Abstract

Hardware reusability in high-level synthesis is based on the possibility of mapping several operators to the same hardware module. This possibility may dramatically depend on the ability of the design tool to recognize operators that can be mapped to a single module. This work presents a uniform framework to formally express the semantics of hardware modules, in order to support transformational hardware allocation based on symbolic reasoning.

1. Introduction

Usually, complex circuits have a rich variety of operators and data formats (bit width, numerical representation, ...) that can hardly be handled with the current approaches. Recently, some ad-hoc techniques to improve resource sharing have been suggested [1], but an integral approach is still missing. Nevertheless, hand made designs show that making some minor transformations on basic modules and data formats leads to results with a high degree of hardware reuse. The inclusion of such abilities in synthesis tools would give birth to transformational allocation systems. Transformational allocation is understood here as an allocation process where functionality of hardware modules can be extended or adapted by means of system supported automatic transformations. In addition, the correctness of each transformation should be guaranteed. The main problem is that doing so requires more sophisticated allocation tools, that are able to reason on the meaning of hardware modules and make transformations on them.

The purpose of this work is to create a uniform framework to express the semantics of hardware modules, that can be later used to perform transformational allocation based on symbolic reasoning. Uniformity here means that the framework can be used to express semantic relationships between modules as well as the semantic links between operators and modules. To that, we propose an algebraic approach, which is inspired in the management of abstract data types in the field of theoretical computer science.

2. An algebraic framework.

To express the semantics we propose the following steps:

a. Specification of the design data types. This means the algebraic specification of the data types Bit and BitVector, as well as the algebraic description of the numerical formats implemented as BitVector. For each type we will only give the name of the associated data set, and the names of the operators together with its characterizing properties. These properties are described by equations. The quotient term algebra will define a unique initial semantics, that will support the meaning of all the symbols. This kind of type specifications have been widely used in the literature [2].

b. Module specification. For each module we must give its name, the names of its observable functionalities and a set of equations to construct its algebraic spec. These equations must be such that the specification of modules is a conservative extension of the design data types specification. This means that new elements are not added to the primitive semantics, and that the old elements are not identified.

c. Functionality assignment. This is understood as the association of a concrete binary behaviour to each observable functionality.

d. Declaration of valid transformations. The existing relationships between the functionalities of the modules are declared by means of a set of equations that enrich the model stated in b. This will be later used by the synthesis tool.

Unlike other approaches, this will set a framework for precise reasoning based on symbol handling, since any well formed sentence has a unique meaning (no confusion) and there are no meanings without an associated symbol (no junk).

Example: signatures, equational specs and libraries.

Consider a library only composed of adders and comparators. A specific sort identifier is used for each module family. Associated to each sort identifier, there is one primitive constructor that embodies all the functionalities of a module of the family. The result of each functionality is accessed through the corresponding observer. Equalities will serve to declare component properties and relationships between their functionalities.

Example:

<table>
<thead>
<tr>
<th>Signature</th>
<th>TargetLibrary</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinAdder</td>
<td>BinComp</td>
</tr>
</tbody>
</table>

Operations:

- BinAdderOp : Bit BitVector BitVector -> BinAdder
- BinCompOp : BitBitVector BitVector -> BinComp

Equations:

1066-1409/96 © 1996 IEEE