Copyright (C) 1995 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or permissions@acm.org

From VHDL to Efficient and First-Time-Right Designs: A Formal Approach

Peter F.A. Middelhoek
University of Twente
Department of Computer Science
P.O. Box 217, 7500 AE Enschede, The Netherlands
e-mail: pfam@cs.utwente.nl
<http://wwwspa.cs.utwente.nl/aid/aid.html>

Sreeranga P. Rajan[1]
Fujitsu Laboratories of America
3350 Scott Blvd. #34
Santa Clara, CA 95054-3104, USA
e-mail: srajan@fla.fujitsu.com
<http://www.csl.sri.com/sree/sree.html>

Abstract

In this paper, we provide a practical transformational approach to the synthesis of correct synchronous digital hardware designs from high-level specifications. We do this while taking into account the complete life cycle of a design from early prototype to full custom implementation. Besides time-to-market, both flexibility with respect to target architecture and efficiency issues are addressed by the methodology. The utilization of user-selected behavior-preserving transformation steps ensures first-time-right designs while exploiting the experience, flexibility, and creativity of the designer.

To ensure that design transformations are indeed behavior-preserving a novel mechanized approach to the specification and verification of design transformations on control data flow graphs which is independent of a specific behavioral model or graph size has been developed.

As a demonstration of an industrial application we use a video processing algorithm needed for the conversion from a line-interlaced to progressively scanned video format. Both a video signal processor-based prototype implementation as well as a very efficient full custom implementation are developed starting from a single high-level behavioral specification of the algorithm in VHDL. Results are compared with those previously obtained using different tools and methodologies.

1. Introduction

The primary motivation for the development of tools for the automatic synthesis of behavior-level specifications into efficient RT- or gate-level descriptions is the reduction of time-to-market while dealing with increasing design complexity. In this paper we take a broad view on high-level synthesis of synchronous digital systems and define it to include what some call algorithmic or architectural synthesis. This in contrast to [WaC95] which limits high-level synthesis to the scheduling, allocation, and binding problem.

The design time of a system consists of the time required for synthesis and achieving correctness [HLD90]. To reduce design time most research in high-level synthesis concentrates on the *automatic* synthesis of efficient implementations of well-defined behavioral specifications using a parameterized target architecture consisting of functional, memory, and interconnection units. We call this approach, which is based on fixed application domain, target architecture pairs: a *vertical* approach to synthesis. It is, however, well known that for complex designs verification can take from 30% to 80% [Pre92] of the total design time. Therefore, the methodology we propose focuses on minimizing the verification effort by eliminating the need for verification between the behavioral/algorithmic and logic level by utilizing formal methods to achieve *correctness by construction*. A combination of formally verified, *designer-driven* synthesis steps is used to reduce the number of design iterations and to obtain more efficient designs.

Novel aspects discussed in this paper include: life cycle support through the construction of target architectures instead of using fixed pairs of application domains and target architectures, the use of designer-selected optimization, refinement, and assignment transformations to support full-scale transformational design, and a different view on the implementation suggestion contained in specifications. Furthermore, we propose a mechanized transformation verification approach independent of graph structure, size, and behavior model, and the use of a correctness definition that enforces just behavioral refinement rather than the stricter equivalence.

The next section addresses correctness in high-level synthesis. Additional requirements of a design methodology imposed by the need to support the complete life cycle of a system are discussed in Section 3. Section 4 discusses and motivates the choices made in our methodology. Details on the design representation, and the classification of transformations are the topic of Section 5. In Section 6 the design of an example video processing algorithm is shown. A novel approach to the verification of design transformations is discussed in Section 7.

2. Correctness

Achieving correct designs is crucial in high-level synthesis and played the central role in the development of our methodology and design system TRADES [Mid94a, Mid94b, MMM95, Mid95].

We identify four major levels in the design of digital systems: *requirements*, *specification*, *implementation*, and *realization*. The system requirements capture the desired properties of a system *informally*, often in natural language. At the specification level system requirements are *formalized* and described in hardware description languages (HDLs) with unambiguous (formal) semantics. Through a process of stepwise refinement the final implementation is derived from the specification. The result of a design step can be considered an intermediate implementation which functions as a specification for the next step. In this paper we reserve the term 'implementation' for abstract design representations that focus on describing the required system behavior in such a way that it can be realized *efficiently*. Although in our case there is no fundamental difference between specification and implementation the former is more oriented towards behavior while the latter emphasizes structure of the design. In case of an executable specification, the specification can also be considered the initial implementation. We use the term 'realization' to refer to the actual product after manufacturing, i.e. the *physical structure* exhibiting the desired behavior.

[1] Formerly at SRI International, Computer Science Laboratory, Menlo Park, CA 94025, USA.

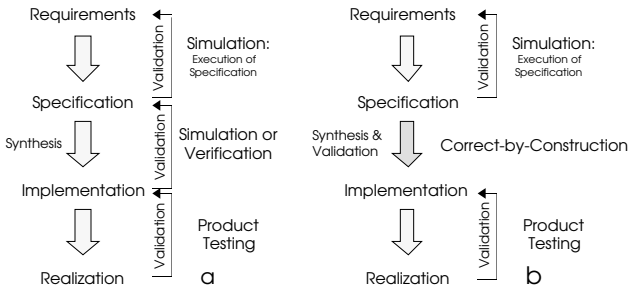

Figure 1. Top-down design methodology

Correctness in design deals with maintaining consistency between these levels. Figure 1 illustrates this for a top-down design methodology. The specification is usually written in HDLs such as VHDL [VHD88] or Silage [Hil85]. The only practical way to validate the behavior of a specification against the user requirements is through the execution of the specification (i.e. simulation) using a test suite as input. This process cannot guarantee correctness because it is not exhaustive. Execution requires an implementation and therefore structure. This initial structure is called the *implementation suggestion*. For this reason specifications, in practice, are not purely behavioral but algorithmic. In fact an executable specification can be viewed as an initial, though most likely not very efficient, implementation.

For the next step from specification to implementation three approaches towards design correctness can be identified: *simulation*, *verification,* and *correctness by construction* [Eve87]. Simulation is the most popular but cannot be used to assure the correctness of a design with respect to its specification since exhaustive simulation quickly becomes infeasible for even relatively small designs. The latter two methods are formal and can be used to guarantee the correctness of an implementation.

Formal verification validates the implementation with respect to the specification. A disadvantage of formal verification is that errors are caught only after the design stage. As a consequence there will still be a need for time-consuming debug cycles [MuH95]. [SFC94] for instance, uses post-verification techniques based on a geometric model to verify the application of loop transformations. Incremental verification can be used to avoid some of the problems associated with post facto verification but it puts the burden of verification on the circuit designer or requires integration of automatic verification tools. The Lambda/Dialog system is an example of this latter approach. A more thorough overview of verification techniques including the Lambda system can be found in [MuH95] contains.

In the third, transformation-based method, the design flow from specification to implementation consists of the consecutive application of *pre-proven behavior-preserving transformations* in a *compositional design description*. The use of pre-proven transformations removes the burden of verification from the chip designer. *Compositionality* is the property that the behavior of the whole is equal to the composition of behaviors of its parts. Therefore, replacing a part with a part that has the same behavior is guaranteed not to change the behavior of the whole. This property is essential for transformational design. In Section 7.3 a formal definition of compositionality is given.

Figure 2 shows a data flow graph representation of a design before and after the application of the distributivity transformation. The circular nodes indicate operations, the edges communication, and the boxes constant values. The distributivity transformation allows for the implementation of the multiplier and adder by means of two shifters and an adder. The preconditions of the transformation define when the transformation may be applied from the point of view of correctness. They do not impose a notion of efficiency. For the distributivity transformation the preconditions define, among other things, that the transformation requires a '+' and '*' node, that the output of the '+' is connected to the right input of the '*' node, and that the types of the signals entering and leaving the nodes should be sufficiently wide to avoid overflow. Theorem 7.3 in Section 7.4 gives an example of transformation preconditions. The transformation changes the structure of the graph locally in the *transformation domain* and thereby the *internal behavior* of both transformation domain and graph. Transformations are defined to keep the *external behavior* (I/O behavior) on the boundary of the transformation domain the same. Correctness of a transformation is the same as preservation of external behavior. The external behavior of the graph (on the input/output nodes) remains the same due to compositionality of behavior.

The use of local behavior-preserving transformations and compositionality can be exploited when dealing with changing specifications or partitioning the design for co-development. For instance, if we modify the specification by changing the value of the constants in Figure 2, the distributivity transformation is still valid because the change is outside the transformation domain. Changing the constants after applying a constant propagation transformation on the constants and adder would invalidate that transformation, because the change is inside the domain, and thus requires some 'redesign'. The formal definitions of transformations and transformation domains make it possible to track the influence of changes in the specification on the correctness (not necessarily on the optimality) of the existing design and limit the amount of redesign.

When defining transformations special care has been taken to ensure that correct data types are used after the application of the transformation so that no overflow conditions can occur. In transformational design correctness is guaranteed and is an *integral* part of the top-down design flow (Figure 1b). Since it is only possible to design correct implementations the designer can concentrate on efficiency.

The correctness of the implementation depends on the ability to build hardware and software that behaves according to the model assumed for the implementation. In our case this is a straightforward synchronous digital model which abstracts from time and assumes that combinational logic stabilizes before a clock edge arrives. To assure the

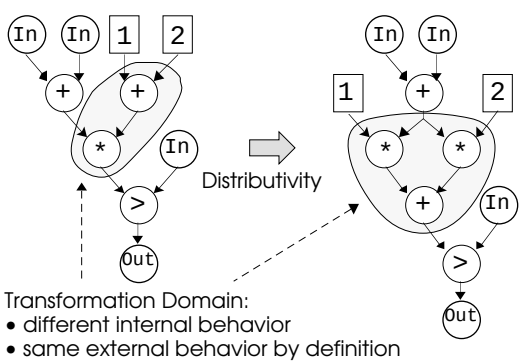

Figure 2. Transformation in a compositional design representation.

correctness of the physical realization with respect to the implementation and the abstract model we must, for product testing, resort to inexhaustive testing. The use of techniques known as 'design for testability' during the step from specification to implementation might significantly improve the test coverage while minimizing testing time.

For complex designs, such as large microprocessors, many iterations of silicon are currently needed to assure both correctness of the implementation and conformance of circuit timing in the realization with the synchronous model while meeting performance constraints. This despite the fact that just as many engineers are employed for verification as are used for design tasks.

In this section we have discussed two approaches that can guarantee correctness of implementations with respect to their specification. As argued in the introduction the design time and thereby time-to-market can be significantly reduced by eliminating debug cycles. This results in a preference for an approach towards synthesis that achieves correctness by construction.

3. Life Cycle-Imposed Constraints

While minimizing design time is certainly a major issue we feel that the importance of minimizing time-to-market is sometimes overrated, certainly in the field of high-level synthesis. [Ast95] shows that we are not alone in this observation. We believe that when taking the complete life cycle of a design into account, *flexibility* and *efficiency* of the design method are at least as important. If we consider a good design, one that provides the right functionality at the right time at the right price, it acknowledges that the balance between functionality, design time, and efficiency changes during the life of the product.

Because of changing requirements different target architectures and design flows are needed. This is in contrast to what current 'vertical' approaches in synthesis offer, which divide the design space in slices of application domain and target architecture pairs.

Rapid system prototyping is used to quickly evaluate a system's functionality or performance. To achieve the desired quick turn-around times and to reduce the cost of making many design iterations, general-purpose reusable and programmable hardware is required. The ability to model a system at different levels of detail is crucial in this phase to speed up simulation times [Hen95]. Products initially often have strict time-to-market requirements in order to capture market share. In product volume this phase represents only a few percent of total sales. Later on in its life cycle other considerations become important. High-end features move down towards the domain of mass production where implementation cost becomes the dominant driving force.

Marketing issues might demand such things as performance differentiation or low power consumption, requiring parallel architectures in combination with voltage-scaling techniques. At the same time process technology is improving which allows for more sharing of hardware and even more design trade-offs. Fully exploiting these different requirements and possibilities, while maintaining the same functional behavior, calls for completely different target architectures. Many examples of such retargeted designs can be found in the computer and consumer electronics markets. Unfortunately this differentiation is not supported by existing tools for high-level synthesis— in fact, it opposes their vertical approach. In Section 6 we will discuss an example of retargeting.

When the design has matured changes are mainly cost-driven. Every small improvement in efficiency at this stage can result in a significant improvement in profits because of the high volume. Modifications consist of small incremental upgrades in functionality, small changes to the specification, and further integration and optimization to reduce system cost. This is typically a 'replacement market' where existing products are substituted by lower-cost versions. Another important difference is that designers have experience with the design since they have already worked on or seen previous generations. A design methodology should exploit this experience.

To support the complete life cycle, a design methodology must be very flexible with respect to the level of optimization and the supported target architectures. We classify a methodology that supports multiple target architectures as 'horizontal'.

4. Design Methodology

Time-to-market, efficiency, and flexibility with respect to target architecture have been identified as the three issues that must be tackled by a design methodology that supports the complete life cycle of a system. In Section 2 we addressed design correctness and proposed the use of formal and in particular transformation-based methods to reduce design time. In this section we will investigate how the other objectives can be achieved and what choices we have made in the development of our design system, TRADES.

