VIS 코드 분석

2012. 11

Dependable Software Laboratory
건국대학교

This research was supported by the National Research Foundation of Korea (NRF-2015R1D1A1A02062154).

Contact Point:

- Eui-Sub Kim (atang34@naver.com) : MS Student at Konkuk University

내용

그림 목차

1 서론

VIS [1]는 정형검증, 논리합성, 클럭 기반의 시뮬레이션, CTL(Computational Tree Logic) 모델 체킹, 순차적 동시성 검증 (sequential equivalence checking) 기능, 조합적 동치성 검증 (combinational equivalence checking) 기능 등을 제공하는 통합 개발 및 검증 도구이다. VIS는 Verilog를 입력언어로 받아들이며, 내부적으로 vl2mv [2] 라는 변환기를 이용하여 VIS 내부 포맷인 BLIF-MV [3] 파일로 변환한 후 기능을 수행한다. 동치성 검사는 두 프로그램이 동일한 입력에 대해 동일한 출력을 내보내는지 확인하는 정형 기법으로 모든 입력 조합을 이용하여 출력을 확인한다. 내부적으로 회로를 이진 결정 다이어그램(Binary Decision Diagram)을 생성하여 수행을 하며, 이 이진 다이어그램의 크기에 따라 검증 속도 차이 나게 된다.

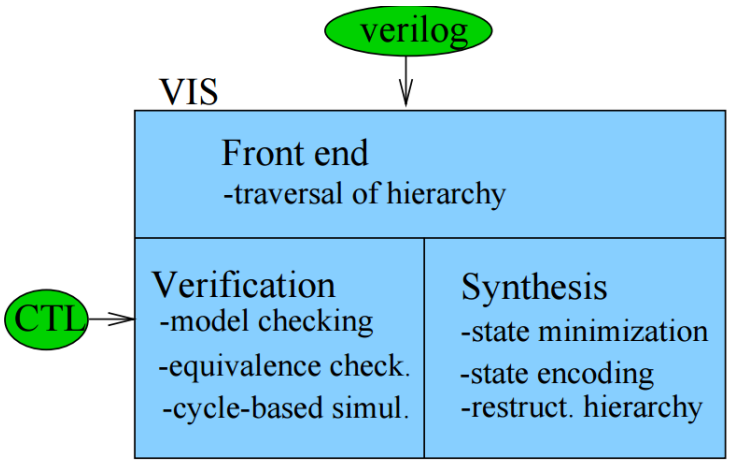

그림 1. VIS의 기능 블록 다이어그램 [4]

그림1은 VIS의 대표 기능을 블록 다이어그램으로 보여주고 있다. VIS는 크게 3파트로 나누어진다. (1) Front end 파트, (2) Verification 파트 (VIS-V), (3) Synthesis 파트 (VIS-S)

Front end 파트에서는 high-level language인 Verilog를 읽고 컴파일한다. 여기서 사용되는 도구는 bl2mv이고 Verilog는 BLIF-MV로 파일로 변환된다.

Verification 파트는 검증을 수행하는 부분이다. 오직 CTL formula만 검증이 가능하며, fairness constraints 는 Buchi 타입으로 정의되어야 한다.

Synthesis 파트는 system을 간략화하는 부분으로써, multi-level circuit의 verilog를 gate level로 변환한다.

2 VIS 코드 분석

VIS는 아래 Directory List로 구성되어 있고, 각각의 Directory의 관계는 아래 그림과 같다

Directory	Synopsis
abs	Incremental CTL model checker
amc	Model Check using over(under)-approximation of the transition relation
baig	Binary AND/INVERTER graph
bmc	bounded model checking (BMC) header file
cmd	Implements command line interface, and miscellaneous commands
ctlp	Routines for parsing, writing and accessing CTL formulas
ctlsp	Routines for parsing, writing and accessing CTL* formulas
eqv	Provides commands for doing combinational and sequential verification on two networks
fsm	Finite state machine abstraction of a network
grab	Abstraction refinement for large scale invariant checking
hrc	Hierarchical representation of a design
imc	Model Check using over(under)-approximation with automatic iterative refinement
img	Methods for performing image computations
io	Routines for reading and writing BLIF-MV files
ltl	Internal declarations
maig	multi-valued AND/INVERTER graph
mark	Data structures used in Markovian analysis
mc	Fair CTL model checker and debugger
mvf	Creation and manipulation of MDD-based multi-valued functions
mvfaig	Creation and manipulation of And Inv-based multi-valued functions
ntk	Flat network of multi-valued combinational nodes and latches
ntm	Construction of MDDs from a flattened network
ntmaig	Construction of mAigs from a flattened network
ord	Routines for ordering MDD variables of a flattened network
part	Partition of a system and creation of MDDs
puresat	Abstraction refinement for large scale invariant checking
res	Combinational verification by residue arithmetic
restr	STG restructuring package
rst	Restructuring package for restructuring the hierarchy
rt	Regression test
sat	Internal data structures of the sat package
sim	Simulation of a flattened network
spfd	SPFD-based wire removal and replacement algorithm for logic optimization of combinational circuits mapped to FPGAs
synth	Symbolic synthesis package
tbl	Routines for manipulating a multi-valued relation representation
truesim	Exported functions and data structures for the truesim package
tst	Test package illustrating VIS conventions
var	Multi-valued variables
vm	"Main" package of VIS ("vm" = VIS main)

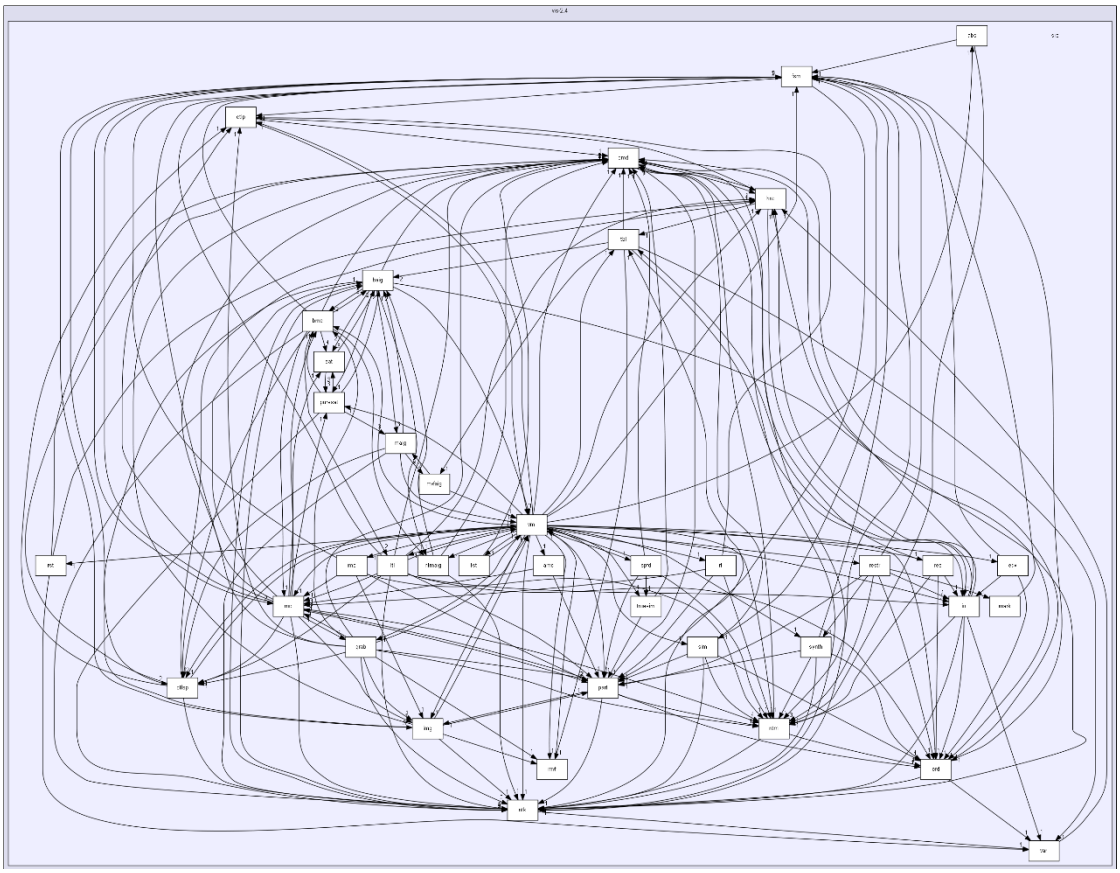

그림 2 VIS directory references

2.1 abs

This package provides the command "incremental_ctl_verification", an algorithm to verify CTL formulas starting from an initial abstraction and applying refinements to increment the level of accuracy in the verification.

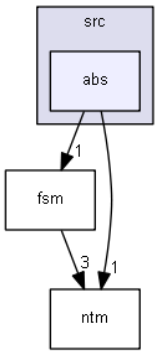

그림 3 ABS directory reference

file	Synopsis
abs.h	Incremental CTL model checker
absCatalog.c	Functions to handle a catalog of sub-formulas to detect common sub-expressions
absCmd.c	Encapsulation for the incremental_ctl_verification command
absEvaluate.c	Evaluation procedures for the abstraction based mu-calculus model checker
absInt.h	Internal declarations required for the incremental CTL model checker
absInternal.c	Miscelaneous functions to handle caches, don't care conditions, initialization and deallocation of structures, etc
absRefine.c	Abstraction Refinement procedures for the mu-calculus model checker
absTranslate.c	Functions to translate CTL formulas to mu-calculus graphs
absUtil.c	Some miscelaneous functions for non-critical tasks such as printing information, sanity checks, etc

2.2 amc

The package implements the approximate_model_check command. The command is designed as a wrapper around the model_check command. The command contains dual algorithms. Predefined abstraction of the transition relation is performed in a controlled way to ensure that the result is reliable. By default, the command makes its effort to prove whether the given ACTL formula is positive. User must set an environment variable "amc_prove_false" when he/she wishes to prove whether the given ACTL formula is negative. When the formula is proven FALSE, then the error trace is returned as in the model_check command. Currently, one predefined approximation method, namely a "block-tearing" method, is implemented. The package manipulate the internal data structure of FSM to obtain over(under)-approximation of a given system. The algorithm begin with coarse approximation in which the degree of initial approximation is set by the user. Initial degree of approximation can be set by using "amc_sizeof_group" environment. When the initial attempt using the coarse approximation fails, the algorithm automatically combine subsystems(take synchronous product) to obtain refined approximations. The procedure is repeated until the approximation is good enough so that the result is reliable.

