Constructing AIGs in ABC

A Tutorial

February 28, 2007

Abstract: And-Inverter Graphs are an efficient data structure for manipulating large sequential Boolean networks in a variety of applications dealing with synthesis, technology mapping, and formal verification. This tutorial describes how to construct AIGs in the application code when ABC is used as a static library. To allow for such use of ABC, consult the main webpage for instructions on compiling ABC as a static library.

Construction of AIGs is performed in several steps:

Step 1: Start a new AIG network with the functionality manager, which supports structural hashing:

```
Abc_Ntk_t * pAig;
pAig = Abc_NtkAlloc( ABC_NTK_STRASH, ABC_FUNC_AIG, 1 );
```

Step 2: Name the new network:

```
char * pName; // the name comes from the user's application
pAig->pName = Extra_UtilStrsav( pName );
```

The network name should be a unique identifier given as a zero-terminated C string. In the above code, the network name is copied and stored, so after the above function call the user can dispose of the original name pointed to by variable `pName`.

Step 3: Create primary inputs:

```
int nPrimaryInputs; // this number comes from the user's application
Abc_Obj_t * pObj;
int i;
for ( i = 0; i < nPrimaryInputs; i++ )
     pObj = Abc_NtkCreatePi( pAig );
```

When the AIG is constructed from an available network, it is often convenient to associate the new primary inputs with the corresponding objects in the calling application. So the pointer `pObj` can be passed to the application to create this association.

Now, the primary input number k can be accessed as follows:

```
int k; // given by the user
Abc_Obj_t * pObj;
pObj = Abc_NtkPi( pAig, k );
```

The number of already created primary inputs can be retrieved as follows:

```
int nPIs;
nPIs = Abc_NtkPiNum( pAig );
```

If primary input names are available, they can be assigned using the following procedure:

```
Abc_ObjAssignName( pObj, pName, NULL );
```

The primary input names should be unique identifiers given as zero-terminated C strings.

Step 5: Creating latches involves creating single-input terminals, called "latch input" and "latch output", as follows:

```
Abc_Obj_t * pLatch, * pLatchInput, * pLatchOutput;
pLatch = Abc_NtkCreateLatch( pAig );
pLatchInput = Abc_NtkCreateBi( pAig );
pLatchOutput = Abc_NtkCreateBo( pAig );
Abc_ObjAddFanin( pLatch, pLatchInput );
Abc_ObjAddFanin( pLatchOutput, pLatch );
```

Note that the latch input became a fanin of the latch object, and the latch object itself became a fanin the latch output. After latches are constructed this way, the latch outputs can be used the same way as the primary inputs, and latch inputs can be used the same way as the primary outputs (see Step 6). The user's names can be assigned to all three types of objects: latches, latch inputs and latch outputs.

The name of a latch output should be unique and different from the names of primary inputs. The names of a latch input can be the same as a name of a primary input, latch output, or primary output.

Setting the latch initial value can be done by calling one of the three APIs:

```
Abc_LatchSetInit0( pLatch );
Abc_LatchSetInit1( pLatch );
Abc_LatchSetInitDc( pLatch );
```

In should be noted that ABC currently does not distinguish between different types of sequential elements. The name "latch" is used in ABC for historical reason, to follow the SIS tradition. It may be better called a "technology-independent D-flip-flop".

Step 5: Add internal AIG nodes in a topological order by calling procedures `Abc_AigAnd`, `Abc_AigOr`, `Abc_AigXor`, `Abc_AigMux`, `Abc_ObjNot`, on the already existing AIG nodes. The AIG manager given as the first argument is `pAig->pManFunc`.

Example. Suppose we already created three primary input variables. Now we have to create function F = AB + C. First, we get AIG nodes corresponding to the primary inputs:

```
Abc_Obj_t * pObjA, * pObjB, * pObjC;
pObjA = Abc_NtkPi( pAig, 0 );
pObjB = Abc_NtkPi( pAig, 1 );
pObjC = Abc_NtkPi( pAig, 2 );
```

Next, we add the nodes in a topological order:

```
Abc_Obj_t * pObjAnd, * pObjF;
pObjAnd = Abc_AigAnd( pAig->pManFunc, pObjA, pObjB );
pObjF = Abc_AigOr( pAig->pManFunc, pObjAnd, pObjC );
```

Alternatively, we could have constructed F as follows:

```
pObjF = Abc_AigOr( pAig->pManFunc,
   Abc_AigAnd( pAig->pManFunc, pObjA, pObjB ), pObjC );
```

Currently, the user should not do any referencing or dereferencing of the created AIG nodes.

The resulting pointers to the AIG nodes can be complemented. So, if a data-member of an available AIG node should be accessed, use the regular version of the pointer to the node.

Example. Suppose the AIG node `pObjF` was created, as shown above. To determine the unique integer ID of the node, call `Abc_ObjId( Abc_ObjRegular( pObjF ) );`

Although it is not an error to assign a name to an internal AIG node, there is no need to do so. Since the internal nodes are extensively manipulated during AIG-based synthesis, the internal nodes names are not carried through. If there is a need to reconstruct the internal AIG node names after synthesis, command “dress” often comes handy.

Step 6: When AIG construction is finished, some AIG nodes should be connected to the primary outputs. First, the primary output nodes should be created similarly to how primary inputs are created. For this, procedure `Abc_NtkCreatePo` should be called, instead of `Abc_NtkCreatePi`. The names of the primary outputs can be assigned the same way as the names of the primary inputs. The names of the primary outputs can be assigned the same way as the names of the primary inputs. There should be only one primary output with a given name. However, the name of a primary output can be the same as the name of a primary input, latch input, or latch output.

Step 7: To connect already constructed internal nodes to the primary output nodes, use the following procedure:

```
Abc_Obj_t * pObj;   // internal AIG node constructed before
Abc_Obj_t * pPo;    // the primary output node create before
Abc_ObjAddFanin( pPo, pObj );
```

Note that the internal node `pObj` may be complemented when this procedure is called. The primary output node is never complemented.

Step 8: Due to structural hashing performed while constructing the AIG, some nodes without fanouts (dangling nodes) may result, even if the original logic network did not have dangling nodes. Before any AIG optimization is called using the ABC procedures, the dangling nodes should be removed as follows:

```
Abc_AigCleanup( pAig->pManFunc );
```

If names of the primary inputs/outputs/latches are not given or not assigned when these objects are created, the following procedures can be called to create default names:

```
Abc_NtkAddDummyPiNames( pAig );
Abc_NtkAddDummyPoNames( pAig );
Abc_NtkAddDummyBoxNames( pAig );
```

Step 9: Finally, before returning the constructed AIG and calling optimization or technology mapping in ABC, it should be checked for correctness using an internal sanity-checking procedure:

```
if ( !Abc_NtkCheck( pAig ) )
{
    printf( "The AIG construction has failed.\n" );
    Abc_NtkDelete( pAig );
    return NULL;
}
return pAig;
```

Additional comments:

In should be noted that the primary inputs, latches, and primary outputs can be created in any order as long as the following restrictions are observed:

- create a primary input or latch before creating internal node depending on it
- create a primary output or latch before connecting it to any already created node

This tutorial should work for the latest version of ABC. The data structures are still changing, so the rules and the API names may change in the future versions.

If you have questions or comments, please email **alanmi at eecs dot Berkeley dot edu**.