4.1. Target Independence

Most of the dependence on a specific target architecture in existing design systems is concentrated in the calculation of the design cost estimators and the ordering of design steps in optimization algorithms and heuristics. The actual basic design steps (i.e. transformations) used in different systems, such as retiming, scheduling, and constant propagation, are very similar and can be reused in different target-specific design flows after proper generalization. These generalized transformations, when used in different order and combinations make it possible to *construct* multiple target architectures from an algorithm. Existing design systems, because they use a fixed sequence of transformations, can only map onto a predefined parameterized target architecture. In fact, because these systems are optimized for a particular application domain the input (i.e. the specification) indirectly determines the target architecture. A methodology in which the designer interactively selects which transformation to apply where enables the construction of different target architectures from a specification. The example of Section 6 will demonstrate how both a processor based and full custom architecture can be used. Other examples of architecture construction have been discussed in [Mid95].

Generalizing design cost estimators and the ordering of design steps in heuristics to make them target-independent seems to be much more difficult. Implementation cost expressed in area, speed, and power is directly related to the target architecture, as is the ordering of design steps. For these reasons design cost estimators are no part of TRADES. If required by the designer they will be provided as add-on analysis tools.

The *control data flow graphs* (CDFG) often used as design representation in high-level synthesis [CaT89] are not limited to a specific architecture. On the contrary, CDFGs are very flexible and have been used to efficiently represent both hardware and software structures [Mid95]. Much

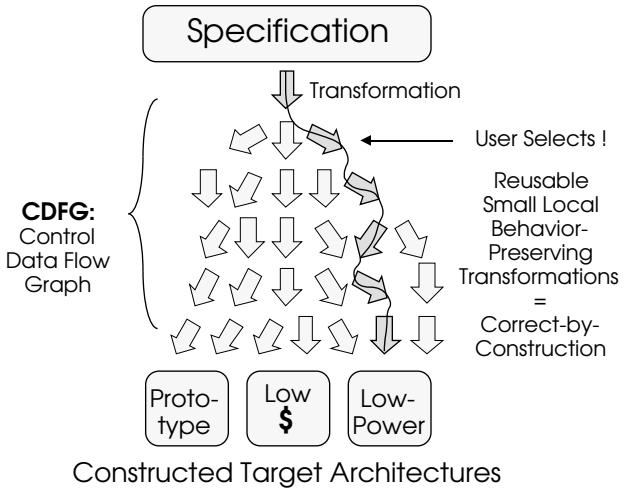

Figure 3. Transformational design

literature is available on the application of different data flow models to the design of digital signal processing systems. An excellent overview can be found in [LeP95].

4.2. Efficiency

Achieving efficient and preferably optimal solutions is a difficult problem in high-level synthesis and much effort has been put in developing good optimization algorithms. Because of the size of both designs and design space, and the nature of the optimization problems (many are NP-hard), optimal solutions can in general not be guaranteed.

We think, and early results confirm this [Mid94a, Mid94b, MMM95, Mid95], that better, more efficient designs can be obtained by exploiting the flexibility, creativity, and experience of the designer in an interactive design environment. As noted earlier, in real life a designer most often already has experience with a design or similar designs. Automatic push-the-button synthesis tools do not exploit this knowledge. For this reason, we have chosen to develop a user-centered design methodology, which as a side effect should ease acceptance by designers. Within TRADES this implies that the selection of design steps (i.e. transformations) and where to apply them is done manually by the designer, aided by a rule-based system that can suggest design strategies [Mid94b]. Obviously, some well-understood parts of design flows, especially the lower levels (e.g. logic synthesis), for which good, generally applicable optimization algorithms exist, or in cases where efficiency is not the primary concern, the selection and application of transformations can and will be automated to further reduce design time. For these situations we suggest using existing optimization and cost estimation techniques as developed for silicon compilers on top of our behavior-preserving transformations, to maintain correctness by construction. The current version of TRADES is still fully interactive.

For TRADES the above considerations resulted in a combination of formally verified, designer-selected, pre-proven behavior-preserving transformations and a CDFG as design representation. This approach is known as *transformational design* and illustrated in Figure 3.

4.3. Existing Solutions

Existing silicon compilers (e.g. Cathedral Compilers [DeM86], Piramid [WBM90], and the commercial version DSP Station by Mentor Graphics, Phideo [LMW91], Hyper(-LP) [CPM95]) are specialized towards specific application and target domains. While this specialization was initially introduced to allow construction of efficient automatic synthesis tools, consisting of design steps, cost estimators, and optimization algorithms, it is now more and more becoming a limitation. Partitioning restricts the design space that can be explored to the common subset of the application domain and the parameterized target architecture. This is clearly in conflict with the need to support the complete life cycle. It also limits the often quoted usefulness of silicon compilers to quickly explore the design space since the design space is both restricted and fragmented.

Efficiency of the results of these automatic tools is for many applications still insufficient. While this is improving it must come very close to good manual designs to be acceptable for mass production applications. This is often still not the case for anything other than benchmark examples. In [Pig89] it is reported that due to the use of design automation the transistor density normalized to minimum feature size has decreased significantly over the years, indicating less efficient use of silicon. Experience from the high-volume embedded controller market shows that efficiencies within 20% of manual designs are needed to successfully introduce automatic compilation techniques. Such levels of quality can not yet be achieved consistently using automatic synthesis tools. However, for first generation implementations where efficiency is less critical silicon compilers have proven to be valuable tools.

Although the use of formal methods is increasing, design correctness is currently not very well dealt with. For many systems it mainly depends on simulation and the assumption that automating the synthesis process will reduce the number of errors compared to manual designs. This is probably true, but at the same time the complexity of designs increases as does the cost of going to silicon. Furthermore, due to the black-box nature of silicon compilation systems, tracing an error in the realization back to its origin can be rather difficult. We also feel that formal methods do not naturally appeal to chip designers [MuH95]. To overcome this the formal aspect of a design methodology should be presented to the designer in a convenient and easy to understand manner. The use of behavior-preserving transformations that have been proven correct by the tool designer provides this.

It is also clear that in order to successfully introduce a new design methodology, integration with existing tools is essential. For the design of TRADES integration with the PHIDEO silicon compiler from Philips Research was required. In the design example in Section 6 TRADES is used to design a very efficient processing unit for use by PHIDEO which is responsible for scheduling and allocation, and the synthesis of memory and interconnections.

5. Design Representation and Transformations

The requirements on a representation format that can be used as *intermediate* language between different specification languages and synthesis systems and as synthesis *backbone* in transformational design will be investigated. Issues like expressive power, formal semantics, implementation suggestion, and ability to manipulate the format will be addressed. Furthermore, the transformations required for full-scale transformational design and related work on transformational design are discussed.

5.1. Design Representation

One of the most important requirements in transformational design is a representation format with compositional, formal semantics to allow proving the correctness of transformations. This is unlike VHDL where the semantics are informally defined by the simulation model. CDFG and the similar signal flow graph (SFG) design representation

formats have shown to be very useful in high-level synthesis [CaT89] and allow for formalization of their semantics [HuK94]. An overview of data flow languages and their semantics can be found in [LeP95]. The best known CDFG variants in high-level synthesis are those derived from the applicative language Silage [Hil85]. They are, for instance, used in the Cathedral [DeM86], Piramid [WBM90], and as DFL in the commercial DSP Station tools. The proven applicability of CDFG languages to digital design and the need for compositional formal semantics made us choose the CDFG-like language SIL [Klo92, Klo93, KMN92], which integrates both control and data flow into a single graph and has formal semantics [HuK94]. SIL has been developed cooperatively by Philips Research, IMEC, and the University of Twente as part of the EC-funded ESPRIT/SPRITE project. It includes support for hierarchy, structured data types, recursion, multi-rate, ordering, and control structures.

The integration of control and data flow allows the same transformations to be used in both domains. Furthermore, it eases the transition of functionality between data and control flow as occurs when, for instance, the amount of hardware sharing is changed. Examples of this can be found in [Mid95] where we show how filter implementations with different levels of hardware sharing, optimized for area and power, can be derived using transformations. The amount of control flow ranges from none to extensive control flow in a processor/program-based implementation. In this last implementation both processor architecture and program are constructed from the filter algorithm by means of these generalized design steps.

The use of SIL as a language backbone for transformational design requires not only possession of the usual *abstraction* and *composition* properties, but in addition both syntax as well as semantics should be easy to *manipulate*. In our experience this requires a one-to-one mapping between syntactic and semantic components. The use of hierarchy as an abstraction mechanism, in contrast to its use as an information hiding mechanism, requires the treatment of hierarchically defined operations in the same way as primitive operations. This includes the ability to transform them without flattening them first. We define a hierarchical operation, |a–b|, which calculates the absolute difference of two signals. The `Abs_Diff` graph defining this operation is shown in Sidebar I. It should be possible to apply the same commutativity transformation to an operation that is defined by the `Abs_Diff` graph as to one defined by the primitive `add` operation.

One of the key objectives during the design of SIL was to make it suitable as an *intermediate design representation* between functional, applicative, and imperative languages used for specification (e.g. Miranda, Silage, VHDL, C) and high-level synthesis tools. This includes translation of the implementation suggestion hidden in specifications, such as the ordering and overwriting mechanism in imperative languages. In [HHK92] it was demonstrated how a SIL graph with an imperative implementation suggestion can be transformed into a functional-style graph. The next section will discuss our view on the implementation suggestion. Section 5.3 focuses on constructs of SIL, which are required for preserving the implementation suggestion and are different from most other CDFG languages.

5.2. Implementation Suggestion

While others consider the implementation suggestion an unwanted side-effect of the specification process requiring a normalization step to dispose [Jan94, CGR93, Gaj94], we do not. On the contrary, the implementation suggestion may very well be intended by the designer and contain valuable 'pointers' in the design space towards the optimal solution. This is especially so if the specification is written by an experienced designer. Furthermore, preserving the original implementation suggestion makes the design process more transparent to the designer, which is essential in a user-centered methodology. In [Sam95] support for our view of the implementation suggestion can be found.

```
Sum(a,b,c:bit[0…k];d:bit[0…l])y:bit[0…m]=
begin
        y = ((a+b)+c)+d;
end
```

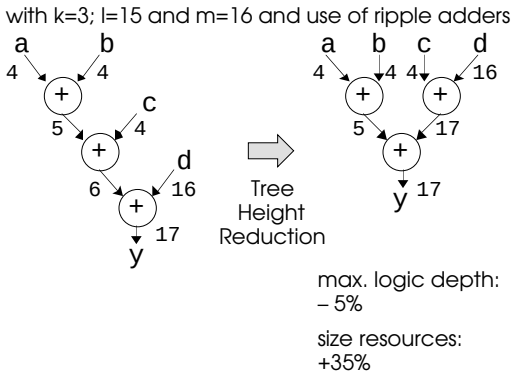

Figure 4. Implementation suggestion

Since specification and implementation should by definition exhibit equivalent behavior (a refinement in behavior could be allowed, see also Section 7.2), adding or changing the implementation suggestion of a specification is the sole purpose of synthesis. Therefore, any approach not capable of representing such an implementation suggestion in an unambiguous way is not suitable as backbone for synthesis. On the other hand, being able to change or remove part of the implementation suggestion is essential. Techniques proposed in [CGR93 and Jan94] are useful for this latter aspect but neither is capable of obtaining a unique, normalized representation for each behavior. Due to the (intentional) ambiguity in the representation of the implementation suggestion in [CGR93] we think the format is not suited as backbone for synthesis.

Figure 4 shows an example with parameterized data types where maintaining the implementation suggestion is important. The graph on the left most closely matches the implementation suggestion contained in the specification. Many automatic synthesis tools apply a design step known as tree height reduction, which is based on the associativity property of addition, to reduce the depth of the logic in adder chains, as shown on the right. This step potentially reduces the critical path, power consumption, and increases the amount of parallelism [Cha92]. However, because of the unbalanced data types in this instantiation of the parameterized design, tree height reduction has the opposite effect and the critical path, in fact, increases due to the larger circuit size. Apparently the designer had additional knowledge about the relative sizes of the data types when the specification was written, which was not explicit from the behavior but was contained in the implementation suggestion.

5.3. SIL Syntax and Informal Semantics

In this section, the relevant aspects of SIL will be discussed. Figure 5 shows a simple recursive *SIL graph* `FAC` which calculates the factorial of natural numbers.

A SIL description consists of one or more graphs of which one is identified as the top level, system graph. A SIL

graph consists of *nodes*, *data flow edges*, and *sequence edges*. Nodes denote operations. They 'fire' a result after they have received all necessary input values. These values are communicated along the data flow edges according to a *single token flow* model— i.e., there can be no accumulation of tokens on edges. Sequence edges, represented by dotted arrows, define additional constraints on the firing order besides those imposed by the data causalities. They are required for modeling the overwriting mechanism of imperative languages and can be used to indicate scheduling constraints. In a SIL graph *input* and *output* nodes serve as communication *ports* to the outside world. Besides the *primitive* nodes whose definition is part of the language, such as the '+' node, a SIL graph like the FAC graph can be used as definition for (recursive) *hierarchical* nodes. Recursion is used to model loops. As mentioned in the previous subsection, SIL is a control data flow graph language which integrates control and data flow in a single graph. The small open (or solid) bullets on the side of some nodes are *condition ports*. A true (false) value on an open (solid) bullet indicates execution of the node. A false (true) value denies execution and in our example recursion is terminated. A denied node in that case fires an *empty token*, which we will discuss later. This distributed control model has some significant advantages over the common 'switch/branch' and 'merge' nodes. The control flow is localized which is required when using local transformations and avoids some of the problems normally associated with crossing boundaries of basic blocks. Furthermore, the single token flow execution model is preserved.