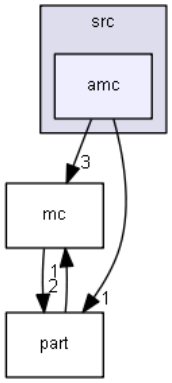

그림 4 amc directory reference

	Synopsis
amc.h	Model Check using over(under)-approximation of the transition relation
amcAmc.c	Model check formula with approxmiations
amcBlock.c	Set best system
amcCmd.c	Ends the amc package
amcInt.h	Internal data type definitions and macros to handle the structures of the amc package

2.3 baig

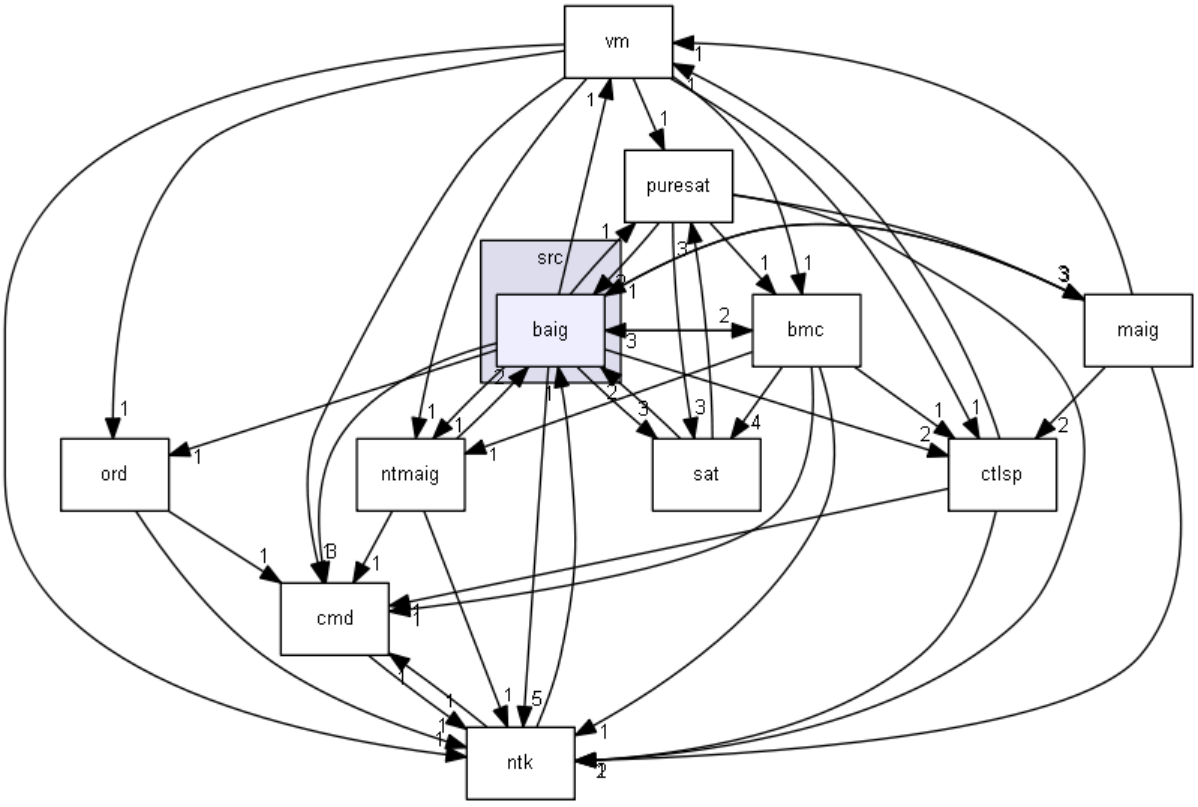

그림 5 baig directory reference

	Synopsis
amc.h	Model Check using over(under)-approximation of the transition relation
amcAmc.c	Model check formula with approxmiations
amcBlock.c	Set best system
amcCmd.c	Ends the amc package
amcInt.h	Internal data type definitions and macros to handle the structures of the amc package

2.4 bmd

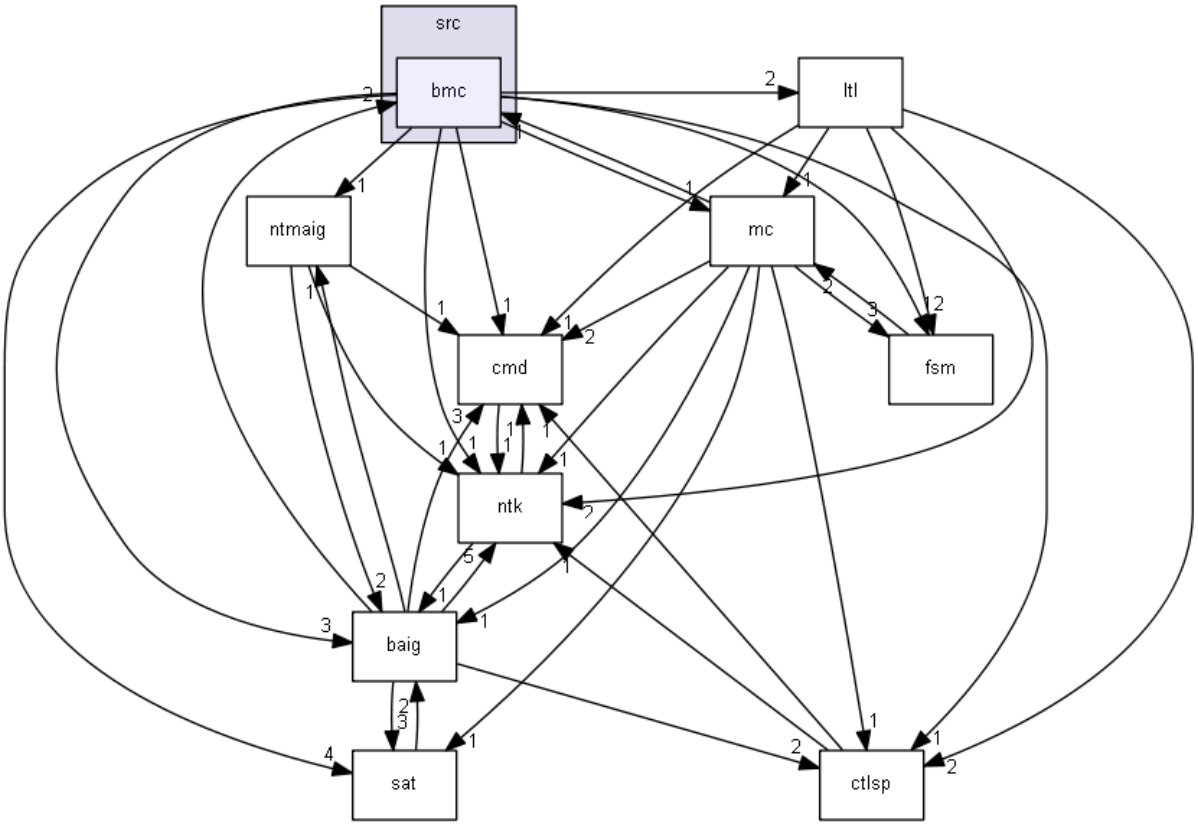

그림 6 bmc directory reference

	Synopsis
bmc.h	bounded model checking (BMC) header file
bmcAutSat.c	Automaton for BMC
bmcAutUtil.c	Utility file for BMC_Automaton
bmcBmc.c	SAT-based ltl model checker
bmcCirCUs.c	BMC ltl model checker using CirCUs
bmcCirCUsUtil.c	Utilities for bmcCirCUs
bmcCmd.c	Command interface for bounded model checking (bmc)
bmcInt.h	Internal header declarations
bmcUtil.c	Utilities for BMC package

2.5 ctlp

This package implements a parser for CTL (Computation Tree Logic) formulas. CTL is a language used to describe properties of systems.

External functions and data strucures of the binary AND/INVERTER graph package.

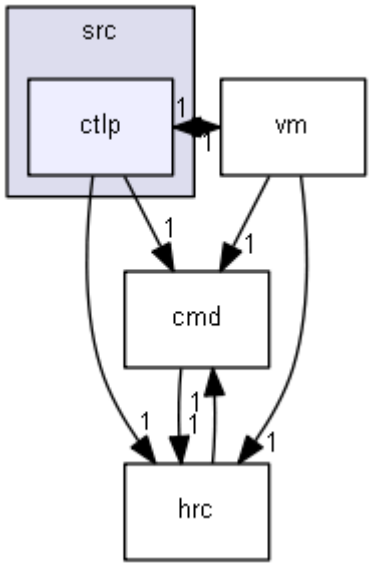

그림 7 ctlp directory reference

	Synopsis
ctlp.h	Routines for parsing, writing and accessing CTL formulas
ctlpCmd.c	Command to read in a file containing CTL formulas
ctlpInt.h	Declarations for internal use
ctlpUtil.c	Routines for manipulating CTL formulas

2.6 ctlsp

This package implements a parser for CTL* formulas. CTL* is a language used to describe properties of systems

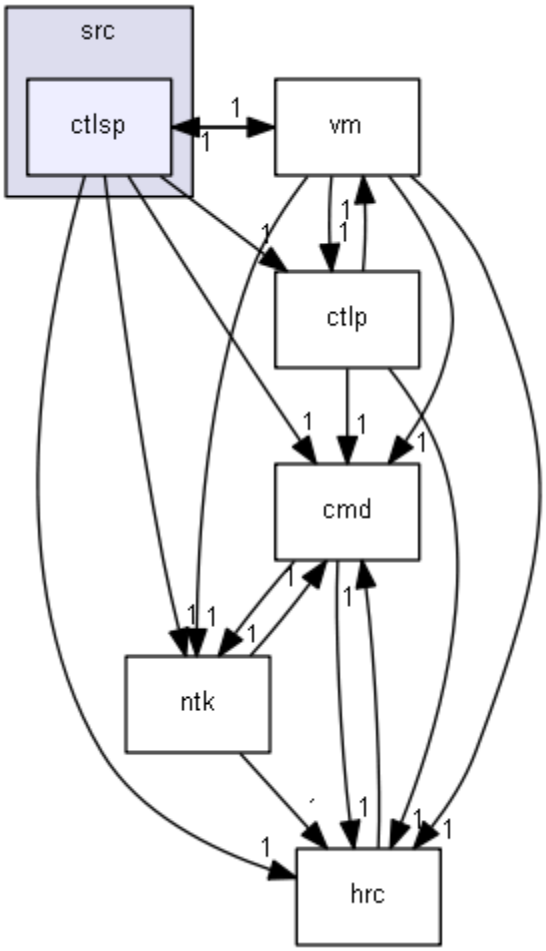

그림 8 ctlsp directory reference

	Synopsis
ctlsp.h	Routines for parsing, writing and accessing CTL* formulas
ctlspCmd.c	Command to read in a file containing CTL* formulas
ctlspInt.h	Declarations for internal use
ctlspUtil.c	Routines for manipulating CTL* formulas

2.7 eqv

The eqv package provides two commands - comb_verify and seq_verify for doing combinational and sequential verification respectively. Two networks along with options for partitioning and ordering form the input to the commands.

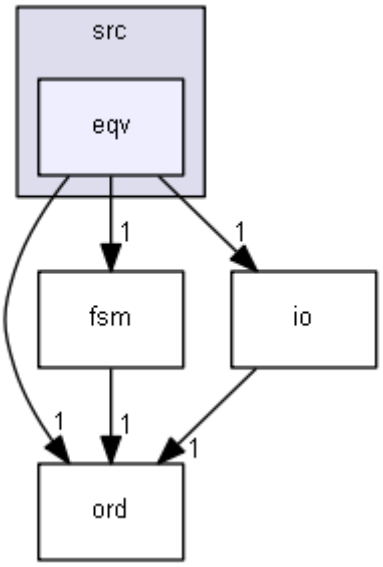

그림 9 eqv directory reference

	Synopsis
eqvCmd.c	Implements the eqv commands
eqvInt.h	
eqvMisc.c	This file provides some miscellaneous functions for the eqv package
eqvVerify.c	This file contains the two routines which do combinational and sequential verification
eqv.h	Provides commands for doing combinational and sequential verification on two networks

2.8 fsm

A finite state machine contains a pointer to a network, so it inherits all the information that is contained in the network, such as node names and MDD ids. An FSM has a partition associated with it; the output and next state functions are derived from the partition. This partition does not necessarily have to be the default partition associated with the network.

