, pages 79–90. Springer, July 2003.

[15] E. Teica and R. Vemuri. Mechanizing in higher-order logic proofs of correctness and completeness for a set of RTL transformations. In R. J. Boulton and P. B. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, Informatics Report Series, pages 368–383, Sept. 2001.