One important aspect in which SIL is different from the single-assignment language Silage, is language support for modeling the overwriting of variables in imperative languages. This construct is called a *join*. A join can be found on the input (or *sink*) port of the output node where two data-flow edges come together. The semantics of this construct is that the last arriving non-empty token is accepted. Note that in Figure 5 a sequence edge is used to impose this firing order. If only empty tokens arrive, an arbitrary value from the type is chosen. To determine which token arrived last the inputs to the join are statically ordered. In our example this is done by means of a sequence edge. This construct eases the translation of imperative languages to SIL and helps preserve the implementation suggestion.

The factorial example demonstrated the basic features of SIL. The timing and unfolding model will be explained by means of a simple accumulator circuit shown in Figure 6a.

Figure 6a shows the SIL representation of a simple circuit that sums all incoming data. In CDFG and SFG representations each edge represents an infinite *stream* of tokens. This representation is actually a shorthand notation for the infinite dependency graph shown in Figure 6b. Like Silage, the '@' operation maintains state information and represents the connection between different unfoldings. Since synchronous digital hardware and many signal processing algorithm operate on an infinite number of samples the SIL syntax implicitly models this infinite loop, i.e. Figure 6a is a shorthand notation for 6b. Figure 6a can

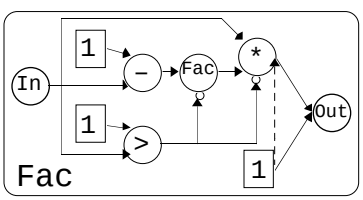

Figure 5. Recursive SIL graph for factorial calculation

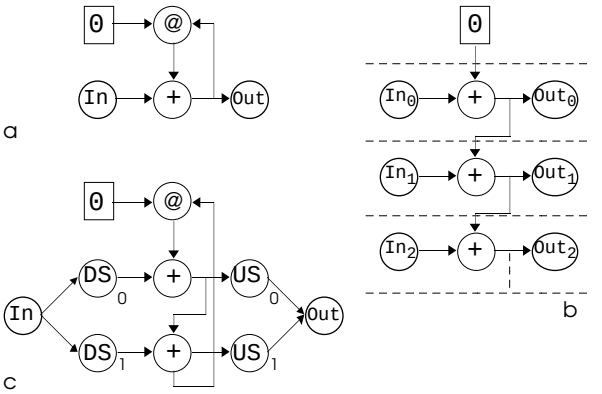

Figure 6. Accumulator circuit a. Folded SIL representation b. Unfolded data dependency graph c. Partially folded SIL graph

be obtained by folding the dependency graph from Figure 6b. This process is very similar to the folding (projection) techniques used in array processor synthesis where dependency graphs are folded onto a regular array of processing elements [Kun88].

In Figure 6c a partially folded, multi-rate SIL description of the circuit is shown. The upsample (US) and downsample (DS) nodes function as cyclo-static de/multiplexers. The numbers indicate the active phase. During this phase the US nodes consume a token from their input and fire it at their output. An empty token is produced during the other phases. The step from Figure 6a to c can be made by means of simple transformations. Therefore, both exhibit exactly the same behavior. We will now roughly estimate the power consumption of both circuits. The area of Figure 6c is about twice that of Figure 6a. The critical path however, assuming the use of ripple adders, is in first order equal (bit delay + word width · carry propagation delay, versus 2 · bit delay + word width · carry propagation delay). To maintain the same throughput the operating frequency of the adders can be halved. This allows a reduction of the supply voltage from 5 to 3 Volts and a power reduction of a factor 2.5 in a CMOS implementation. In [Mid95] we have shown how space-time mappings can be used for both power and area optimizations.

Some important aspects of SIL that we have skipped are the typing and parameter mechanisms. All ports in SIL are typed. SIL supports the use of bit, bit-vector, abstract integer, and structured types (arrays, records, etc.). The parameter mechanism of SIL can be used to describe and transform designs which for instance use parameterized types (Figure 4). This was used but not shown for the `Abs_Diff` design in Sidebar I. A complete definition of syntax and informal semantics of SIL-1 and -2 [Klo92, Klo93] can be found on our Web site:

<http://wwwspa.cs.utwente.nl/aid/aid.html>

5.4. Transformations

Our design flow is based on the consecutive application of pre-proven behavior-preserving transformations. The use of *pre-proven generalized* transformations in a compositional design description eliminates the need for (post-) verification or simulation steps. Verification has been performed during the implementation of the design tool freeing the chip designer from this effort and minimizing run-time overhead. A transformation is said to be correct if the set of behaviors allowed by the implementation is a subset (i.e. a refinement) of the behaviors permitted by the specification. In Section 7.2 a formalization of the notion of refinement is given.

The definition of a transformation consists of *preconditions* and *actions*. Preconditions define if a transformation may be applied to a selected part of the graph without compromising correctness. The designer has to decide where and when a transformation is applied; preconditions do not contain a notion of optimality. For the example distributivity transformation shown in Figure 2 the preconditions state how the '+' and the '*' should be interconnected and how the data types of the inputs and output of the '+' node should relate to avoid overflow. Theorem 7.3 in Section 7.4 shows the preconditions of a transformation. The actions state which elements should be removed from or added to the original graph to execute the transformation.

Transformations are defined to be primitive, small, and local. *Primitive* implies that redundancy in the set of transformations is minimized, i.e. a transformation cannot be further decomposed into smaller behavior-preserving transformations without moving to a lower level of abstraction. *Smallness* is required to ensure that it is reasonable to verify the correctness of both the definition and implementation of the transformation. Achieving correct implementations can be further aided by reverse mapping techniques where both the forward transformation and its inverse are implemented independently [Józ95]. *Locality* is needed to exploit the compositionality of behavior. It also guarantees that execution of transformations is nearly instantaneous and independent of the size of the design.

At first sight it may seem that some transformations, such as partitioning, are global, but this is not the case. While the decision about *where* to partition is a global problem the partitioning itself can be done by many small local steps. The same is true for scheduling. In [Sam95] it is claimed that loop transformations cannot be tackled with this approach, but as we have shown in [Mid96] loop transformations discussed in [Sam95] can in fact be decomposed into smaller transformation steps.

In high-level synthesis we recognize three tasks: optimization of the algorithm, refinement to a lower level of abstraction closer to hardware, and dealing with the time-space trade-off, i.e. defining the relation between operations in the algorithm and (hardware) operators. These tasks are reflected in the three categories of transformations we identify: *optimization*, *refinement*, and *assignment* transformations.

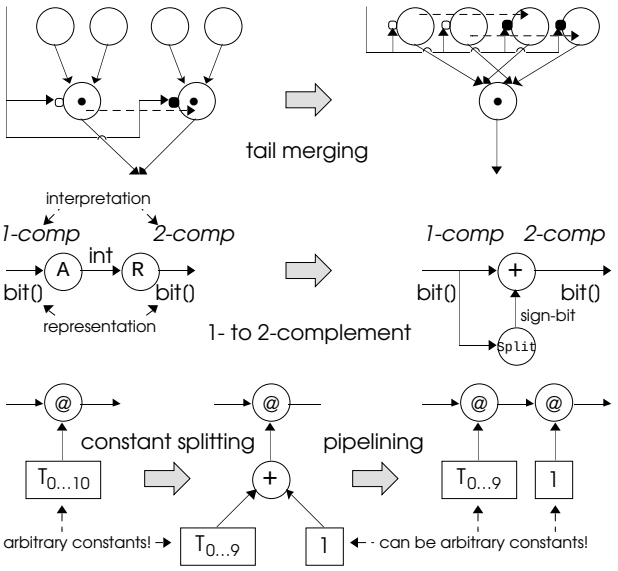

Figure 7. Optimization, refinement and assignment transformation

Examples of the first category, of which many are inspired by software compiler optimizations, are algebraic transformations, like the one shown in Figure 2, and constant propagation, both of which change the structure of the algorithm. Examples of refinement transformations are strength reduction, decomposition (of for instance a constant multiplier into a shift operation), and transformations between different data types. Good methods for dealing with data typing are very important both from a correctness and efficiency point of view. Achieving bit-true behavior between the original design and the transformed one is non-trivial [Mid94a]. The same is true for the calculation of minimal bit-widths to optimize area and timing. The structure of the algorithm in time and space can be altered by means of retiming, multi-rate, loop, and scheduling transformations. Traditionally most emphasis in high-level synthesis has been on scheduling and allocation steps [Gaj94, WaC95]. An overview of transformations used in TRADES can be found in [EMH93].

Figure 7 shows examples from the three categories. The tail merging transformation is used to merge identical operations in two mutually exclusive data paths, like the ones generated in the translation of `if-then-else` constructs. The transformation is used in the design of the `Abs_Diff` function as shown in Sidebar I. In Section 7 the correctness proof of this transformation is discussed. The second transformation from Figure 7 is used for the implementation of the conversion between data types which use the same representation but different interpretation: bit vectors interpreted as 1-complement and those interpreted as 2-complement. This is modeled by means of Abstraction (A) and Representation (R) nodes. The inverse conversion from 2- to 1-complement is somewhat more difficult due to the required extension of the representation. The last transformation is a variant of retiming. To illustrate its use we have already supplied values for the constants. On the left side the designer specified a delay element with a static though as yet undetermined amount of delay between 0 and 10. In SIL this can be specified by means of the special *top* 'T' constant which represents an undetermined value of the type. The first constant-splitting transformation uses some of this *design freedom*— the delay is at least 1 and may be up to 10. This allows the designer to specify that the latency of the circuit may be altered (within bounds), as is often the case in signal processing applications where throughput is the constraint. The second pipelining transformation rewrites the single delay element into the concatenation of two delay elements. These transformations allow the designer to deal with flexible latency while remaining within the bounds of the specification.

In TRADES the designer instantiates a generalized pre-proven transformation from a menu and applies it to a selected part of the design. The design system automatically checks if the preconditions of the transformation are satisfied; if so, the transformation is executed on the selected part, otherwise the designer is informed why.

Sidebar I Design of an Abs_Diff

An important part of the video conversion algorithm described in Section 6 is the `Abs_Diff` block discussed here. It calculates the absolute difference between two signals and is used to estimate the direction of edges. The block is reused four times in the original algorithm. Because the four different instantiations of the design require different data types, the design of the `Abs_Diff` function is parameterized in the data types. This introduces some additional problems which will not be discussed in this paper. We use transformations to derive the same solution obtained previously at Philips Research using manual design.

Figure I1.a shows the specification of the `Abs_Diff` block in Pascal-like pseudocode. It will be optimized for area. The use of integer data types is assumed but not shown in the figures. The straightforward translation of the pseudocode into SIL is shown in Figure I1b. The open and solid bullets on the left sides of nodes are conditions and are used to model conditional execution. A true (false) value on an open (solid) bullet indicates that execution is enabled. The dotted edge is a *sequence edge* and models an extra constraint on the firing order of the nodes.

To reduce the area required for implementing the design the *number* of and the *cost* per operation must be minimized. The former can be achieved by reducing redundancy and increasing the amount of hardware sharing. Operation cost can be minimized by exploiting special cases for which efficient implementations exist. Both objectives can be achieved by first performing a refinement step of operations and data types. Sharing and redundancy reduction are further aided by applying transformations based on, for instance, commutativity and the algebraic identity relation to regularize the graph. Figure I1c shows the result of these steps and temporarily moving the conditions to assign nodes which in combination with the join model a multiplexer. The design after redundancy is removed using common subexpression elimination on the multiply and add nodes is illustrated in Figure I1d.