	Synopsis
fsm.h	Finite state machine abstraction of a network
fsmArdc.c	Routines to perform overapproximate reachability on the FSM structure
fsmCmd.c	Commands for the FSM package
fsmFair.c	Implementation of fairness constraints data structure
fsmFsm.c	Routines to create and manipulate the FSM structure
fsmHD.c	Routines to perform high density reachability on the FSM structure
fsmInt.h	Internal declarations for fsm package
fsmReach.c	Routines to perform reachability on the FSM structure

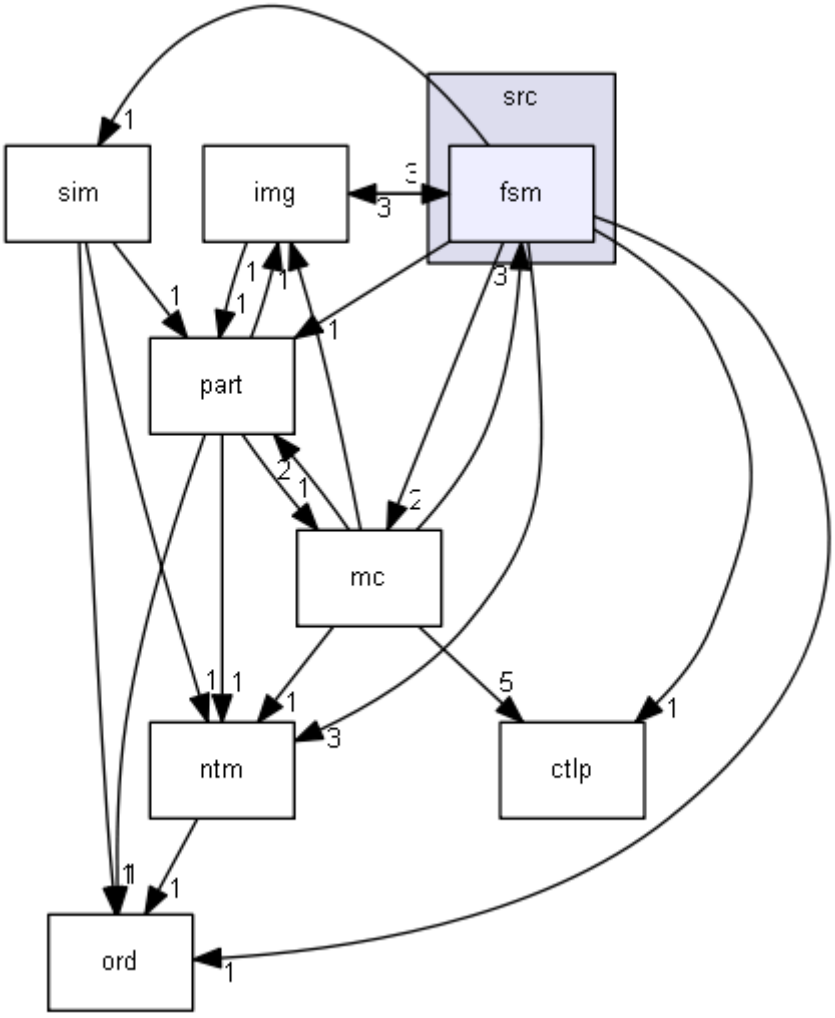

그림 10 fsn directory reference

2.9 grab

Abstraction refinement for large scale invariant checking

	Synopsis
grab.h	Abstraction refinement for large scale invariant checking
grabBMC.c	Check abstract paths on a more concrete model using SAT
grabGrab.c	The GRAB algorithm of computing a set of refinement variables
grabInt.h	Internal declarations
grabUtil.c	The utility functions for abstraction refinement
grab.c	Abstraction refinement for large scale invariant checking

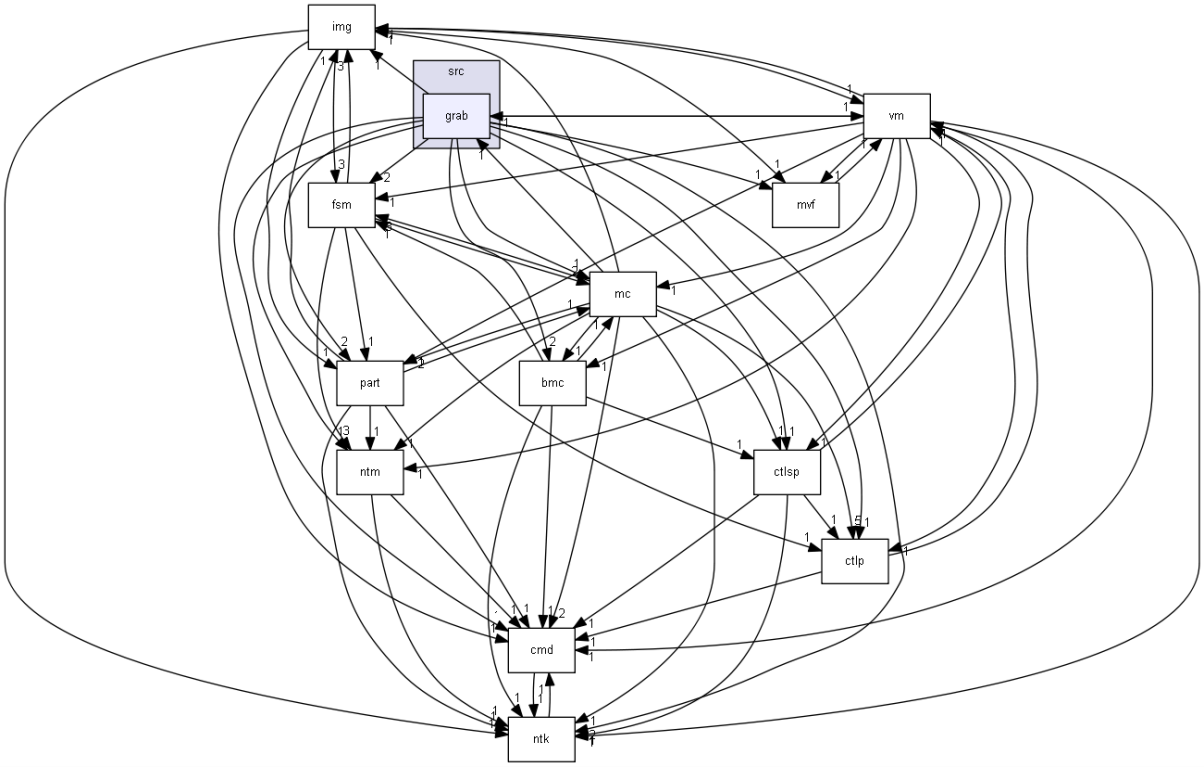

그림 11 grab directory reference

2.10 hmc

The concept of a hierarchy is directly related to the constructs in a blifmv file. It is built around three important data structures viz. Hrc_Model_t, Hrc_Subckt_t and Hrc_Node_t. These will be referred to as model, subckt and node.

A model corresponds directly to a model definition in a blifmv file. It can be viewed as a black box with some I/O pins. A model could call another model as a module within itself using names for the I/O pins which are, in general, different from those actually used inside the model that is called. The correspondence between the two sets of names is stored in a subckt. A call to a model will henceforth be referred to as an instantiation of the model.

Since a given model could be called by many other models, it is necessary to distinguish between its different instantiations. An instantiation is represented by a node. A hierarchy is a tree of nodes. The root node of the tree corresponds to the single instantiation of the root model in a blifmv file. An instantiation of a model results in the instantiation of all models recursively called by it. Thus, the instantiation of root model results in a tree structure being formed.

There is also a structure called a hierarchy manager which contains a list of all models and pointers to the root node and the current node. The current node represents the current position in the hierarchy. The designer can, if he wishes, make changes in this node only, or modify the whole sub-tree below it.

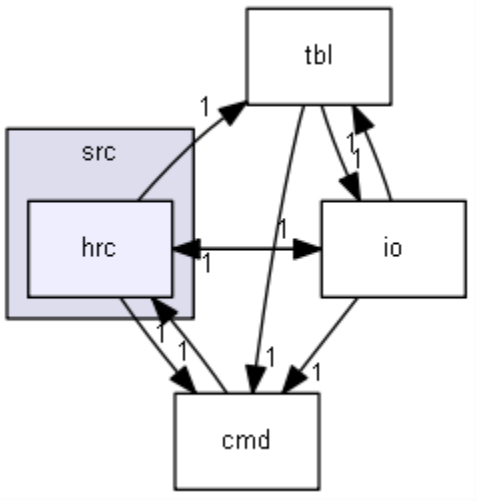

그림 12 hrc directory reference

	Synopsis
hrc.h	Hierarchical representation of a design
hrcCmd.c	Commands for walking around HSIS/VIS hierarchies
hrcHierarchy.c	Creates a hierarchy from model definitions
hrcInOut.c	This file provides the functions for accessing the fields of the data structures of the hrc package
hrcInt.h	The internal declarations needed for the hierarchy
hrcMemUtil.c	This file deals with the memory utilities for the hrc package
hrcMisc.c	This file provides some miscellaneous functions
hrcModify.c	This files provides the basic functions concerned with modifying the hierarchy.

2.11 img

	Synopsis
imc.h	Model Check using over(under)-approximation with automatic iterative refinement
imcCmd.c	Command interface for the imc package
imcImc.c	Incremental Model Checker
imcInt.h	Internal data type definitions and macros to handle the structures of the imc package

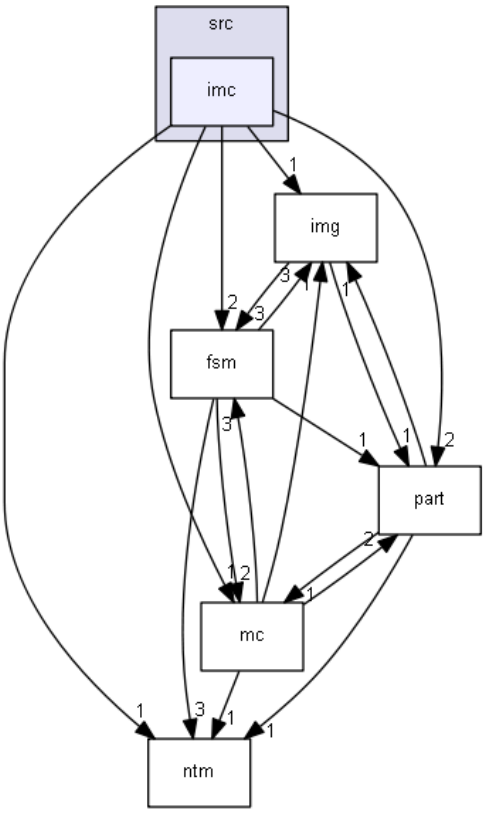

그림 13 imc directory reference

2.12 img

The image package is used to compute the image (forward or backward) of a set under a vector of functions. The functions are given by a graph of multi-valued functions (MVFs). This graph is built using the partition package. Each vertex in this graph has an MVF and an MDD id. The fanins of a vertex v give those vertices upon which the MVF at v depends. The vector of functions to use for an image computation, the "roots", is specified by an array of (names of) vertices of the graph. The domain variables are the variables over which "from" sets are defined for forward images, and "to" sets are defined for backward images. The range variables are the variables over which "to" sets are defined for forward images, and "from" sets are defined for backward images. The quantify variables are additional variables over which the functions are defined; this set is disjoint from domain variables. These variables are existentially quantified from the results of backward image computation.

Computing images is fundamental to many symbolic analysis techniques, and methods for

computing images efficiently is an area of ongoing research. For this reason, the image package has been designed with lots of flexibility to easily allow new methods to be integrated (to add a new method, see the instructions in imgInt.h). Applications that use the image package can switch among different image methods simply by specifying the method type in the image initialization routine. By using the returned structure (Img_ImageInfo_t) from the initialization routine, all subsequent (forward or backward) image computations will be done using the specified method.

VIS users can control which image method is used by appropriately setting the "image_method" flag. Also, VIS users can set flags to control parameters for different image computation methods. Because the user has the ability to change the values of these flags, Img_ImageInfo_t structs should be freed and re-initialized whenever the VIS user changes the value of these flags.

Following are descriptions of the methods implemented. In the descriptions, x=x_1,...x_n is the set of domain variables, u=u_1,...,u_k is the set of quantify variables, y=y_1,...,y_m is the set of range variables, and f=f_1(x,u),...,f_m(x,u) is the set of functions under which we wish to compute images.