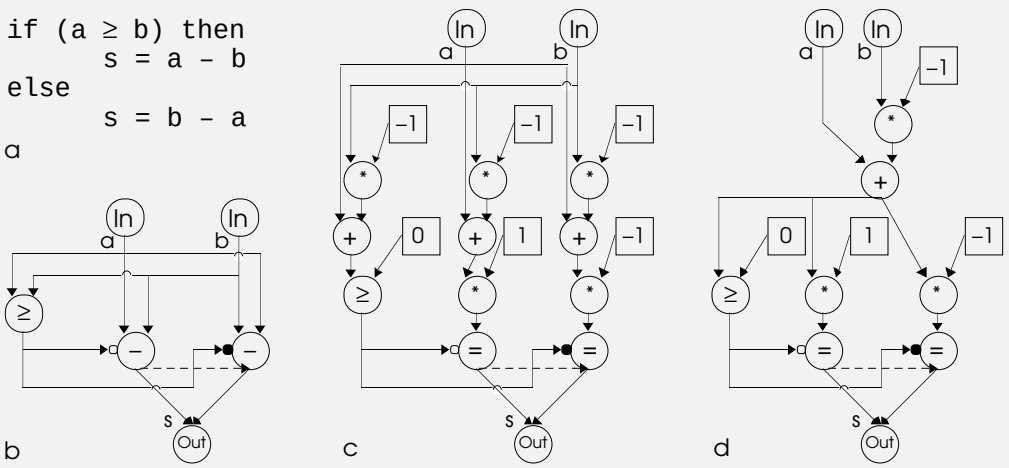

Figure I.1. Abs_Diff example

Figure I.2.e shows further refinement utilizing type transformations, from the integer to the 2-complement bit-vector domain, and exploitation of special cases. A valid intermediate step could be first typing the · –1 sections of the graph in 1-complement where this operation can be realized using a simple vector-wide bit inversion. Tail merging, which is formalized in Section 7, is used for merging mutually exclusive operation in Figure I2f. The final result after further tail merging, and clustering of the primitive operations in such a way that the behavior of the clusters corresponds to that of the building blocks in a hardware library, is shown in Figure 2g.

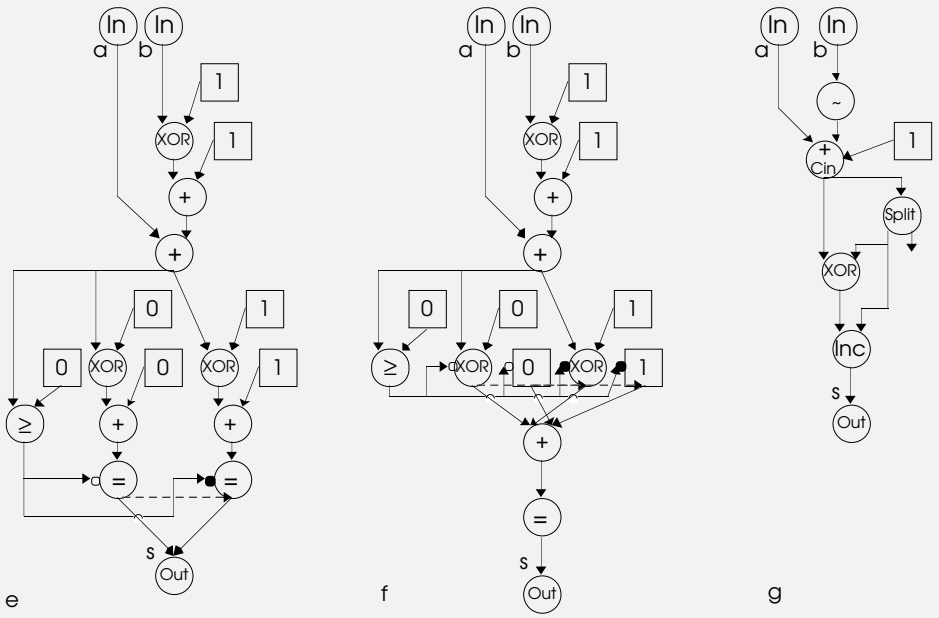

Figure I.2. Abs_Diff continued

As illustrated in Figures I1 and I2, a combination of data path optimizations (e.g. algebraic transformations and common subexpression elimination), refinements (e.g. strength reduction and type transformations) and control transformations (e.g. condition moving) is needed for the complete design. The final result is much smaller and somewhat slower than the original. Notice that the initial transformation steps actually increase the size of the design— this behavior seems to be typical. Specifications often represent a local optimum since designers tend to optimize the obvious inefficiencies and use high-level generalized operators. Our experience indicates that a very large set of transformations is needed to deal with real-world designs. Even the small `Abs_Diff` design required more than 25 different kind of transformations. Other systems do not offer such a wide spectrum of transformations and are often limited to data path optimization and some refinements. In the final implementation of the video processing algorithm the solution shown in Figure I2g is not used. We found that still better optimizations are possible if we take the context of the `Abs_Diff` block into account. This requires a partial flattening of the design. While area optimization was the primary objective in the `Abs_Diff` example, the methodology can also be applied to minimize power or increase speed which typically requires the use of time-space transformations [Mid94a, Mid94b, MMM95, Mid95].

The set of transformations required for full-scale transformational design is very large even though we have attempted to minimize this set by using *generalized*, *primitive* transformations. For instance, if we would not use generalized transformations, over 20 different variants of associativity using different data types and operations would have to be defined. Even for the transformational design of the small `Abs_Diff` design, which calculates the absolute value of the difference of two signals, shown in Sidebar I, well over 25 different kinds of transformation are needed (not counting type transformations). In [Jan94] it is suggested that selecting a set of transformations for transformational design is "relatively simple"; our experience however indicates the opposite. The set of primitive transformations needed in full-scale transformational design necessary for achieving correctness by construction, is very large and directly related to the expressive power of the design representation. Preferably we would like to define a *complete* set of primitive design transformations. A set of primitive transformations is said to be complete if for all pairs of behaviorally equivalent designs in the design space, there exists a valid sequence of primitive transformations between them. In general there will be more than one set of transformations that satisfies this property. In [Vem90] a method for proving the completeness of a set of 18 RTL design transformations for the Single Architectural Register Transfer model is presented. Unfortunately this representation model is very simple compared to SIL and uses only structural transformations. Furthermore, the proof assumes the existence of a normal form of a behavior and the use of invertible transformations. As far as we know such a normal form is not known and attempts to define one [Jan94, CGR93] have failed. Because transformations on SIL can reduce design freedom (for example the constant-splitting transformation shown in Figure 7) they are generally not invertible.

Because of the large number of transformations, the efficiency of the process of implementing and verifying transformations requires special attention. Manual verification of all transformations required in the complete system is not feasible and a mechanized approach is needed. In Section 7 we propose a novel method for the verification of transformations on CDFGs.

5.5. Related Work

The use of transformations for the design of both software and hardware is not new. While formal program refinement techniques never really caught on, the use of transformations in software compilers is common [Lov77, ASU86]. There are, however, several aspects that make hardware design different from software design and the use of formal techniques more attractive. In hardware design, rebuilding hardware normally has a very high cost and is time-consuming. Furthermore the efficiency and real-time requirements are also very stringent compared to most software.

Many (partially) transformation-based design systems exist or are under construction as part of larger projects. The HYPER system developed at the University of Berkeley [Bro92, Cha92, IPS93, Pot94] provides a set of transformations for automatic algorithmic-level design optimization. HYPER consists of transformations from the optimization, refinement, time-space assignment categories. The refinement transformations are however limited to strength reduction. Silage is used as specification language and translated to the HYPER CDFG [RaH90]. Repetition is modeled by means of iteration nodes. Conditional execution is emulated through multiplexer-like constructs. Within HYPER many automatic optimization scripts have been developed successfully for different purposes, including optimizing resource utilization, critical path and power reduction, and to improve testability. The transformations are not verified and designs are not bit-true with respect to the specification [Mid94a]. Simulation is required to evaluate both correctness and the numerical behavior of the transformed design. The methodology is therefore not correct by construction. The system does not support mixed-level designs or refinement (of data types) nor does it support the construction of architectures. TRADES and HYPER are largely complementary. The automatic optimization algorithms developed for HYPER could be applied within TRADES while HYPER could benefit from our more powerful CDFG language, support for VHDL, and much larger set of pre-proven design transformations which guarantees correctness by construction.

The CAMAD [Pen89, Pen94, HaP95] high-level synthesis system uses automatic transformations for scheduling and allocation. Algorithms written in Pascal or Behavioral VHDL used for specification are translated into an Extended Timed Petri Net representation in which data and control flow are separated. In comparison with TRADES this system is automatic and only supports a very small part of the design path from high-level specification to efficient implementation. For hardware/software codesign the ETPN representation is partitioned in a hardware and software part [StP94]. The hardware part is synthesized using CAMAD while C compilers are used for software synthesis. However, for many applications (e.g. DSP based systems) where codesign techniques are useful the quality of code produced by C-compilers is not nearly sufficient. Furthermore, for many DSPs such as those used in Section 5, no C compiler is available. While their approach is interesting for prototyping we feel hardware/software codesign of commercial products will require many more transformation steps and could be better tackled with TRADES.

For automatic optimization of sparsely-multiplexed data paths at the algorithmic level the GATE tool [Jan94] has been developed at IMEC, Belgium. The tool uses optimizing, strength reduction, and retiming transformations. The optimization strategy is based on transforming DFG-based specifications of data paths into a pseudo-normalized

expanded form. Normalization makes the optimization more or less independent from variations in the specification. In the next step area is minimized by transforming using a 'greedy algorithm'. GATE is one of the few systems that discusses problems related to the assignment of data types in a behavior-preserving way. Compared to TRADES this approach provides a much smaller set of transformations (no refinement for instance), uses a simple architectural model, does not support control flow and purposely removes the implementation suggestion.

For the optimization of specifications which use multi-dimensional signals, such as those from the video and image processing domains, SynGuide [Sam93, Sam95] has been developed at IMEC, Belgium. The tool provides (affine) transformations on loops with manifest (i.e. data-independent) bounds. Transformations are performed on Silage and can be used in both an interactive and automatic mode using the memory optimization tool MASAI [FNS94, SFC94]. A geometry-based verification model has been developed for automatic post-verification of the design after applying loop transformations. SynGuide is very application-specific compared to TRADES and used post-verification, but it shares our philosophy on the implementation suggestion and interactive design.

The Olympus tool set developed at Stanford University [MKM91] is oriented to high- and low-level synthesis. The tool achieves some target architecture independence by using bottom-up synthesis to calculate design costs instead of using architecture dependent cost estimators. HardwareC is used as specification language at the algorithmic level. As internal format at the behavioral/algorithmic level the Sequencing Intermediate Format (SIF) is used. The Hercules tool for behavioral synthesis provides both interactive, such as hierarchy expansion, and automatic transformations like loop unrolling, constant propagation, common subexpression, and dead code elimination. The tool only provides a very small number of transformations and uses simulation to determine correctness.

A complete set of transformations is defined for a class of RT-level designs called Single Architectural Register Transfer designs [Vem90]. This work is limited to structural transformations (basically scheduling and allocation) which can be used as basis for an interactive design exploration tool. While the architectural model is very simple the work is interesting because it addresses the completeness problem of a set of design transformations as discussed in Section 5.4.

As part of the Format project [TLG93] behavior-preserving transformations are used to synthesize behavioral specifications of communication-dominated hardware into structural VHDL. Specifications are defined by means of timing diagrams which are translated to T-LOTOS. An interactive bottom-up transformational approach is used. Each transformation adds a module to the current partial (initially empty) implementation. A transformation is correct if there exists a remainder that when combined with the new partial implementation implements the specification. This approach is completely different from TRADES.

The System Architect's Workbench [TDW88, WaT89] uses the Value Trace CDFG as design representation. For specification, the Instruction Set Processor Specification (ISPS) language is used. Transformations supported include hierarchy introduction, expansion, constant propagation, loop unrolling, and common subexpression elimination. Interactively used transformations on the control flow are discussed in [WaT89]. In [MCF93] correctness of transformations in the System Architect's Workbench is investigated. Several errors were found, often related to incorrect use of bit array data types.

The Yorktown Silicon Compiler-based [BCD88] system [Cam89] emphasizes transformations to improve scheduling and allocation. The computations and control structure are not changed. The Yorktown Internal Form, which is similar to CDFG languages, is used as intermediate language. The transformation steps used for scheduling and allocation are proven correct. Correctness by construction is based on transitivity of behavioral equivalence but no use is made of compositionality.

Most systems incorrectly ignore the influence of data typing on both correctness at the bit-level [Mid94a] and efficiency (Section 5.2, Figure 4). Very few can guarantee correctness because the transformations are not formally verified. That formal verification of transformations is not a luxury has been shown in [Raj95ab] and [MCF93]. Furthermore most of these systems employ an automated approach to the application and selection of transformations. To facilitate this they limit the application domain to for instance data path [Jan94] or memory optimization [SFC94].

Although such approaches have proven to be useful in many cases (examples include memory reduction, optimizing resource utilization, critical path and power reduction, and to improve testability, and scheduling, allocation), we found that they are too limited for full-scale transformational design.

6. Design of an IPS Converter

In this section the interactive design of both a real-time multi video-signal-processor based prototype as well as a full-custom implementation of a real-time video processing algorithm are discussed. As a test case for our transformational design methodology we have selected the edge direction detection section of interlaced-to-progressive scan conversion (IPSC) algorithms. Figure 8 shows a diagram of the complete IPSC processor.