Monolithic: This is the most naive approach possible. A single relation T(x,y) is constructed during the initialization phase, using the computation (exists u (prod_i(y_i = f_i(x,u)))). To compute the forward image, where fromUpperBound=U(x), fromLowerBound=L(x), and toCareSet=C(y), we first compute a set A(x) between U(x) and L(x). Then, T(x,y) is simplified with respect to A(x) and C(y) to get T*. Finally, x is quantified from T* to produce the final answer. Backward images are computed analogously. The monolithic method does not recognize any user-settable flags for image computation.

IWLS95: This technique is based on the early quantification heuristic. The initialization process consists of following steps:

- Create the relation of the roots at the bit level in terms of the quantify and domain variables.
- Order the bit level relations.
- Group the relations of bits together, making a cluster whenever the BDD size reaches a threshold.
- For each cluster, quantify out the quantify variables which are local to that particular cluster.
- Order the clusters using the algorithm given in "Efficient BDD Algorithms for FSM Synthesis and

Verification", by R. K. Ranjan et. al. in the proceedings of IWLS'95{1}.

• The orders of the clusters for forward and backward image are calculated and stored. Also stored is the schedule of variables for early quantification.

For forward and backward image computation the corresponding routines are called with appropriate ordering of clusters and early quantification schedule.

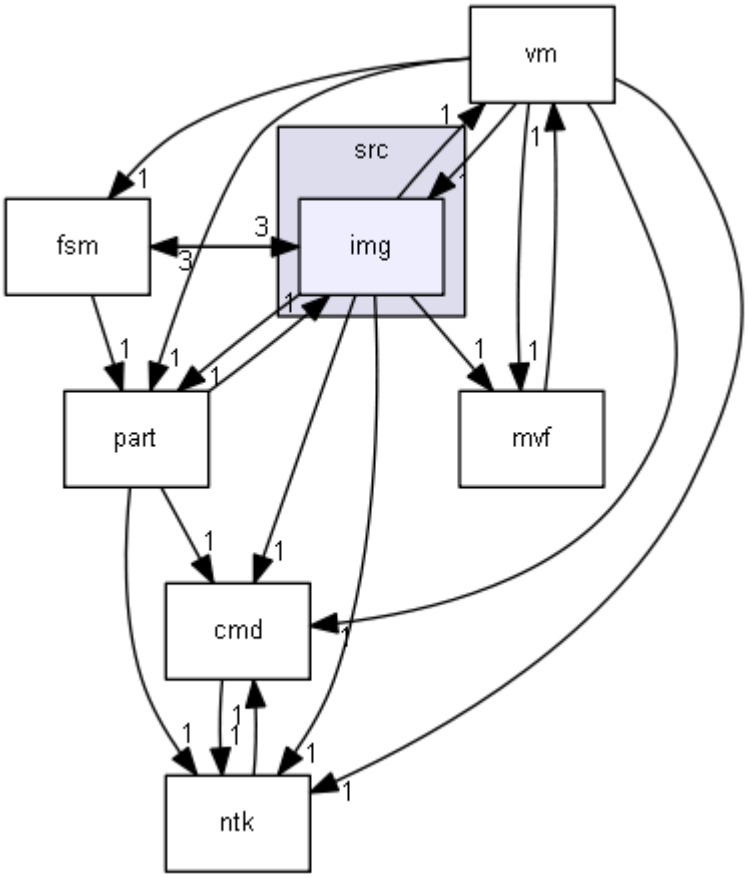

그림 14 img directory reference

	Synopsis
img.h	Methods for performing image computations
imgTfm.c	Routines for image and preimage computations using transition function method
imgTfmBwd.c	Routines for preimage computation using transition function method
imgTfmCache.c	Routines for image cache in transition function method
imgTfmFwd.c	Routines for image computation using transition function method
imgInt.h	Internal declarations for img package
imgIwls95.c	Routines for image computation using component transition relation approach described in the proceedings of IWLS'95
imgLinear.c	Routines for image computation using Linear Arrangement published in TACAS02
imgMlp.c	Routines for image computation using MLP(Minimal Lifetime Permutation) published in FMCAD00
imgMonolithic.c	Routines for image computation using a monolithic transition relation
imgTfmUtil.c	Routines for image computation using transition function method
imgUtil.c	High-level routines to perform image computations

imgHybrid.c	Routines for hybrid image computation

2.13 io

Routines for reading and writing BLIF-MV files. Routines to test if a blif-mv file is consistent. They are also responsible for setting several internal data structures. Here are the list of checks we can do on an hsis network. For each model, we first check if there is no node labeled both as PI and PS or both as PI and PO. Then, for each subcircuit in the model, the compatibllity of the interface is verified namewise and rangewise. This is detailed below. Then, we verify that there is no combinational cycle in any model. Furthermore, for each latch in the model, we make sure that the input and the output of the latch are of the same type and that every latch has a reset table. Finally, we check to see if each variable is an output of at most one table. As for the check to be done for a subcircuit, we first check if the model to be instantiated is present in the hmanager. Then, we test if all the formal variables in the subcircuit definition exist in the model and at the same time they are of the same type of the corresponding actual variables. More thoroughly, we have to check if a flattened network has no cycle, but it is not currently implemented.

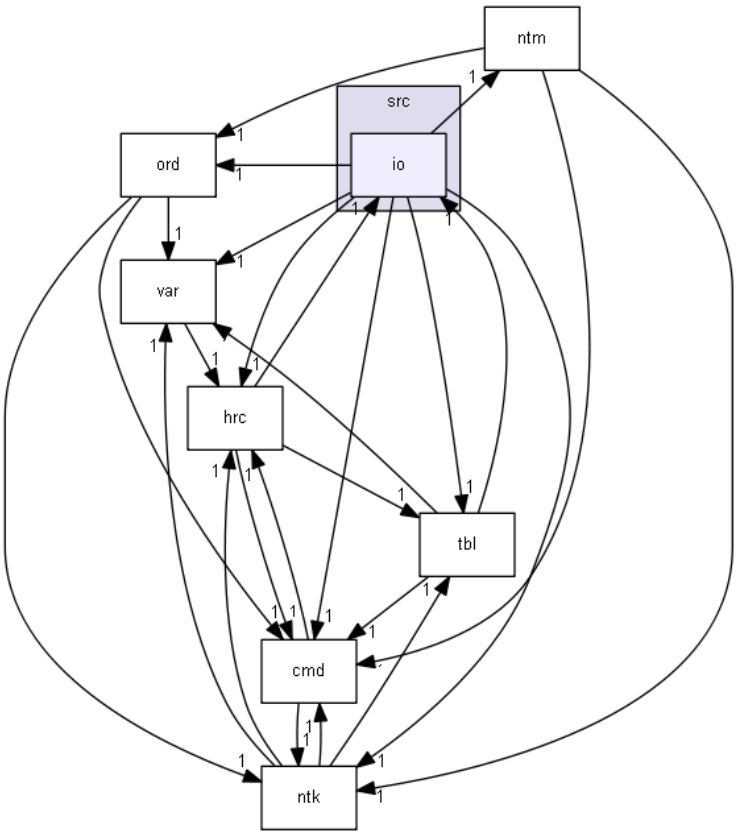

그림 15 io directory reference

	Synopsis
io.h	Routines for reading and writing BLIF-MV files
ioCheck.c	Routines to test if a blif-mv file is consistent
ioCmd.c	Top-level routines for reading and writing files
ioInt.h	Internal declarations of the I/O package
ioParse.c	Functions used in parsing BLIF-MV files
ioReadBlifMv.c	Routines related to reading in blif-mv files
ioTable.c	Routines for generating the table data structure from textual information. Used in the parser
ioWriteBlif.c	This file contains blifmv -> blif write routines
ioWriteBlifIo.c	This file contains blifmv -> blif write routines that handle the functionality of files IO
ioWriteBlifMv.c	Writes out a blif-mv file
ioWriteBlifUtil.c	This file contains blifmv -> blif write routines, which perform miscellaneous lower-level functions
ioWriteSmv.c	Writes out an Smv file
aiger.c	
aiger.h	

2.14 ltl

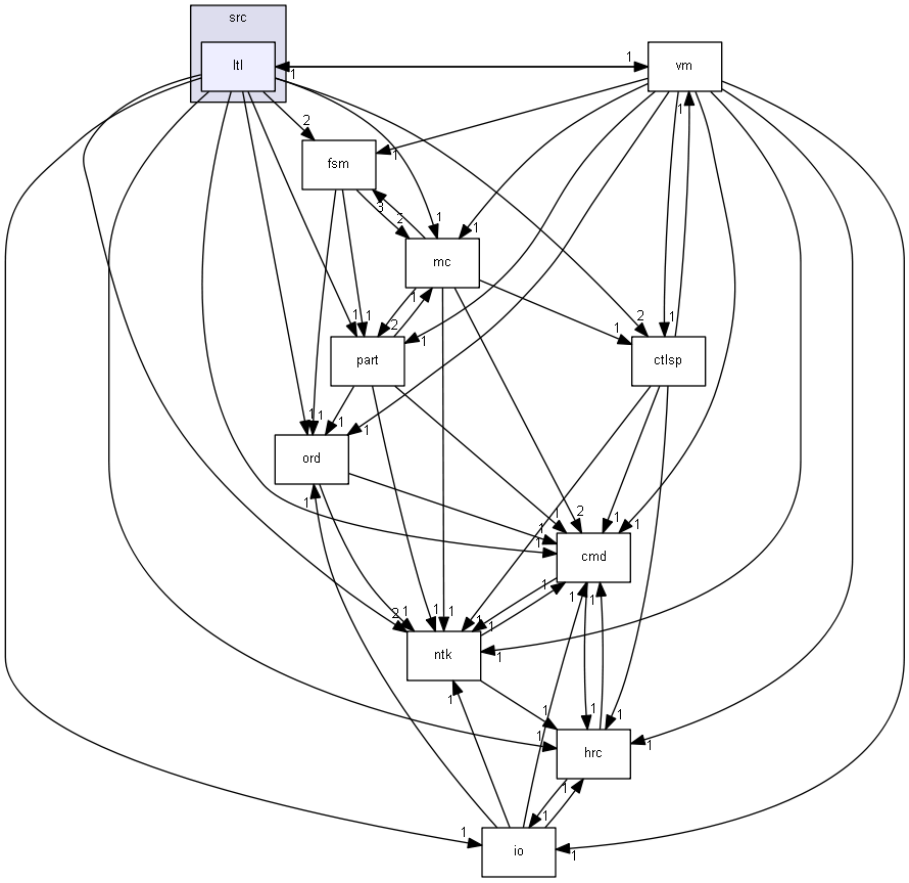

그림 16 ltl directory reference

	Synopsis
ltl.h	Internal declarations
ltl.c	LTL model checking
ltlAutomaton.c	Translate LTL formula to the Buechi Automaton
ltlCompose.c	Write the Buechi automaton into a file
ltlInt.h	Internal declarations
ltlMinimize.c	Buechi automaton minimization
ltlSet.c	Set/Pair/Cover Manipulation functions used in the ltl package
ltlTableau.c	Expand the LTL Formula by applying the Tableau Rules
ltlUtil.c	Utilities for LTL model checker

2.15 maig

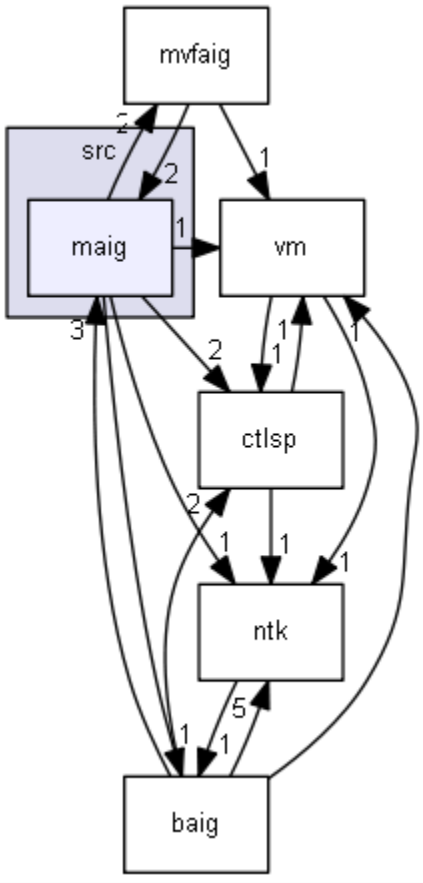

그림 17 maig directory reference

	Synopsis
maig.h	multi-valued AND/INVERTER graph
maigUtil.c	Utilities for Multi-Valued AndInv graph
maigInt.h	Internal data structures of Multi-vlaued AND/INVERTER package

2.16 mark

Data structures used in Markovian analysis

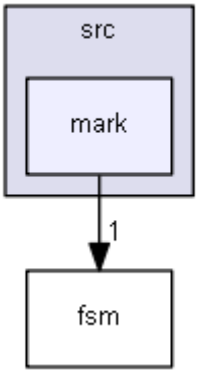

그림 18 mark directory reference

	Synopsis
mark.h	Data structures used in Markovian analysis
markFPSolve.c	This file contains functions that implement the fixed point method.
markGetScc.c	Functions to compute terminal strongly connected components in a markov chain
markInProb.c	Computation of transition probability matrix and convergence checks in markov analysis
markInt.h	Data structures used in Markovian analysis
mark.c	External procedures included in this module: - Mark_FsmComputeStateProbs() - Mark_ComputeStateProbsWithTr() Internal procedures included in this module: - MarkAverageBitChange()

2.17 mc

Fair CTL model checker and debugger. Works on a flattened network

	Synopsis
mc.h	Fair CTL model checker and debugger
mcCmd.c	Functions for CTL model checking commands
mcCover.c	Functions for coverage estimation
mcDbg.c	Debugger for Fair CTL models
mcDnC.c	The Divide and Compose (D'n'C) Approach of SCC Enumeration
mcGFP.c	Computation of greatest fixpoints
mcInt.h	Internal declarations
mcMc.c	Fair CTL model checker
mcSCC.c	Computation of Fair Strongly Connected Components
mcUtil.c	Utilities for Fair CTL model checker and debugger
mcVacuum.c	Functions for vacuity detection

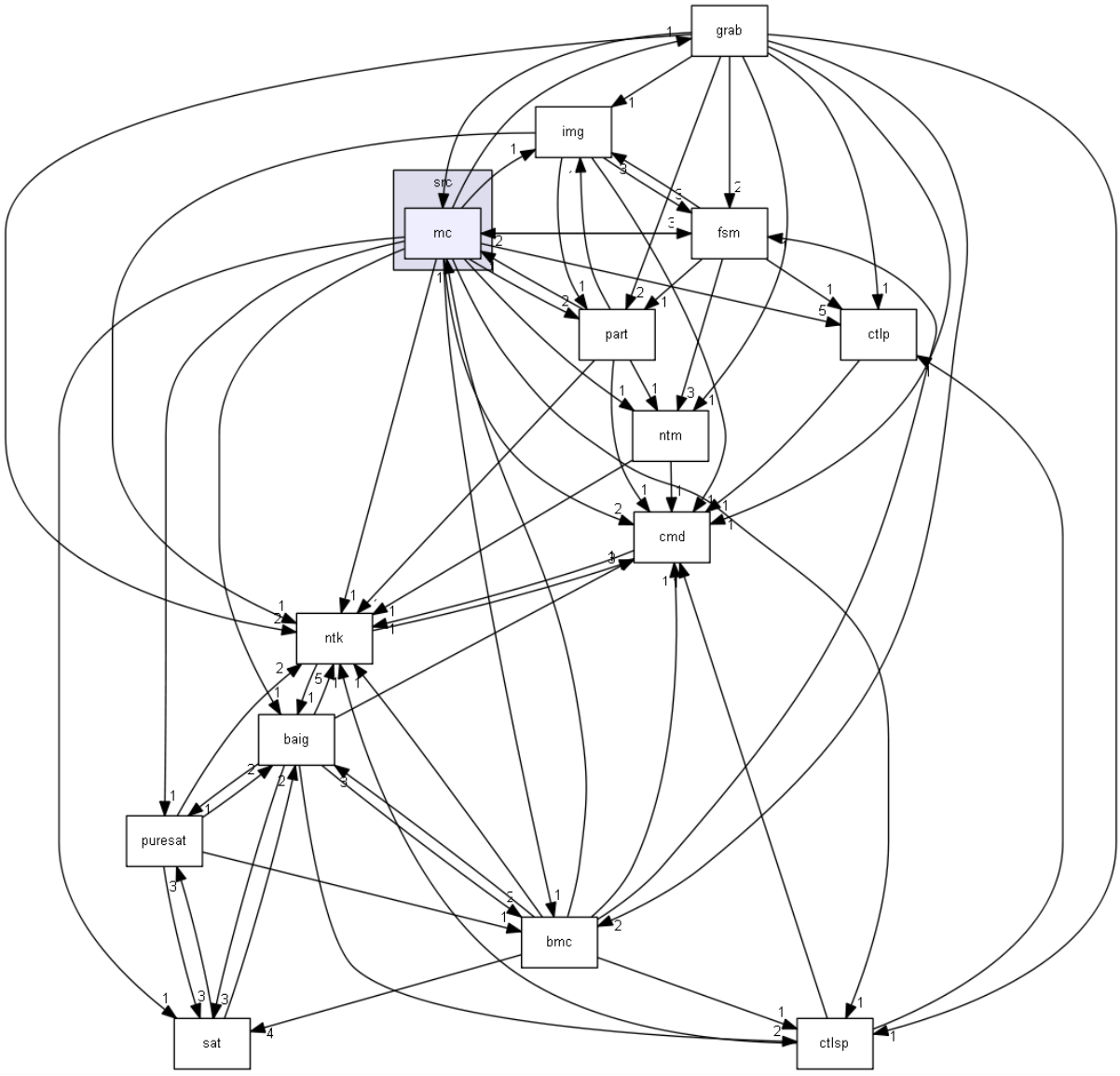

그림 19 mc directory reference

2.18 mvf

This package is used to create and manipulate single output functions that take multiple values, and are defined over multi-valued variables. Mathematically, such a function is described as, f: Y1 x Y2 x ... x Yn –> Yn+1. Each Yi is a finite, ordered set; if Yi is of cardinality k, then the elements of Yi are {0, 1, ..., k-1}. We use yi do denote a variable over Yi. If

A single MDD over variables y1,...,yn cannot be used to represent f, because an MDD can only represent binary-valued functions, not multi-valued functions. Instead, to represent f, we use

an array of MDDs, of length equal to the cardinality of Yn+1. Each MDD of this array is defined over y1,...,yn. Furthermore, the minterms for which the ith MDD, fi, evaluates to one, are exactly those minterms for which f evaluates to the ith member of Yn+1. If f is deterministic, then the intersection of fi and fj, for i not equal to j, is empty. If f is completely specified, then the union of the fi's is the tautology. The union of the fi's is referred to as the "domain" of the function.

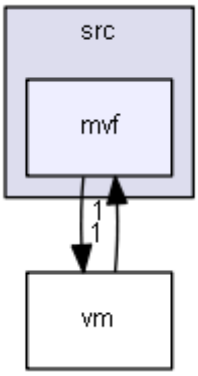

그림 20 mvf directory reference

	Synopsis
mvf.h	Creation and manipulation of MDD-based multi-valued functions
mvfInt.h	Internal definitions for the mvf package
mvfMvf.c	Routines to create, manipulate and free multi-valued functions

2.19 mvfaig

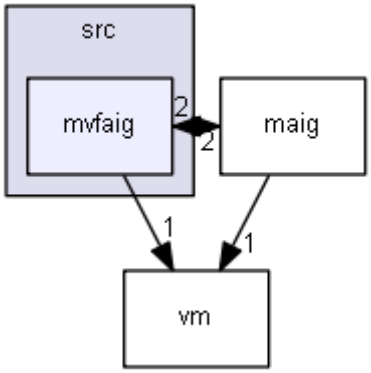

그림 21 mvfaig directory reference

	Synopsis
mvfaig.h	Creation and manipulation of AndInv-based multi-valued functions
mvfaigInt.h	Internal definitions for the mvf package using And/Inverter graph
mvfaigUtil.c	Routines to create, manipulate and free multi-valued functions

2.20 ntk

A network is a directed graph, where the vertices are nodes (of type Ntk_Node_t) and the edges are stored as fanin and fanout arrays of each node. Each node has a single output, which may fanout to multiple nodes. A node is one of 5 types. In addition, it may have one or more optional attributes, respecting the following table:

type	attributes			
	primary output	latch data input	latch init input	constant
pseudo input	x	x	x	
primary input	x	x	x	
latch	x	x		
shadow				
combinational	x	x	x	x

Legend:

primary input:	a node with no fanins that is a port of the network; does not have a table defining its function - it can take any value in its domain
pseudo input:	a node with no fanins, introduced to model nondeterminism; its table defines which values it can take
latch:	an edge triggered latch with data and initial inputs
shadow:	a node with no fanins and no fanouts; it serves as a "shadow" for a node; for example, a shadow node for a latch is used to store information about the next state variable; any node can have a shadow, except a shadow node
combinational:	a node that has a table and is not a primary or pseudo input
primary output:	a node that is a port of the network
latch data input:	a node driving the data input of a latch
latch init input:	a node driving the initial input of a latch
constant:	a combinational node with no fanins that can take exactly one value

An "x" in the table means that it's permissible for a node of the given type to have the given attribute; the absence of an "x" means it's not permissible.

In addition, there are several derived attributes: all primary inputs, pseudo inputs, and latches are "combinational inputs"; all latch data inputs, latch initial inputs, and primary outputs are "combinational outputs"; all primary inputs and pseudo inputs are "inputs".

Each combinational node has a table defining the function of the node in terms of its

immediate fanins. This function must be deterministic (i.e. for a given valuation of the fanins, the function can assume at most one value) and completely specified (i.e. for a given valuation of the fanins, the function can assume at least one value) in order to ensure correctness of verification results downstream . (The exception is that pseudo inputs are nondeterministic.) The table may have multiple output columns, so the node keeps an index giving the output column to which it refers.

The ntk package contains facilities to create a network (Ntk_Network_t) from a hierarchical network. In particular, it supports the concept of "actual names" and "formal names". Consider the following hierarchical network:

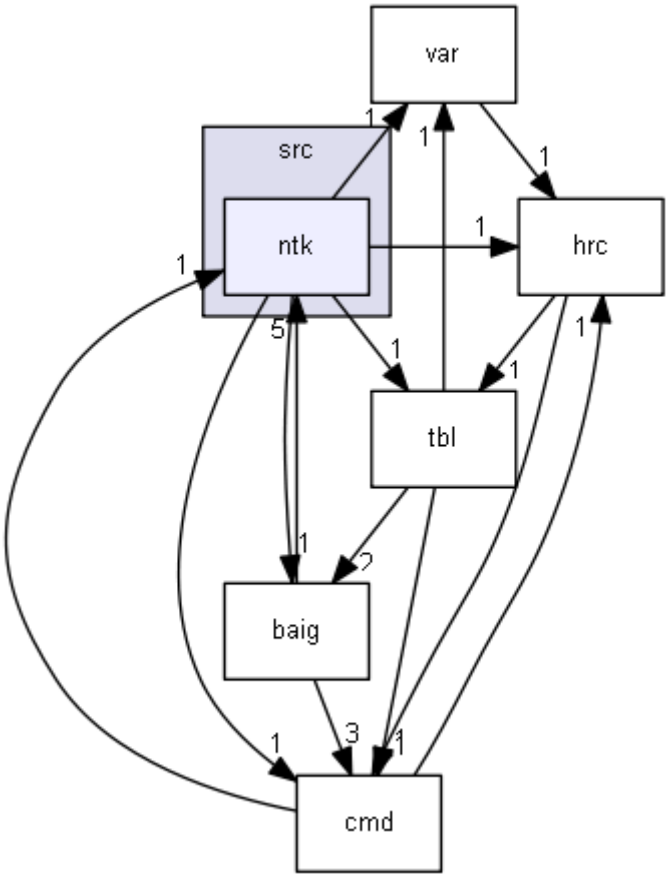

그림 22 ntk directory reference

	Synopsis
ntk.h	Flat network of multi-valued combinational nodes and latches
ntkFlt.c	Routines for creating a network from a hierarchy manager
ntkGraph.c	Routines related to the abstract graph of a network
ntkInt.h	Internal declarations for the network package
ntkNode.c	Routines to access the node data structure
ntkNtk.c	Routines to access the network data structure
ntkSweep.c	Utilities for Cleaning the Ntk_Network_t
ntkCmd.c	Command interface to the ntk package

2.21 ntm

Provides a routine to build the MVFs of the roots of an arbitrary region of a network, in terms of the leaves of the region. The leaves can be treated as variables or as specific constants.