IPSC algorithms are used to double the screen retrace frequency by interpolating intermediate scan lines of a field of an interlaced frame. If an edge is present in a field, interpolation takes place in the direction of the edge. Detection of edges is based on the gradient in the luminance. Three gradients are calculated, based on the difference in luminance of three pairs of opposing pixels in the line before and after the estimated line. The smallest gradient indicates the direction of the edge. In combination with linear interpolation of the luminance signal this is known as the *edge-based line average* (ELA) algorithm. A diagram of the direction detector is shown in Figure 9. A more detailed description can be found in [Mid94b, MMM95]. Our direction detector was used in combination with an extended form of the ELA algorithm which first feeds the input signal

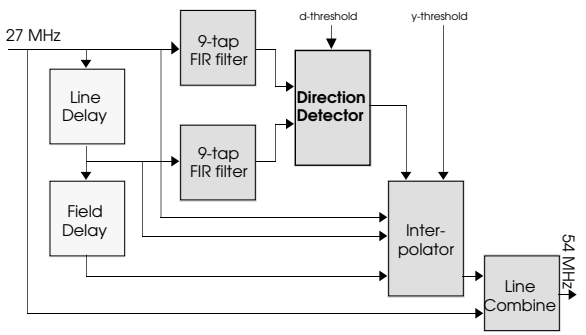

Figure 8. Architecture of a line interlaced-to-progressive scan conversion processor

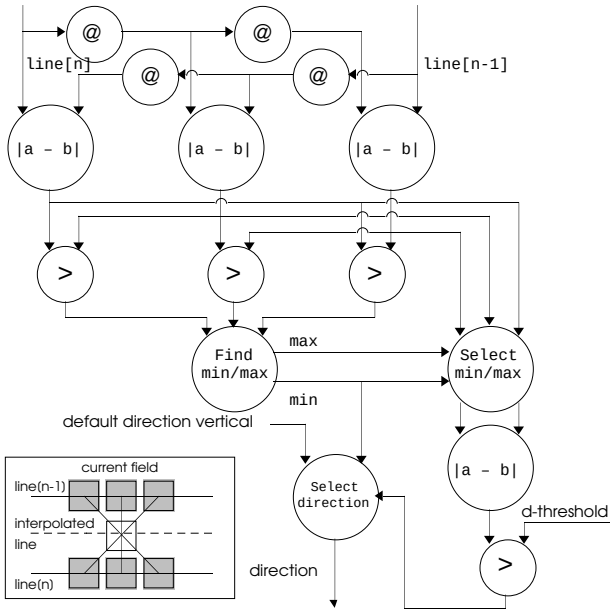

Figure 9. Flow graph of a direction detector for IPSC algorithms

through two low-pass filters and uses median interpolation instead of linear interpolation (Figure 8). This last modification requires the luminance of the pixel from the previous field with the same spatial location as the pixel to be estimated. Recent extensions to the filter include the use of motion compensation for determining the correct pixel from the previous field. Further extensions have been proposed in [LKL94], which also contains a more complete overview of the algorithm and its history.

We have selected the direction detector because it is relevant in industry and currently used in television systems and could also prove useful for IPSC in the new HDTV standard [BCD95]. In a previous custom implementation the direction detector accounted for 40% of the size of the data path while the field and line memory consumed the most chip area. The example has been studied extensively both at Philips Research [LMW91] and IMEC [SaR92] as part of the ESPRIT/SPRITE project and both a prototype and custom implementations have been developed. This allows us to compare our results with respect to efficiency and flexibility with those obtained previously using different methodologies and tools.

Using our methodology we interactively designed two implementations for the direction detection algorithm, starting from a single high-level VHDL specification. A prototype software implementation was designed for realization on a multi video-signal-processor based prototyping system. This system has been developed by Philips [RSD90] for the evaluation of video processing algorithms in real time. In addition an efficient, low-cost full-custom hardware implementation was designed. For the latter the PHIDEO silicon compiler was used to synthesize the memory and interconnection structures as shown in Figure 8. This approach has the advantage that changes in the memory access profile, due to modifications in the latency of the direction detector, can be easily and automatically accounted for. The two designs are typical for the kind of implementations that are developed during the life cycle of a product. Figure 10 illustrates both design flows. Each box except for the VHDL-to-SIL translator represents the application of a number of transformations. The designer interacts with TRADES by selecting a transformation from a menu. By selecting a part of the graph the designer indicates where TRADES must apply the transformation when the preconditions of the transformation are satisfied. Estimating the cost of design alternatives is not part of TRADES and must be assessed by the designer or estimated by external tools.

Our prototype was compared to one previously implemented at Philips Research. To evaluate our synthesis results with respect to flexibility and efficiency we derived two alternative custom implementations starting from VHDL. First, as a reference to the current state of commercial synthesis tools, a direct implementation of the high-level VHDL description of the edge detection algorithm was synthesized with Viewlogic Systems' ViewSynthesis (VS) version 2. Next, an implementation was derived using VS starting from a lower-level, manually optimized structural VHDL specification developed at Philips Research which is illustrated in Figure 9. At IMEC this same structural specification, although specified in ELLA, was further optimized for area and speed using synthesis tools from the academic community. The results obtained at IMEC are scaled towards the VS system by using the original Philips implementation as a reference. We refer to these latter two respectively as the 'structural' and the 'optimized structural' specifications. Results are compared to those obtained using TRADES and VS as a back end for low-level synthesis. Figure 11 shows the different paths. We used extensive simulation to verify the correctness of our results. Both implementations obtained using TRADES were indeed correct by construction even though TRADES is not yet fully implemented and verified.

6.1 VHDL to SIL translation

Different approaches to the translation of VHDL to SIL have been investigated. Within both Philips and the University of Twente experimental tools for the semantics-based translation of a subset of VHDL to SIL have been implemented. Such a semantics-based approach has, however, some significant drawbacks. The translation process is complicated because it requires the interpretation of the meaning of a VHDL description. Furthermore it is limited to a subset of VHDL, requiring complex synthesis guidelines to define the synthesizable subset which again introduces compatibility problems. Within TRADES we are therefore developing an alternative syntax-based approach to the translation of full VHDL (with the exception of the explicit time constructs AFTER and FOR) [MMH95, MMM96]. It requires only a

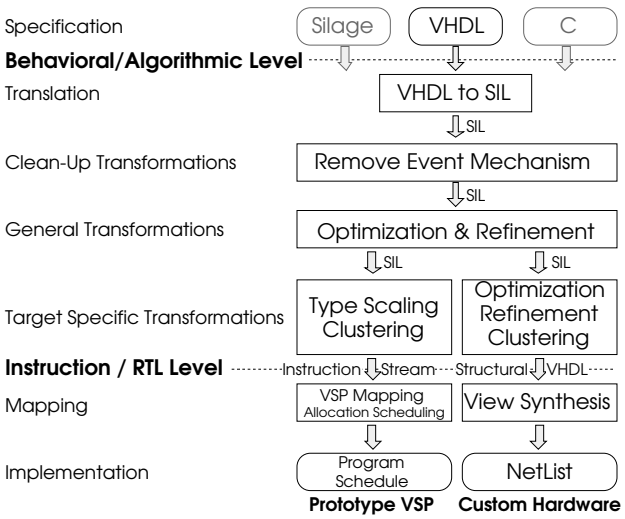

Figure 10. Design flows for prototyping and custom hardware design

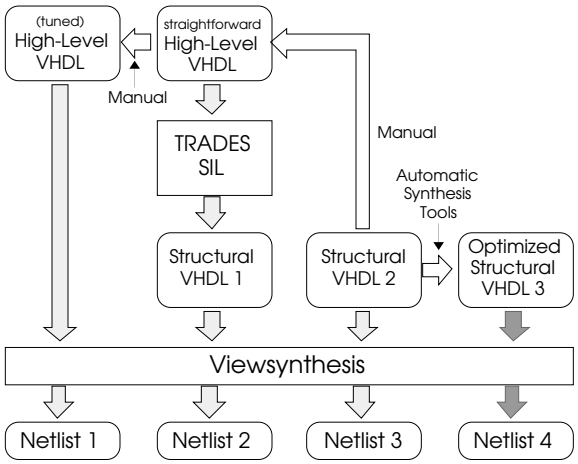

Figure 11. Experiment: four alternative design flows from VHDL to silicon

relatively straightforward and transparent statement-by-statement translation which preserves the implementation suggestion contained in the specification.

After translation a SIL description results which is basically a VHDL simulator modeled in SIL. Although this represents a valid way to implement the desired behavior in hardware, it is not very efficient. Therefore, the resulting SIL description is 'cleaned up' and optimized using the same behavior-preserving transformations as are used in the rest of the design flow. The designer decides which transformations are applied where and thereby which part of the original implementation suggestion is preserved. An important advantage is that the complexity of the translation process is now moved to the SIL domain, where correctness can be guaranteed and transformations reused.

As a starting point the most straightforward high-level VHDL specification of the direction detection algorithm was used. This specification was simple and about a fifth of the size of the structural description that resulted from the manual design. A first attempt to synthesize this description directly using VS failed because it violated the synthesis guidelines for VS. After some tuning of the original specification the VS tool accepted the specification. Note that every manual rewrite of the specification requires a validation step of the specification by means of simulation. Our VHDL-to-SIL compiler, however, accepted the initial high-level specification without any problem and generated useful SIL which was cleaned up interactively using transformations. This demonstrates the flexibility of the syntax-based translation method. The low-level structural VHDL description used by Philips proved to be no problem for VS. Details on the efficiency of the four alternative paths are discussed in Section 6.5.

For the mapping onto the VSP system we compared our results with an implementation of the algorithm obtained previously at Philips. Results are discussed in Section 6.4

6.2. 'Clean-Up' Transformations

An obvious disadvantage of the syntax-based method is the large size of the SIL graphs that result after the translation. This is somewhat compensated for by the fact that the straightforwardness and transparency of the translation process preserves the structure of the original VHDL specification, which is especially important in a user-centered methodology.

Due to the syntax-based translation, the VHDL event mechanism appears in SIL and manifests itself as two nested loop levels, modeled in SIL by means of recursion. The inner loops model the continued execution of processes. The outer loop triggers computation if internal events occur (the delta mechanism) [MMH95, MMM96]. Loop and other transformations are used to remove this event mechanism. How this is done largely depends on the structure of the wait-statement in the original VHDL specification. For the IPS example the clean-up step was performed interactively. Currently we are working on automating the recurring parts of the clean-up process. 'Clean-up' transformations reduced the number of operations in the SIL description to 30% of the original. The final graph after cleaning up looks very similar to the one in Figure 9.

6.3. General Optimizations

Our primary design objective for the direction detector was the minimization of area. From Figure 9 it will be clear that the top three comparators which can be implemented as subtractors can be combined with the bottommost absolute difference function. The steps used are very similar to those needed in the design of the `Abs_Diff` function as shown in Sidebar I. Further optimizations are possible by integrating the top three absolute difference functions with the comparators below and not calculating the *absolute* difference in all cases. This step requires a partial flattening of the design. As a side effect the size of the select min/max function can also be halved because only the correct max–min pair need be selected.

In general, area is reduced by minimizing the number of operations and implementation cost per operation. The former can be reduced by increasing the amount of sharing or reducing redundancy. If we decompose operations into more primitive operations the chances for sharing and redundancy reduction increase. This was demonstrated in Sidebar I. As it turns out we can use tail merging and common subexpression elimination in combination with many small transformations (like those based on the algebraic identity relation) to effectively halve the required implementation cost.

6.4 Prototype: VSP Implementation

To map the SIL graph onto a processor architecture the behavior of operations in the graph should correspond to the instructions of the processor. This requires the clustering of SIL operations (i.e. the introduction of hierarchy) to make the behavior of the clusters correspond with that of processor instructions. Furthermore, type transformations are required to scale the data types of all operations to the (fixed) word width of the processor.

The mapping of the transformed direction detection algorithm onto the VSP-2 shows a reduction in operations from 15 to 13. The main reason for this small gain is that execution of control operations is relatively expensive on a processor. These can only be performed on word-wide signals and are just as expensive as operations in the data path, whereas the custom implementation can use efficient single bit-wide signals. Within the small algorithm this control aspect becomes dominant.

For the scheduling and allocation (i.e. the mapping in time and space) of the instruction stream onto the processors, commercial tools specifically designed for the VSP-1 and VSP-2 were used. It would, however, also be possible to perform these steps within TRADES.

When comparing our prototype implementation to the original prototype we found a small bug in the original related to an incorrect comparison operation which had not yet been discovered by simulation. This illustrates the usefulness of our approach for obtaining correct implementations.

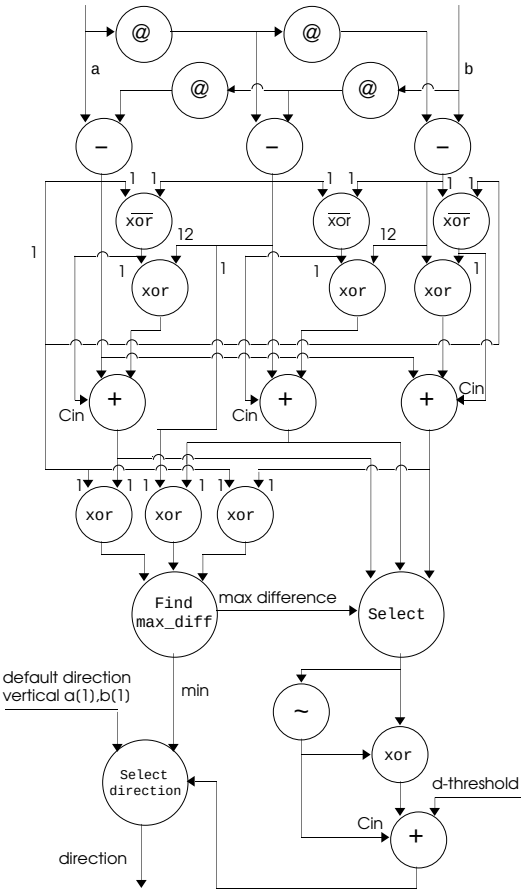

Figure 12. Flow graph of the optimized direction detector

The VSP-based prototyping system can now be used to evaluate the algorithm in real time. In comparison, our compiled code-based SIL simulator running on a fast HP workstation required 30 seconds per field.