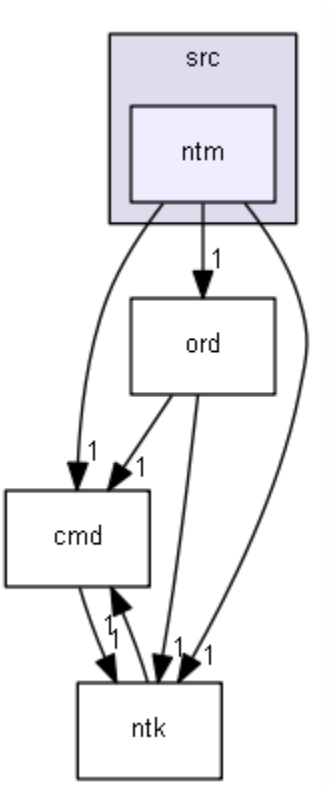

그림 23 ntm directory reference

	Synopsis
ntm.h	Construction of MDDs from a flattened network
ntmInt.h	Internal declarations
ntm.c	Routines to build MDDs from a network

2.22 ntmaig

	Synopsis
ntmaig.h	Construction of mAigs from a flattened network
ntmaigCmd.c	Command interface for the Aig partition package
ntmaigInt.h	Internal declarations
ntmaig.c	Routines to build mAigs from a network

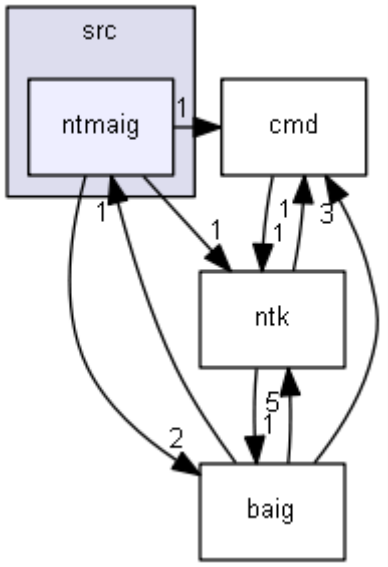

그림 24 ntmaig directory reference

2.23 ord

The routines in this package relate to ordering MDD variables corresponding to the nodes of a network. The enumerated type Ord_OrderType is used to specify to which set of nodes certain ordering routines should be applied. The following matrix shows which node types are included in each ordering type:

	Ord_OrderType		
node type	All	InputAndLatch	NextStateNode
primary input	x	x	
pseudo input	x	x	
latch	x	x	
shadow	x	x	x
combinational	x		

In addition, the order type Partial can be used to specify an arbitrary subset of nodes.

There are various methods for ordering the roots of a network, and for ordering the nodes of a network. These are explained in the documentation for the static_order command

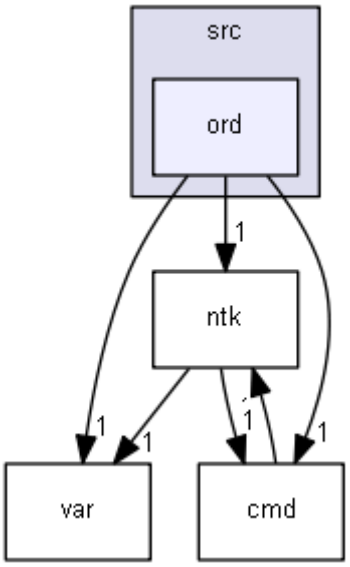

그림 25 ord directory reference

	Synopsis
ord.h	Routines for ordering MDD variables of a flattened network
ordCmd.c	Command interface to the ordering package
ordInt.h	Internal declarations for the order package
ordIo.c	Routines to read and write variable orderings
ordMain.c	Routines for static ordering of MDD variables
ordNodes.c	Routines to order the nodes in the TFI of the roots of a network
ordPerm.c	Routines to find permutation on latches to minimize MDD size
ordRoots.c	Routines to order the roots of the network

2.24 part

Once the description of a system has been read, and the ordering of the variables has been assigned, the partition package creates an abstracted view of the system in which only information in terms of MDDs is stored. The MDDs belong to the MDD manager of the system. Different options may be considered when creating this abstracted view. If the system is described as a network there are several options to create this partition. As a first choice, the system may be considered as a set of functions representing the combinational outputs as functions of the combinational inputs. In general, the latch functions may be specified as functions of any intermediate variables. These intermediate variables are themselves functions of other variables, and ultimately the dependency on the combinational inputs is achieved. The structure to represent these arbitrary dependencies is a DAG. The combinational inputs of the network will be represented as vertices. Every other function (either representing a latch or any other intermediate node) is represented as a vertex with in-coming edges from the vertices representing the

function's domain. Hence, the vertices representing the combinational inputs will not have any in-coming edges, and conversely, the vertices representing the combinational outputs will not have any fanout edges.

The partition may have two types of vertices, called single and clustered. Single vertices are the ones that represent, for example, nodes in a network. Clustered vertices are used solely for the purpose of grouping single vertices into disjoint sets. No clustered vertex can be member of a clustered vertex, and every single vertex may be a member of a unique clustered vertex. The edges of the graph are connecting only single vertices. Functions are provided to access the list of vertices represented by a clustered vertex as well as for testing the type of a vertex.

A partition is the central input to the image computation package. However, it is important to note that there is no network-specific information stored in the partition data structure itself. Hence, it is possible that another application (i.e. besides the network application) could create a partition, and use that partition as input to the image computation package.

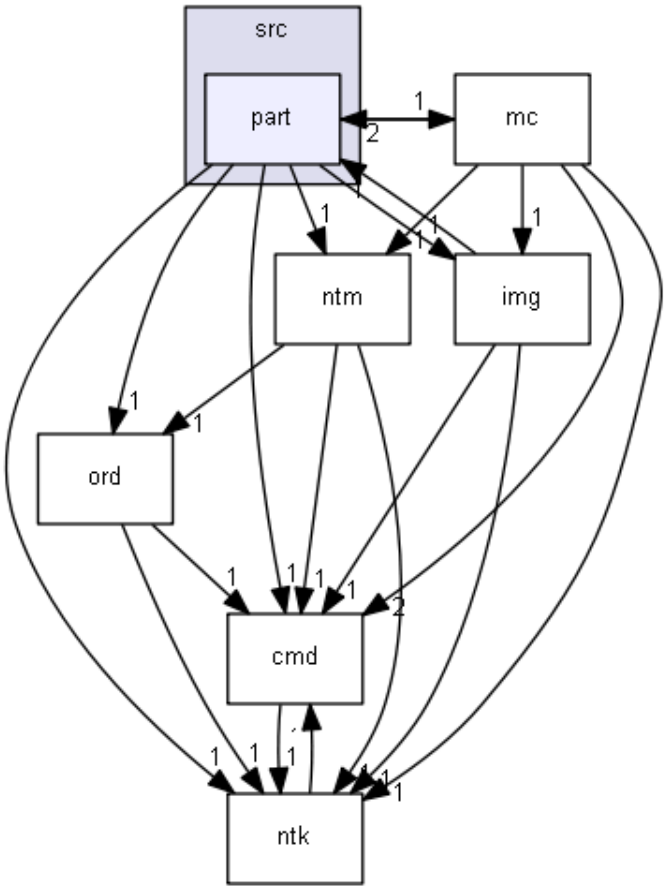

그림 26 part directory reference

	Synopsis
part.h	Partition of a system and creation of MDDs
partFine.c	Implements the partition of the network with respect to a list of nodes provided by the user
partFrontier.c	Implements the partition of the network based on the strategy of creating a node whenever the size of the BDD representing the functionality of the node increases the threshold value
partGroup.c	Routines for grouping vertices
partInOut.c	Implements the partition of a network considering only the functions at the combinational outputs in terms of the combinational inputs
partInt.h	Internal data type definitions and macros to handle the structures of the partition package
partPart.c	Routines to initialize the command build_partition_mdds, create and delete internal data structures of the partition, and print information about the partition
partPartial.c	Implements the partition of the network with respect to a list of nodes provided by the user
partTotal.c	Implements the partition of the network replicating exactly the network structure in the partition graph
partBoundary.c	Implements the partition of the network with respect to the nodes that comprise the submodules boundaries
partCmd.c	Command interface for the partition package
partCollapse.c	Implements a procedure to collapse several internal vertices of a partition into a single one

2.25 puresat

Abstraction refinement for large scale invariant checking

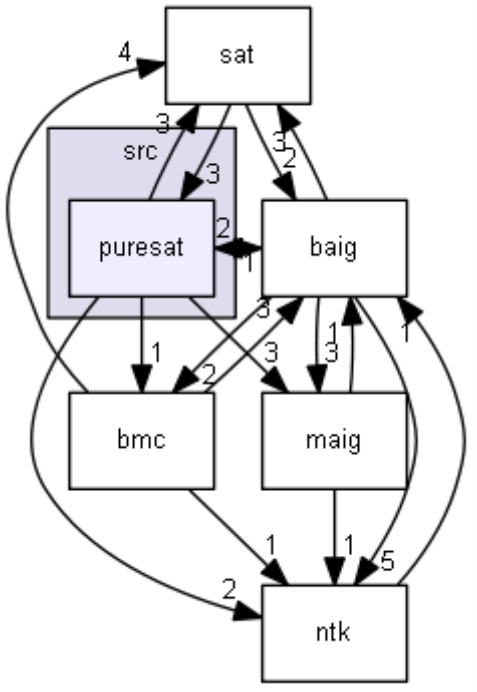

그림 27 piresat directory reference

	Synopsis

puresat.h puresatAig.c puresatArosat.c puresatMain.c puresatFlatIP.c puresatAbRf.c puresatIPRefine.c puresatIPUtil.c puresatMain.c puresatRefine.c puresatTFrame.c puresatUtil.c puresat.c	Abstraction refinement for large scale invariant checking
puresatInt.h	Internal declarations

2.26 res

This package implements residue verification between two networks. The method used is based on residue arithmetic and the Chinese Remainder theorem. Verification is performed by interpreting the outputs of the circuits as integers and verifying the residues of the outputs with respect to a set of moduli. The choice of moduli is directed by the Chinese Remainder Theorem in order to prove equivalence of the two circuits. This method works well with multipliers and possibly other arithmetic circuits (due to its dependence on residue arithmetic). Discretion should be exercised in applying this method to general combinational circuits. Residue verification is provided with vis-cu ONLY. It reads both blif and blif-mv files. However, it does NOT support multi-valued variables. Residue verification is primarily for combinational verification, but may be applied to sequential circuits with the same state encoding. The latch outputs are then considered to be combinational inputs of the circuits and the latch inputs and reset are considered to be combinational outputs of the circuits. This package provides some combinational verification also. Some/all of the outputs of the circuit may be verified directly (without using residues). In using both direct and residue verification, verification of arithmetic circuits may become easier.

	Synopsis
res.h	Combinational verification by residue arithmetic
resCmd.c	Implements the different commands related to the residue verification
resCompose.c	This file contains all relevant procedures for the composition of the nodes of the circuit into the residue Add
resInt.h	Internal declarations of the residue package
resLayer.c	This file is responsible for computing the "layers" in a circuit depending on the method
resRes.c	Data manipulation routines of the Res_ResidueInfo structure
resSmartVarUse.c	This file provides functions to handle Dd variables in the composition phase of the residue verification
res.c	The main file that incorporates procedures for residue verification

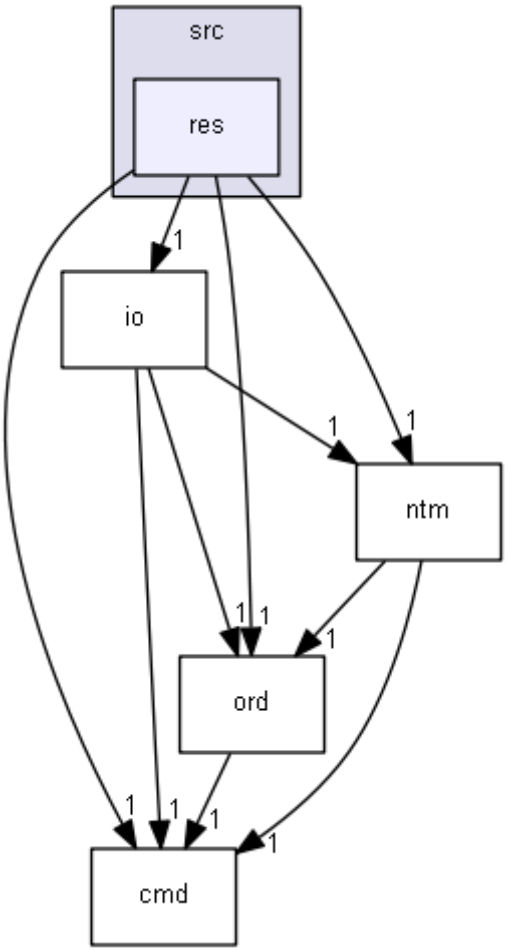

그림 28 res directory reference

2.27 restr

This package provides the capability to restructure a state transition graph (STG).

	Synopsis
restr.h	STG restructuring package
restrCProj.c	This file contains functions that implement the STG restructuring based on compatible projection
restrDebug.c	Utility functions to aid in debugging
restrFaninout.c	Routines in this file implement the Fanin and Fanin-Fanout oriented restructuring heuristics
restrHammingD.c	The function in this file implements the Hamming distance heuristic to restructure the STG
restrInt.h	Internal declarations for state transition graph (STG) restructuring package
restrRestructure.c	This file contains a main procedure that restructures an STG and transforms it into a new multilevel circuit
restrUtil.c	Support functions used in the package
restrCmd.c	Command interface for the restr package

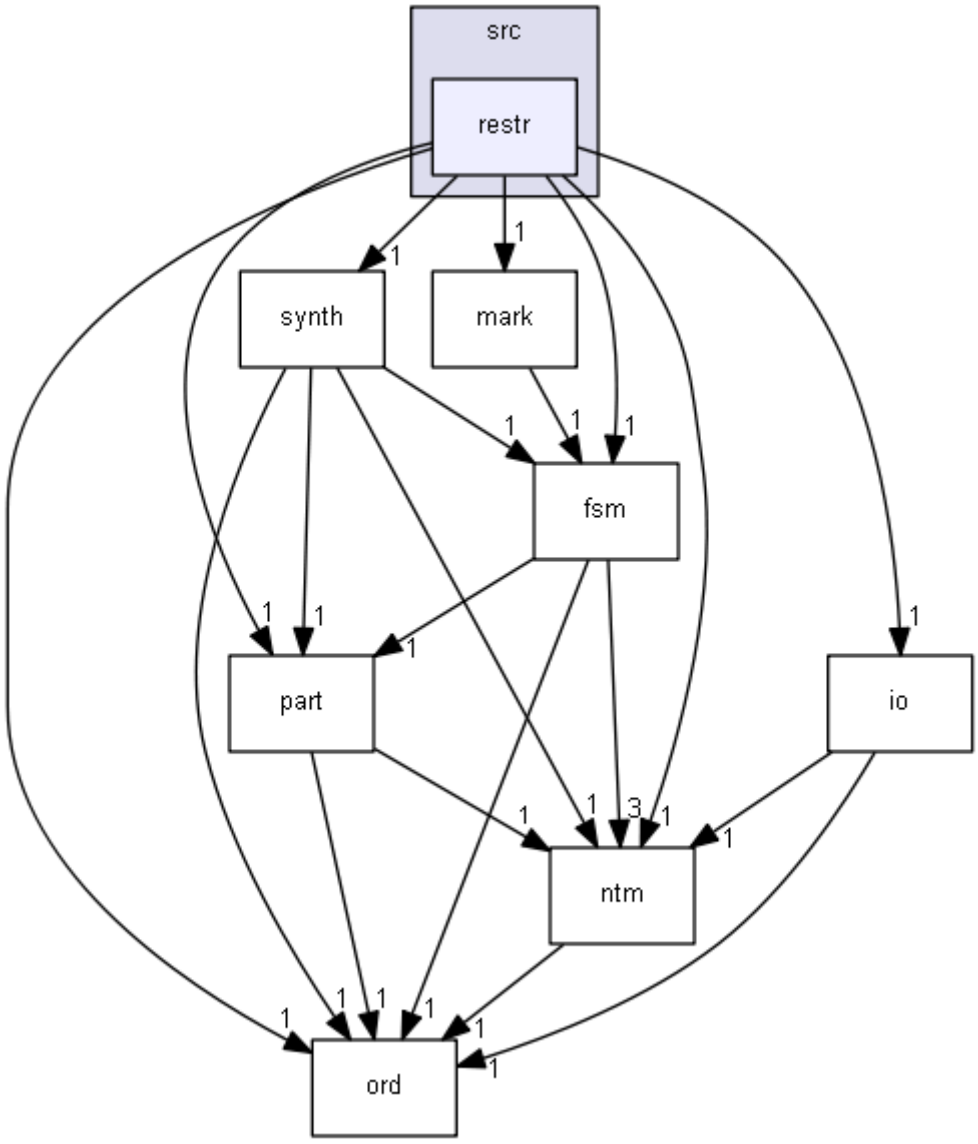

그림 29 restr directory reference

2.28 rst

This package provides the capability to restructure the user's hierarchy. Currently only collapsing adjacent nodes is provided. Parititioning FSMs is expected to be provided in the future.

	Synopsis
rst.h	Restructuring package for restructuring the hierarchy
rstGroup.c	rst package partitioning code with user interface
rstInt.h	Internal declarations
rst.c	utilities for restructuring hierarchy

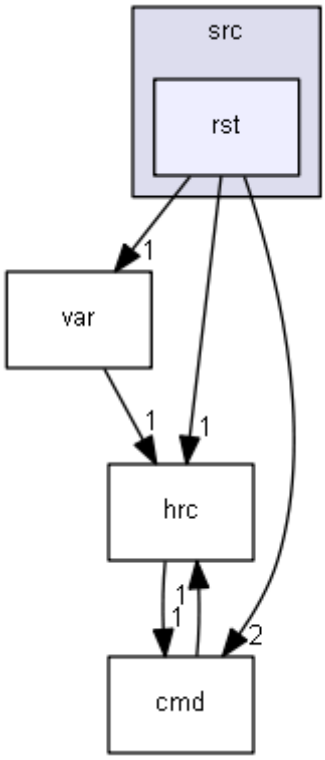

그림 30 rst directory reference

2.29 rt

This file contains the functions for regression test. The command 'regression_test' will read a description of the regression test to be performed from the argument file, and will then carry out the test. Finally, it will produce a LaTeX file with a set of tables summarizing the results. The regression_test command can be used to compare two different versions of vis, or two different scripts on the same version of vis, or even two different scripts on two different versions of vis. regression_test spawns the version(s) of vis to be used for the experiments as child processes. It does no other computation except setting up the experiments, starting the child processes, waiting for their termination, and collecting the results.

	Synopsis
rt.h	Regression test
rtInt.h	Internal declarations
rtMain.c	Regression Test

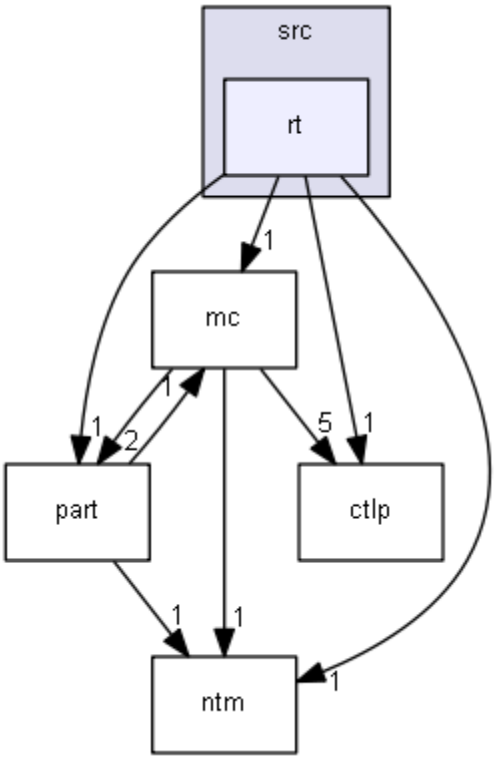

그림 31 rt directory reference

2.30 sat

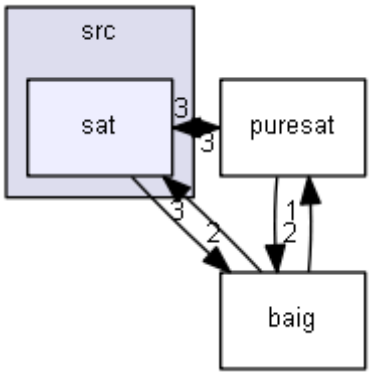

그림 32 sat directory reference

	Synopsis
sat.h	Internal data structures of the sat package
satCore.c	Routines for UNSAT core generation,both CNF-based and Aig-based UNSAT core generetions are available
satDebug.c	Routines for various debug function
satDecision.c	Routines for various decision heuristics

satImplication.c	Routines for BCP
satInc.c	Routines for sat incremental function
satInt.h	Internal data structures of the sat package
satInterface.c	Routines for various decision heuristics
satMain.c	Routines for sat main function
satUtil.c	Routines for various utility function
satBDD.c	Routines for using BDD
satConflict.c	Routines for sat conflict analysism and unsat proof generation for both CNF format and AIG format

2.31 sim

The sim package provides functions for simulation. It also provides the "simulate" command in the VIS environment. The simulate command performs a simulation of a network or a part of a network. sim conceives simulation through three operations:

1- Build simulation vectors.

2- Simulate.

3- Print the result.

Exported functions make it possible to build an internal data-structure, and perform the three operations given above. Simulation vectors may be provided by the user. Using exported functions, it is also possible to generate simulation vectors randomly or reading it from a file. Low level functions are also available that evaluate, for example, a network node using a simulation vector. An example of a simulation vectors file may be generated using the simulate command with random-vectors-generation option, in VIS.