6.5 Custom Implementation

The most important transformations are a combination of strength reduction and type transformations which allow the mapping of abstract operations onto lower-level operations which, after clustering, can be mapped efficiently onto hardware structures. Figure 12 shows the final graph in which clustering (i.e. the introduction of hierarchy) is used, for instance, to construct the adders with carry-in input from two chained SIL adders.

First, the results with respect to area in terms of gate count will be investigated. Table I lists the gate counts resulting from the four different design flows shown in Figure 11. The third row shows the maximum reduction in area obtained at IMEC. Their improvements resulted from using transformations based on *stuck-at* detection (i.e. dead code elimination and constant propagation) and applying them to the original structural specification (second row). Other transformations that were tried, such as logic optimization using SIS, actually increased the size of the circuit, probably because they interfered with regularity of the data path. Their best results have been translated to gate counts obtained by VS using the structural implementation as a reference. The large size of the direct implementation using VS clearly indicates the limitations of the commercial tool. More interesting is understanding the large difference between the implementation of the manually designed structural specification and the TRADES results and the relatively small gain obtained in the implementation of the optimized structural specification. The optimizations used for the optimized structural implementation were too low-level to result in a large area gain. They missed the reduction in redundancy possible at the algorithmic level and the optimizations possible by partial flattening and later reclustering in a different way. For instance the `Abs_Diff` function had to be flattened and optimized across the boundary of the hierarchy because the optimizations shown in the inset were not globally optimal. Our approach, using TRADES, of first optimizing the algorithm at the algorithmic level and the possibility to deal with mixed RT- and gate-level descriptions is much more effective in this example.

TABLE I
Gate count after different design paths

Design	# of gates	% gates
VS	2613	100 %
Structural	1237	47 %
Optimized Structural	1076 *	41 % *
TRADES	685	26 %

* results are translated and scaled relative to structural implementation

To what extent should these improvements be attributed to the design methodology or to the designer? There is no doubt that the designers of the structural and optimized structural implementation were more experienced than we are. We think there are a couple of effects in play. Although we can not quantify it, our belief is that the transparency of the design process and use of small design steps significantly improved understanding and hence quality of the design. Because the designer is stimulated to think about seemingly obvious design steps he understands the design better and as a consequence is able to see more efficient alternatives. Secondly there is a learning effect, as already mentioned in Section 3. By looking at previous implementations we learned quite a few optimizations. Because our approach was completely interactive we were able to exploit this. Furthermore, the ability to mix different levels of abstraction allowed us to control the implementation in sufficient detail to guarantee efficient mapping to hardware by the VS synthesis tool, without getting lost in unnecessary details.

Another important advantage of our methodology is that it allows the algorithm to be specified as it is most natural to the designer and least error-prone. Optimizing the algorithm results in a more complex description (just compare the difference between the intermediate design in Figure 9 and the final in Figure 12). The latter would be difficult to notate correctly without the use of behavior-preserving transformations because the design is less intuitive and certainly less comprehensible than the high-level VHDL specification.

The implementation derived from the structural specification required 13 pipeline stages to reach the 27 MHz throughput requirement. [SaR92] claims that the optimized structural specification resulted in a speed gain of 25%. We noted however that it did not seem very likely that transformations based on stuck-at detection (i.e. constant signal detection) could result in any speed gain other than that obtained because of fan-out reduction. It seemed more likely that the reported gain was due to the inability of the timing tools to calculate data-dependent critical paths. That this is indeed the case has been confirmed by the authors of [SaR92]. Different optimizations used at IMEC resulted in a speed increase of up to 41% at the expense of area. Delay estimation using a carry-ripple model indicates a speed gain close to 50% for the implementation obtained using TRADES.

If we look at power consumption the results obtained with TRADES seem even more favorable because the improvements in area and speed accumulate. The capacitance component in the power equation is half that of the im-

plementation obtained from the structural specification. In addition the increase in speed of up to 50% results in fewer pipeline stages than the structural implementation, again reducing the capacitance component. Alternatively we could also maintain the same number of stages but reduce the supply voltage. Of course this would require the other sections of the design to be speeded up equally, which seems feasible. This results in a power reduction between 50% and 80%. On the other hand it is expected that stuck-at based optimizations will result in a power reduction which is relatively less than the reduction in area. Also, power reductions as a result of the originally reported speed gains are unlikely.

Although measuring the actual design time is very difficult when working with prototype tools we observed that it was very small compared to the time required for gathering all the necessary information on the design. Our experience indicates that elimination of debug cycles does indeed seem very effective in reducing design time. To quantify these savings a comparative study of a large number of designs should be part of future work.

This design exercise of an industrial example demonstrated the feasibility of transformational design. We designed for multiple target architectures very efficient first-time-right implementations, while exploiting existing tools where possible to further reduce design time.

7. Formal Specification and Verification of Transformations

We have seen in Sidebar I how a sequence of transformations is used to optimize and refine specifications at the algorithmic level (also known as behavior level) into implementations at the register transfer and gate level. The transformations are correct if the set of behaviors allowed by the implementation is a subset of the behaviors permitted by the specification. Trivial implementations that allow an empty sequence of behaviors can be ruled out by showing either that at least one behavior is allowed by the implementation, or that the implementation is equivalent to its specification with respect to behavior. A transformation transforms one graph structure into another by removing or adding nodes and edges. An informal representation would lead to subtle errors, making it difficult to guarantee the correctness of the transformations. In this section we undertake to provide guarantees for correctness of transformations on CDFGs, independent of the underlying behavior model[2]. Our verification technique is very powerful in that the correctness verification of a transformation holds irrespective of the size and structure of the graph on which a transformation is applicable. Furthermore, we relax the constraint that two ports connected by a data-flow edge in CDFGs are behaviorally equivalent. We generalize that a data-flow edge can exist between source port and sink port, where the source port is a subtype of the sink port— i.e. we allow behavioral refinement between the source and the sink. Such a generalization is useful if we need to connect a port which allows a subrange of bit vectors to a port that can allow arbitrarily-sized bit vectors.

We use a property-oriented approach to address the correctness problem. In this approach, a small set of basic properties corresponding to SIL graphs, called *axioms*, are asserted. The truth of other properties is checked by applying a small number of inference rules on known true properties. Such derived properties are called *theorems*. We use the Prototype Verification System[3] (PVS) from SRI International to mechanize the verification scheme. The PVS specification language allows us to specify the properties using a convenient level of abstraction. The PVS verifier features automatic procedures and interactive inference rules to check properties of specifications. The inference rules are based on higher-order logic [SOR93]. The rest of the section is organized as follows. We discuss related work in Section 7.1. We provide a formal characterization for SIL, and a verification scheme for transformations in Section 7.2. A brief account of the mechanization in PVS, and results from verifying various transformations are tabled in Section 7.5.

7.1. Related Work

There have been some efforts in analysis and verification of refinement transformations in the past. However, none of the past work has dealt with formal verification of the correctness of transformations on CDFG graphs in general. Most of the efforts have concentrated on specialized hardware description languages tied to specific behavior models. In comparison, our work does not depend on any specific model of behavior. This makes it applicable to a variety of CDFG formalisms used in different high-level synthesis frameworks. Furthermore, we can handle CDFGs of arbitrary structure and size for verification, unlike previously proposed techniques.

A formal model was proposed for verifying correctness of high-level transformations by McFarland and Parker [McP83]. A formal system using transformations for hardware synthesis has been discussed by Fourman [Fou90]. A synthesis system for a language based on an algebraic formalism has been presented by Jones and Sheeran [She90], and its formalization has been presented by Rossen [Ros90]. Another algebraic approach to transformational design of hardware has been worked out by Johnson [Joh84]. Correctness of register-transfer level transformations for scheduling and allocation has been dealt with in [Vem90]. A formal analysis of transformations used in Systems Architect's Workbench (SAW) high-level synthesis was studied by McFarland [McF93]. Transformations used in YIF (Yorktown Internal Form) [BCD88] have been proved to be behavior-preserving [Cam89]. In this work, a strong notion of behavior equivalence based on an imperative semantics tied to a particular model of representation is used. A post facto verification method for comparing logic-level designs against a restricted class of data-flow graphs in SILAGE was presented by Aelten and others [AAD93, Ael94]. A formalization of SILAGE transformations in HOL was studied by Angelo [Ang94].

7.2. Formal Characterization of SIL

In this section we first introduce a relation that describes refinement on ports. The refinement relation is specified without giving a concrete model of the behavior of ports. This is achieved by providing the basic properties that the relation has to satisfy under any model of behavior. We then specify the structural properties that the ports need to satisfy for behavioral refinement. These properties are then used to specify refinement of CDFGs. The fundamental properties are asserted as axioms, while other properties are derived as theorems by application of inference rules on other theorems or axioms. The verification scheme for correctness of transformations with an illustration of its application to the

2 By 'behavior model' we mean the model governing the type and history of data values that the graph may assume.

3 <http://www.csl.sri.com/sri-csl-fm.html>

cross-jumping tail-merging transformation is discussed in Section 7.4.

We introduce an abstract refinement relation `silimp` as a relation on ports[4] of the nodes. Thus, if `p1` and `p2` are sets of ports, `silimp(p1,p2)` means that `p1` is a refinement of `p2`. This relation could be interpreted to mean that the set of values allowed by `p1` is a subset of the values allowed by `p2`. It should be noted, however, that we do not constrain the type or history of data values the ports could assume at any time. The refinement relation `silimp` is asserted to have the properties of *reflexivity* and *transitivity*. Such basic properties have to be satisfied under any behavior model of control data-flow graphs. The equivalence of SIL graphs `sileq` is defined by introducing the *symmetry* property in the refinement relation `silimp`.

A data-flow edge connecting two ports modifies the behavior of the sink in accordance with other data-flow edges connecting the same edge output. The behavior of such a sink, called a join, is determined by an ordering of the data-flow edges as discussed in Section 5.1. We model this ordering by associating weights with the data-flow edges. A function `w` on ports would return a weight, which could be a number:

Definition 7.1 (Weight)

```
w: [port,port -> weight]
```

This means that we need to compare the weights on the data-flow edges that form a join. The weights on data-flow edges that do not form a join need not be compared. However, the definition of SIL specifies that no two data-flow edges communicate tokens simultaneously into a join, and no two weights on the edges forming a join can be equal. This suggests that we need a reflexive, transitive, and antisymmetric ordering relation on weights: such a relation is called *partial order*. We define a partial ordering relation[5] < on weights, and assert that the weights are ordered if and only if the associated data-flow edges form a join. A data-flow edge between port `p0` and `p1` is indicated by the relation `dfe(p0,p1)`. The axiom is specified as follows:

Axiom 7.1 (Partial order on weights)

```
FORALL (p0,p1,p2:port):
NOT (p0 = p1)
IMPLIES
   dfe(p0,p2) AND dfe(p1,p2)
              IFF
  (w(p0,p2) < w(p1,p2) OR
   w(p1,p2) < w(p0,p2))
```

We describe the property that the behavior of a *join* depends on the ordering of the data-flow edges, by comparing weights on the edges flowing into the join port. We state the property that the join port is in a refinement relationship with the source whose associated data-flow edge has the maximum weight:

Axiom 7.2 (Behavior of join)

```
FORALL (p1,p2:port):
 (FORALL (p:port):
   w(p,p2) < w(p1,p2)) IMPLIES
 silimp(p1,p2)
```

We can derive the behavior due to a data-flow edge whose sink is not the output of any other data-flow edge. We will call such an edge an exclusive data-flow edge (`xdfe`). The behavior due to such an edge is derived as the following theorem:

Theorem 7.1 (Behavior in the absence of join)

```
FORALL (n1:graph),(p:port):
xdfe(outport(n1),p)
IMPLIES
silimp(outport(n1),p)
```

We emphasize here that source port connected to a sink port by a data-flow edge has a behavior which is a subset of the behavior of the sink port. This means that the set of values that the source can assume is a subset of the set of values that the sink can assume. Such a refinement relationship, which is more general than a strict enforcement of behavioral equivalence, allows a data-flow edge between a subtype port (such as a fixed range of bit vectors) to its supertype port (arbitrarily-sized bit vectors).

We can now associate the refinement and equivalence relation with a complete graph by using the properties expressed above. A graph `refines` or implements another graph when the data relation of the implementing graph is contained in the data relation of the specification graph. This property is expressed by the final axiom as follows:

Axiom 7.3 (Graph refinement)

```
FORALL (n0,n1:graph):
 refines(n0,n1) AND
       silimp(inports(n0),inports(n1))
 IMPLIES
 silimp(outport(n0),outport(n1))
```

7.3. Compositionality

The axiom of refinement 7.3 allows us to provide a compositional proof of correctness of transformations: i.e., the refinement of component subgraphs of a graph G ensures the refinement of G. This allows us to assert that if a transformation is applied locally to a subgraph `g` of a graph G, leaving the rest of the graph unchanged, then the graph G undergoes a global refinement transformation. Thus, we can state the following theorem:

Theorem 7.2 (Compositionality of Refinement)

Let $g_1, g_2,\ldots,g_N$ be the component subgraphs of `G` and $g'_1, g'_2,\ldots,g'_N$ the subgraphs of `G'`. If every g'_i is a refinement of g_i, then `G'` is a refinement of `G`. i.e., $(\forall\ i$: refines$(g'_i, g_i)) \Rightarrow$ refines(G', G)

The proof follows by induction on graph structure. A detailed proof is given in a doctoral thesis [Raj95b].

7.4 .Verification of Transformations

The general method we employ to specify and verify transformations consists of the following steps:

1. Specify the structure of a SIL graph, i.e. the transformation domain, on which the transformation is to be applied. The structure may have arbitrarily-sized ports or an arbitrary number of ports.
2. Specify the structure of the SIL graph, i.e. the transformation domain, expected after the transformation is applied. The structure may have arbitrarily-sized ports or an arbitrary number of ports.
3. In the case of verifying refinement, we impose the constraint that the corresponding inputs of the SIL graphs before and after transformation are `silimp`— that is, the set of input values to the SIL graph after transformation is a subset of the set of input values to

---

4 We could also allow the `silimp` relation to hold among sets of ports. The size of the sets of ports could be arbitrary.

5 We do not employ the usual notation `<=` to stress that no two weights on different edges forming a join can be equal.

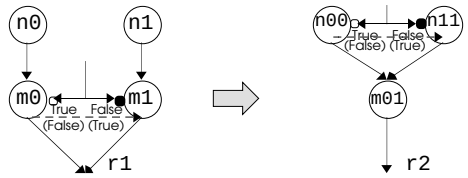

Figure 13. Cross jumping tail merging

the SIL graph before the transformation. For behavioral equivalence, the constraint is imposed as `sileq`: the sets of input values to both graphs are identical.

4. Verify the property that the outputs of the SIL graph before transformation are `silimp`— that is, the outputs of the SIL graph after transformation are refinements of corresponding outputs of the SIL graph before transformation. In the case of behavior-preserving transformations, the corresponding outputs are verified to be `sileq`.

We illustrate this scheme in the cross-jumping tail-merging transformation from Figures 7 and 13 and used in the `Abs_Diff` example of Sidebar I. In this transformation, two conditional nodes of the same kind whose output ports connect to the same sink are checked for being mutually exclusive— that is, that the conditions on both of the conditional ports are not true (or false) at the same time (when exactly one of them is true at any time). In such a case, the two nodes can be merged into one unconditional node of the same kind, and the conditions moved to the nodes of the subgraph connecting it.

In the course of our verification we found a mistake in the informal specification of the transformation. The conditions had incorrectly been placed on the nodes *n0* and *n1* instead of *m0* and *m1*. Furthermore, we could relax the mutual exclusiveness constraint, which is a weakening of the precondition. We introduce the assumption that the ordering of the data-flow edges coming out of the nodes *m0* and *m1* in the original graph is the same as the ordering of the data-flow edges coming into the node *m01* in the optimized graph. We have suitably modified, generalized, and verified the transformation. The generalized transformation is shown in Figure 14. The transformation is stated as a theorem as follows[6]:

Theorem 7.3 Cross jumping tail merging

```
FORALL (par0,par1,par00,par11,r1,r2:port),
       (m0,(m1:same_kind(m0,m1)),
       (m01:same_kind(m0,m01)):graph):
       % Initial graph structure: exclusive data
       flow edges at input ports of m0 and m1 -
       joins disallowed at input ports
xdfe(par0,inports(m0)) AND xdfe(par1,inports(m1))
       AND
       % Ordering of Weights of edges at output
       ports of the initial graph structure are
       the same as that at the output ports of
       the final graph structure
(w(outport(m0),r1) < w(outport(m1),r1)
                IFF
w(par00,inports(m01)) < w(par11,inports(m01)))AND
       % port r1 is connected to output ports of
       m0 and m1, and no other port
dfe(outport(m0),r1) AND dfe(outport(m1),r1) AND
(FORALL pp: ((pp /= outport(m0)) OR
       (pp /= outport(m1)))
                IMPLIES
       NOT dfe(pp,r1)) AND
       % Final graph structure: input ports of
       m01 is connected to par00 and par11, and
       no other port.
dfe(par00,inports(m01)) AND
dfe(par11,inports(m01)) AND
(FORALL par: (par /= par00 AND par /= par11)
                IMPLIES
       NOT dfe(par,inports(m01))) AND
       % r2 is connected to output of m01, and no
       other port - i.e. not a join
xdfe(outport(m01),r2) AND
       % We trigger the inputs of final graph
       with a refinement of the values of the
       initial graph
silimp(par00,par0) AND silimp(par11,par1)
                IMPLIES
       % We show that the outputs of the initial
       and final graphs are refinements
                silimp(r2,r1)
```

7.5 Mechanization in PVS

The Prototype Verification System (PVS) [OSR93] is an environment for specifying entities such as hardware/software models and algorithms and verifying properties associated with the entities. An entity is usually specified by asserting a small number of known true general properties. These known properties are then used to derive other desired properties. The process of verification involves checking relationships that are supposed to hold among entities. The checking is done by comparing the specified properties of the entities. For example, one can compare if a register-transfer level implementation of hardware satisfies the properties expressed by its high-level specification. PVS has been used for reasoning in many domains, such as in hardware verification [Cyr93, CRS94, RSS95], protocol verification, and algorithm verification [LOR93].

We have specified and verified transformations such as copy propagation, constant propagation, common subexpression insertion, commutativity, associativity, distributivity, and strength reduction described by Engelen and others [EMH93] in PVS. In general, the proofs of transformations proceed by rewriting, using axioms and proved theorems, and finally simplifying to a set of Boolean expressions containing only relations between ports and port arrays. At this final stage the Boolean simplifier based on Binary Decision Diagrams (BDD) [BRB90, Jan93] integrated in PVS, is used to determine that the conjunction of Boolean expressions is indeed true. We show the run-times of verifying the various transformations in Table II. It should be noted that the cross-jumping tail-merging transformation required the longest run-times and largest number of major inference rules. Each major inference rule might involve multiple primitive proof rules corresponding to typed higher-order logic. Algebraic transformations such as commutativity and self-inverse required a small number of major inference rules, because the underlying arithmetic decision procedures in PVS are used to automatically prove the algebraic transformations without requiring a manual interaction through a major inference rule.

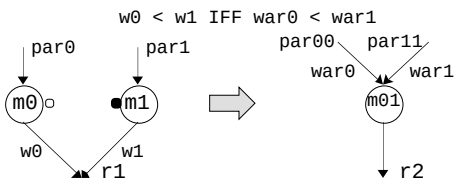

Figure 14. Cross jumping tail merging: generalized and verified. See Theorem 1.3

6 To avoid cluttering the theorem most of the data type information has been omitted.

TABLE II
Run Time in Seconds on a Sparc 2/32MB

Transformation	Run Time in Seconds
Common subexpression elimination	30
Common subexpression insertion	25
Cross jumping tail merging	56
Copy propagation	10
Constant propagation	2
Strength reduction	2
Commutativity	3
Associativity	3
Distributivity	3
Retiming	3
Self-inverse	1

Conclusions and Future Work

We have argued that a design methodology for high-level synthesis which achieves correctness by construction to reduce design time, supports a wide variety of target architectures and different levels of optimization is highly desirable. Existing approaches do not offer this. We have shown how a formal transformation-based design methodology can be used to support the complete life cycle of a system while offering efficient, first-time-right designs. The methodology is based on the application of user-selected pre-proven primitive small local behavior-preserving transformations in a compositional design representation. We identified three categories of transformations which are essential for full-scale transformational design: optimization, refinement, and time-space assignment.

To prove the correctness of transformations a formal characterization of a general control data flow graph specification language has been achieved. We have given a small set of axioms that capture a general notion of refinement and equivalence of such graphs. We have specified and mechanically verified about a dozen of the optimization and refinement transformations. Many transformations have been generalized by weakening the preconditions. We found errors in this process and suggested corrections.

The feasibility of the methodology has been demonstrated on the design of both a prototype and very efficient full custom implementation of a direction detector starting from a single high-level VHDL specification. Furthermore, we have shown the method's ability, through reuse of transformations, to efficiently integrate different design flows as well as its ability to function in a hybrid multi-tool environment. We believe these capabilities, in combination with the interactiveness and transparency of the method, will make transformational design the methodology of choice for high-level synthesis.

Because of the large number of transformations necessary to tackle the design of industrial problems, further integration of our verification scheme with the definition of transformations in TRADES is needed.

Acknowledgments

The authors wish to thank their colleagues at IMEC (Z. Sahraoui and L. Rijnders), Philips Research (G. Essink, W. Kloosterhuis, and T. Kostelijk), SRI International (I. Agi, D. Cyrluk, P. Lincoln, S. Owre, J. Rushby, N. Shankar, and M. Srivas), and the University of Twente (B. Helthuis, J. Hofstede, C. Huijs, Th. Krol, G. Mekenkamp, B. Molenkamp, and J. Posthuma), B. Hekster, and the anonymous reviewers for their valuable comments on this paper, and many others for their contributions. This research was funded by and partially carried out at Philips Research, Eindhoven, the Netherlands.

References

[Ael94] F.V. Aelten et.al, *Event-Based Verification of Synchronous Globally Controlled, Logic Designs Against Signal Flow Graphs*, IEEE Trans. on CAD of ICs., Vol. 13, No. 1 Jan 1994.

[Ang94] C. Angelo, PhD Thesis, IMEC, Leuven, 1994.

[Ast95] P. Asthana, *Jumping the technology S-curve*, IEEE Spectrum, June 1995, pp. 49-54, 1995.

[ASU86] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers, Principles, Techniques, and Tools*, Chapter 10, Addison Wesley, ISBN 0-201-10088-6, 1986.

[BCD88] R.K. Brayton and R. Camposano and G. De Micheli and R.H.J.M. Otten and J.T.J. van Eijndhoven, *The Yorktown Silicon Compiler System*, eds. D. Gajski Addison-Wesley, 1988.

[BCD95] C. Basile, A.P. Cavallerano, M.S. Deiss, et al., *The U.S. HDTV standard: The Grand Alliance*, IEEE Spectrum, Vol. 32, No. 4, pp. 36-45, April 1995.

[BRB90] K.S. Brace, R.L. Rudell and R.E. Bryant, *Efficient Implementation of a BDD Package*, Proc. of the 27th ACM/IEEE DAC, pp. 40-45, June 1990.

[Bro92] R.W. Brodersen, et al, *Anatomy of a Silicon Compiler*, Kluwer, ISBN 0-79239249-3, 1992.

[Cam89] R. Camposano, *Behavior-Preserving Transformations for High-Level Synthesis*, Hardware Specification, Verification and Synthesis: Mathematical Aspects, Cornell MSI Workshop, Lecture Notes in Computer Science 408, pp. 106-128, Springer-Verlag, July 1989.

[CaT89] R. Camposano, R.M. Tabet, *Design representation for the synthesis of behavioral VHDL models*, Proc. 9th Int. Conf. on CHDL, pp. 49-58, eds. J.A. Darringer and F.J. Ramming, Elsevier Science Publishers B.V., North Holland, June 1989.

[CGR93] V. Chaiyakul, D.D. Gajski, L. Ramanchandran, *High-Level Transformations for Minimizing Syntactic Variances*, Proc. of DAC 93, pp. 413-418, 1993.

[Cha92] A. Chandrakasan, M. Potkonjak, J. Rabaey, R. Broderson, *An Approach For Power Minimization Using Transformations*, Proc. of the IEEE VLSI Signal Processing Workshop, pp. 41-50, 1992.

[CPM95] A.P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey and R.W. Broderson, *Optimizing Power Using Transformations*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 14, no. 1, pp. 12-31, January 1995.

[CRS94] D. Cyrluk, S. Rajan, N. Shankar and M. Srivas, *Effective Theorem Proving for Hardware Verification*, Proc. of the 2nd International Conference on Theorem Provers in Circuit Design, Bad Heerenalb (Blackforest), Germany, 26-29 September 1994.

[Cyr93] D. Cyrluk, *Microprocessor Verification in PVS: A methodology and simple example*, SRI-CSL-93-12, Technical Report, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[DeM86] H. De Man, J. Rabaey, P. Six, L. Claesen, *Cathedral-II: A Silicon Compiler for Digital Signal Processing,* IEEE Design & Test, December 1986.

[EMH93] W. Engelen, P.F.A. Middelhoek, C. Huijs, J. Hofstede, Th. Krol, *Applying Software Transformations to SIL*, SPRITE deliverable LS.a.5.2/UT/Y5/M6/1A, June 1993.*

[Eve87] H. Eveking, *Verification, Synthesis and Correctness-Preserving Transformations - Cooperative Approaches to Correct Hardware Design*, HDL Descriptions to Quaranteed Correct Circuit Designs, ed. S. Borrione, Elsevier Science Publishers B.V., North Holland, 1987.

[ELL90] Computer General Electronic Design, *The ELLA Language Reference Manual*, The New Church, Henry St. Bath BA1 1JR, U.K., issue 4.0, 1990.

[FNS94] F. Franssen, L. Nachtergaele, H. Samsom, F. Catthoor, H. De Man, *Control Flow Optimization For Fast System Simulation and Storage Minimization,* Proc. of EDTC 1994, pp. 20-24, Paris, February 1994.

[Fou90] M.P. Fourman, *Formal System Design*, Formal Methods for VLSI Design, ed. J. Staunstrup, North-Holland, IFIP, 1990.

[Gaj94] D.D. Gajski, L. Ramachandran, *Introduction to High-Level Synthesis*, IEEE Design and Test of Computers, Winter 1994.

[HaP95] J. Hallberg, Z. Peng, *Synthesis under Local Timing Constraints in the CAMAD High-Level Synthesis System*, Proc. of IEEE EUROMICRO 95, Como, Italy, 1995.

[Hen95] J. Hennessy, *Hardware/Software Codesign of Processors: Concepts and Example*, Lecture notes NATO/ASI course on hardware software co-design, part I, Tremezzo, Italy, June 1995.

[HHK92] C. Huijs, J. Hofstede, T. Krol, *SIL: a Useful Interface between Specifications and Silicon Compilers,* Proc. of the ProRISC/IEEE Workshop on CSSP, pp. 99-104, Houthalen, April 1992.

[Hil85] P. N. Hilfinger, *Silage: a High-level Language and Silicon Compiler for Digital Signal Processing*, Proceedings of IEEE Custom Integrated Circuits Conference, pp. 213-216, Portland, OR, May 1985.

[HuK94] C. Huijs, Th. Krol, *A Formal Semantic Model to Fit SIL for Transformational Design*, Proc. of 20th Euromicro Conference, pp. 100-107, Liverpool, September 1994.*

[HLD90] F.K. Hanna M. Longley, N. Daeche, *Formal Synthesis of Digital Systems*, Formal VLSI Specification and Synthesis, VLSI Design Methods, I, Editor L.J.M. Claesen, Elsevier, 1990.

[IPS93] Z. Iqbal, M. Potkonjak, S. Dey, A. Parker, *Critical Path Minimization Using Retiming and Algebraic Speed-Up*, Proc. of DAC 93, pp. 573-577, 1993.

[Jan93] G. Janssen, *ROBDD software*, Department of Electrical Engineering, Technical University of Eindhoven, Eindhoven, Netherlands, October 1993.

[Jan94] M. Janssen, F. Catthoor, H. de Man, *A Specification Invariant Technique for Operation Cost Minimisation in Flow-Graphs*, Proc. of the Seventh International Symposium on High-Level Synthesis, pp. 146-151, Niagara-on-the-Lake, Ontario, Canada, 1994.

[Joh84] S. D. Johnson, *Synthesis of Digital Designs from Recursion Equations*, MIT Press, Cambridge, 1984.

[Józ95] L. Józwiak, *An Efficient Verification Method for Application in Transformational Design*, Proc. of Euromicro 95, pp. 118-129, Como, Italy, September 1995.

[Klo92] W.E.H. Kloosterhuis, M.M.R. Eyckmans, J. Hofstede, C. Huijs, Th. Krol, O.P. McArdle, W.J.M. Smits, L.G.L. Svensson, *SIL-2 Language Report*, SPRITE deliverable LS.a.a/Philips/Y3-M12/2, 1993.*

[Klo93] W.E.H. Kloosterhuis, M.M.R. Eyckmans, Th. Krol, O.P. McArdle, W.J.M. Smits, *SIL-2 Language Report*, SPRITE deliverable LS.a.2/Philips/Y3-M12/1, 1993.*

[KMN92] T. Krol, J. van Meerbergen, C. Niessen, W. Smits, J. Huisken, *The Sprite Input Language: An Intermediate Format for High-Level Synthesis*, Proc. of EDAC 92, pp. 186-192, Brussels, March 1992.

[Kun88] S.Y. Kung, *VLSI Array Processors*, Prentice Hall, 1988.

[LeP95] E.A. Lee, T.M. Parks, *Dataflow Process Networks*, Proc. of the IEEE, Vol. 83, No. 5, pp. 773-799, May 1995.

[LKL94] M.H. Lee, J.H. Kim, J.S. Lee, K.K. Ryu, D.I. Song, *A New Algorithm for Interlaced to Progressive Scan Conversion Based on Directional Correlations and Its IC Design*, IEEE Trans. Consumer Electronics, vol. 40, No. 2, pp. 119-125, May 1994.

[LMW91] P.E.R. Lippens, J.L. van Meerbergen, A. van der Werf, W.F.J. Verhaegh, B.T. McSweeney, J.O. Huisken, O.P. McArdle, *PHIDEO: A Silicon Compiler for High Speed Algorithms*, Proc. of EDAC 91, pp. 436-441, Amsterdam, February 1991.

[LOR93] P. Lincoln, S. Owre, J. Rushby, N. Shankar, F. von Henke, Eight Papers on Formal Verification, Technical Report SRI-CSL-93-4, Computer Science Laboratory, SRI International, Menlo Park, CA, May 1993.

[Lov77] D.B. Loveman, *Program Improvement by Source-to-Source transformations*, Journal of the ACM, VOl 24, No. 1, pp. 121-145, January 1977.

[McF93] M.C. McFarland, *Formal Analysis of Correctness of Behavioral Transformations,* Formal Methods in System Design, Vol. 2, No. 3, pp. 231-257, Kluwer, June 1993.

[McP83] M.C. McFarland and A.C. Parker, *An Abstract Model of Behavior for Hardware Descriptions,* IEEE Trans. on Computers C-32(7), pp. 621-36, July 1983.

[Mid94a] P.F.A. Middelhoek, *Transformational Design of Digital Signal Processing Applications*, Proc. of the ProRISC/IEEE Workshop on CSSP, pp. 176-180, Arnhem, Netherlands, March 1994.*

[Mid94b] P.F.A. Middelhoek, *Transformational Design of a Direction Detector for the Progressive Scan Conversion Processor*, CS Memorandum 94-64, University of Twente, Netherlands, October 1994.*

[Mid95] P.F.A. Middelhoek, *Arbitrary Hardware Software Trade-Offs*, Proc. of the 6th IEEE International Workshop on Rapid Systems Prototyping, pp. 19-25, Chapel Hill, NC, June 1995.*

[Mid96] P.F.A. Middelhoek, *Transformations on Loops and Arrays,* CS Memorandum 96, University of Twente, Netherlands, to be published, 1996.

[MKM91] G. De Micheli, D. Ku, F. Mailhot, T. Truong, *The Olympus Synthesis System for Digital Design*, Internal report Center for Integrated Systems, Stanford University, Stanford, CA 94305, 1991.

[MMH95] E. Molenkamp, G.E. Mekenkamp, J. Hofstede, Th. Krol, *SIL: an intermediate for syntax based VHDL synthesis*, Proc. of the VIUF Spring 95, San Diego, CA, April 1995.*

[MMM95] P.F.A. Middelhoek, G.E. Mekenkamp, E. Molenkamp, Th. Krol, *A Transformational Approach to VHDL and CDFG based High-Level Synthesis: a Case Study*, Proc. of the CICC 95, pp. 37-40, Santa Clara, CA, May 1995.*

[MMM96] G.E. Mekenkamp, P.F.A. Middelhoek, E. Molenkamp, J. Hofstede, Th. Krol, *A Syntax based VHDL to CDFG Translation Model for High-Level Synthesis*, Proc. of VHDL International Users Forum (VIUF), Santa Clara, February 28, 1996.*

[MuH95] G. Musgrave, R.B. Hughes, *Review of Verification Techniques*, Lecture notes NATO/ASI course on hardware software co-design, part II, Tremezzo, Italy, June 1995.

[OSR93] S. Owre, N. Shankar and J.M. Rushby, *User Guide for the PVS Specification and Verification System, Language, and Proof Checker (Beta Release)*, Computer Science Laboratory, SRI International, Menlo Park, CA, USA,February, 1993.

[Pen89] Z. Peng, K. Kuchcinski, B. Lyles, *CAMAD: A Unified Data Path/Control Synthesis Environment, Design Methodologies for VLSI and Computer Architecture*, pp. 53-67, ed. D.A. Edwards, 1989.

[Pen94] Z. Peng, K. Kuchcinski, *Automated Transformation of Algorithms into Register-Transfer Level Implementations*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 13, No. 2, pp. 150-166, 1994.

[Pig89] C. Piguet, Design Methodologies and CAD Tools, Proc. of the CICC 89, 1989.

[Pot94] M. Potkonjak, J. Rabaey, *Optimizing Resource Utilization Using Transformations*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, no. 3, March 1994.

[Pre92] R.S. Pressman, *Software Engineering: a Practitioner's Approach*, 3th ed., p. 107, McGraw-Hill, Inc., New York, ISBN 0-07-112779-8, 1992.

[RaH90] J. Rabaey, P. Hoang, *HYPER Flowgraph Policy*, Version 1.2, Internal Document University of Berkeley, March 1990.

[Raj95a] S.P. Rajan, *Correctness of Transformations in High Level Synthesis,* Accepted for the Proc. of the IFIP International Conference on CHDL, Chiba, Japan, August 1995.

[Raj95b] S.P. Rajan, *Transformations on Dependency Graphs: Formal Specification and Efficient Mechanical Verification*, PhD Thesis, Department of Computer Science, University of British Columbia, Vancouver, Canada, Oktober 1995.

[Ros 90] Lars Rossen, *Formal Ruby*, Formal Methods for VLSI Design, J. Staunstrup (ed.), North-Holland, IFIP 1990.

[RSS95] S.P. Rajan, N. Shankar, M.K. Srivas, *An Integration of Model-Checking with Automated Proof Checking*, Lecture Notes in Computer Science, Volume 939, pp. 84-97, editor P. Wolper, Springer-Verlag, Liege, Belgium, 1995.

[Sam93] H. Samsom, L. Claesen, H. De Man, *SynGuide: An Environment for Doing Interactive Correctness Preserving Transformations*, Proc. of VLSI Signal Processing VI, pp. 269-277, Eds. L.D.J. Eggermont, P. Dewilde, E. Deprettere and J. van Meerbergen, IEEE, 1993.

[Sam95] H. Samsom, *Formal Verification and Transformation of Video and Image Specifications*, PhD Thesis, IMEC/University of Leuven, 1995.

[SaR92] Z. Sahraoui, L. Rijnders, *Report on the DIRDET Experiments*, internal report IMEC, 1992.

[SFC94] H. Samsom, F. Franssen, F. Catthoor, H. de Man, *Verification of Loop Transformations for Real Time Signal Processing Applications*, Proc. of VLSI Signal Processing VII, IEEE, 1994.

[She90] G. Jones and M. Sheeran, *Circuit Design in Ruby, Formal Methods for VLSI Design*, J. Staunstrup (ed.), North-Holland, IFIP 1990.

[SOR93] N. Shankar and S. Owre and J.M. Rushby, *The PVS Proof Checker: A Reference Manual (Beta Release)*, Computer Science Laboratory, SRI International, Menlo Park, CA, Februari 1993.

[StP94] E. Stoy, Z. Peng, *A Design Representation for Hardware/Software Co-Synthesis*, Proc. of IEEE Euromicro 1994, pp. 192-199, Liverpool, September 1994.

[TDW88] D. Thomas, E.M. Dirkes, R.A. Walker, J.V. Rajan, J.A. Nestor and R.L. Blackburn, *The System Architect's Workbench*, Proc. of the 25th ACM/IEEE Design Automation Conference, pp. 337-343, 1988.

[TLG93] W.D. Tiedeman, S. Lenk, C. Grobe and W. Grass, *Introducing Structure into Behavioral Descriptions obtained from Timing Diagram Specifications*, Proc. of IEEE Euromicro 93, Barcelona, September 1993.

[Vem90] R.Vemuri, *How to Proof the Completeness of a Set of Register Level Design Transformations*, Proc. of the 27th DAC, pp. 207-212, ACM/IEEE, June 1990.

[VHD88] The Institute of Electrical and Electronics Engineers, *IEEE Standard VHDL Language Reference Manual*, IEEE std. 1076-88, IEEE Press, New York, 1988.

[WaC95] R.A. Walker, A. Chaudhura, *Introduction to the Scheduling Problem*, IEEE Design and Test of Computers, Summer 1995.

[WaT89] R.A. Walker, D.E. Thomas, *Behavioral Transformation for Algorithmic Level IC Design*, IEEE Trans. on CAD, Vol. 8, No.10, October 1989.

[WBM90] R. Woudsma, F. Beenker, J. van Meerbergen, C. Niessen, *Piramid: an Architecture-Driven Silicon Compiler for Complex DSP Applications*, Proc. of IEEE International Symposium on Circuits and Systems, pp. 2696-2700, 1990.

* available at http://wwwspa.cs.utwente.nl/aid/trades/