	Synopsis
sim.h	Simulation of a flattened network
simInt.h	sim package internal declarations file
simIo.c	Routines to read and write simulation vectors
simMain.c	simulation of a Network
simSim.c	Routines to manipulate simstructure
simUtil.c	Basic useful functions for the sim package

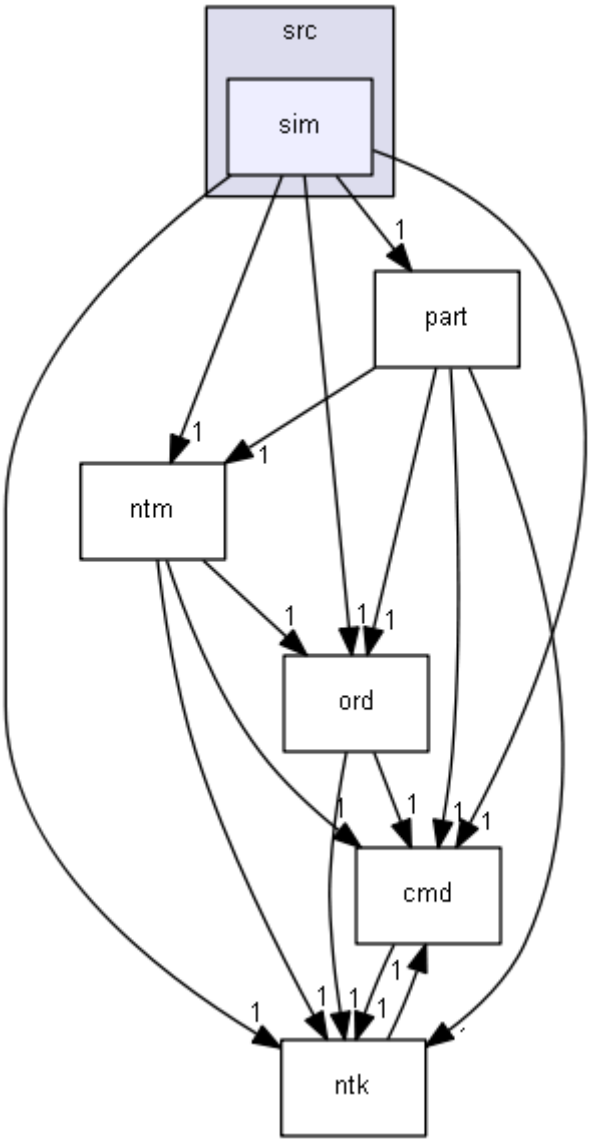

그림 33 sim directory reference

2.32 spfd

This package implements wire removal and replacement based logic optimization for combinational circuits mapped to FPGAs. SPFDs (see the reference below) are used to represent flexibilities in the Boolean network. This package provides two commands: (1) spfd_pilo, a placement independent logic optimization technique. 2. spfd_pdlo: A combined logic and placement optimization technique. spfd_pdlo requires the package VPR.

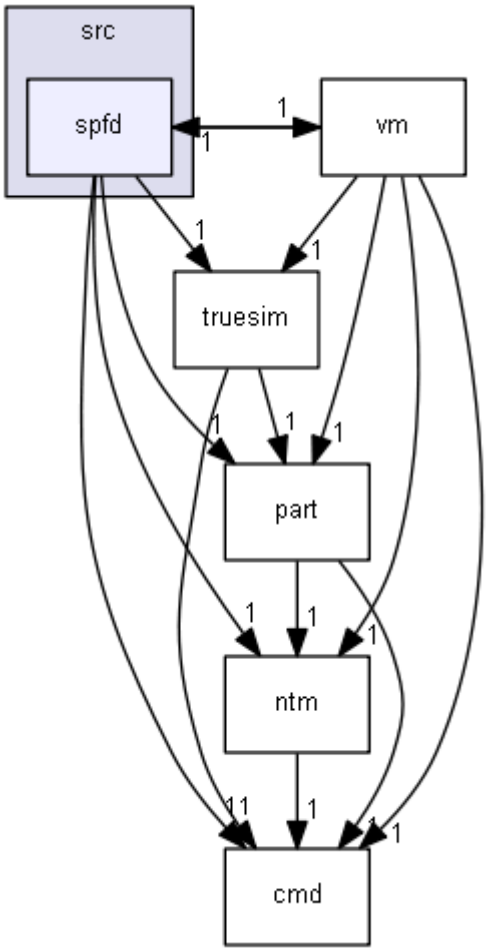

그림 34 spfd directory reference

	Synopsis
spfd.h	SPFD-based wire removal and replacement algorithm for logic optimization of combinational circuits mapped to FPGAs
spfdAPI.c	Routines to read, delete and update data structures used in the package
spfdClean.c	Routines to free memory when not needed
spfdCmd.c	Interface functions for commands spfd_pilo and spfd_pdlo
spfdCommon.c	Essential routines required during SPFD computation
spfdInt.h	Internal data structures for the spfd package
spfdOpt.c	Routines that implement spfd_pilo
spfdProg.c	Routines that perform reprogramming of LUTs (nodes) in the circuit
spfdReg.c	Routines to compute a cluster of nodes for optimization
spfdSpfd.c	Routines to perform SPFD computation
spfdUtil.c	Utility routines for the spfd package

2.33 tbl

External procedures included in this module: Synth_Init(), Synth_End()

Static procedures included in this module: CommandSynthesizeNetwork(), TimeOutHandle

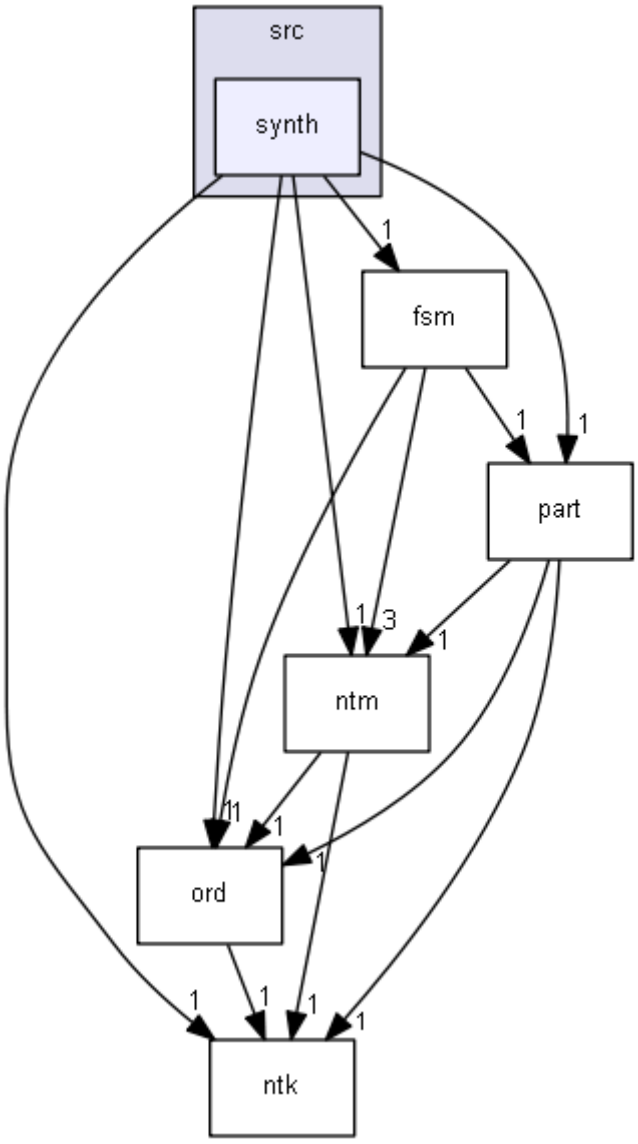

그림 35 synth directory reference

	Synopsis
synth.h	Symbolic synthesis package
synthDiv.c	Divisor functions
synthFactor.c	Multilevel optimization functions
synthGen.c	Generic multilevel factorization method
synthInt.h	Internal declarations
synthOpt.c	Multilevel optimization functions
synthSimple.c	Simple factorization method
synthSynth.c	Synthesis Algorithms
synthUtil.c	Functions to get or to print some information
synthWrite.c	Functions to write blif file and equation file
synth.c	Commands for synthesize_network
synthCount.c	Literal counting functions

2.34 tbl

A Tbl_Table_t is a data structure that contains all the information found in the blif_mv tables (refer to blif_mv document), and represents a multivalued function or group of functions. This can be thought of as a table with inputs and outputs, and the entries in this table describe how the inputs are related to the outputs. The entries in a table may be of two types: they are either a list of ranges that contain values that an entry may take, or are set equal to some other entry in the table. Notice that blif_mv files also have the complement construct, which is absent in the table struct. There are functions for complementing, and canonicalizing a list of ranges

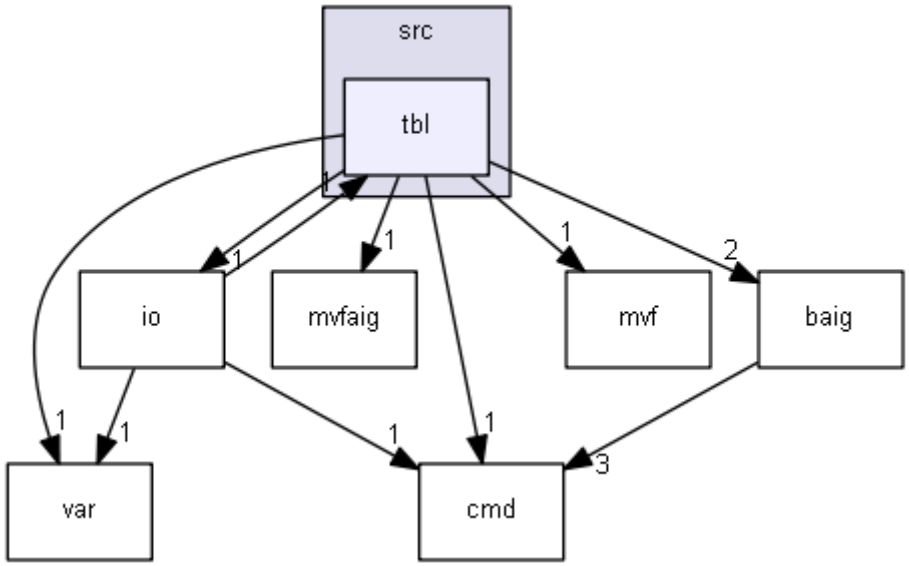

그림 36 tbl directory reference

	Synopsis
tbl.h	Routines for manipulating a multi-valued relation representation
tblAigEntryUtil.c	
tblAigUtil.c	
tblEntryUtil.c	This package describes functions used to manipulate the Tbl_Entry_t ,Tbl_Row_t and Tbl_Range_t structs
tblIdentity.c	Functions used to detect special types of table. Used, for instance, in network sweeping
tblInt.h	Include the internals of the table package
tblSweep.c	Functions to support network sweeping
tblTest.c	This package is used to manipulate the table data structure
tblUtil.c	This package is used to manipulate the table data structure

2.35 truesim

Exported functions and data structures for the truesim package. This package provides procedures to perform pattern based zero delay and levelized two-pass event-driven simulation of circuits described in BLIF format. Pattern vectors to be simulated can either be provided or can be generated using user-specified primary input probabilities. Only combinational circuits are supported at this time

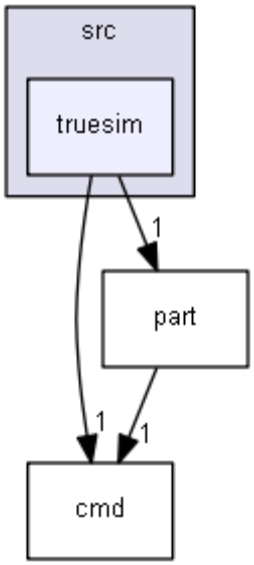

그림 37 truesim directory reference

	Synopsis
truesim.h	Exported functions and data structures for the truesim package
truesimCmd.c	Command interface for the truesim package
truesimInt.h	Internal data structures for the truesim package
truesimMain.c	Routines to perform full delay simulation
truesimSim.c	Top-level routine to perform simulation
truesimUtil.c	Utility functions for the truesim package
truesimZero.c	Routines to perform zero-delay vector simulation

2.36 tst

	Synopsis
tst.h	Test package illustrating VIS conventions
tst.c	Test package initialization, ending, and the command test
tstInt.h	Internal declarations

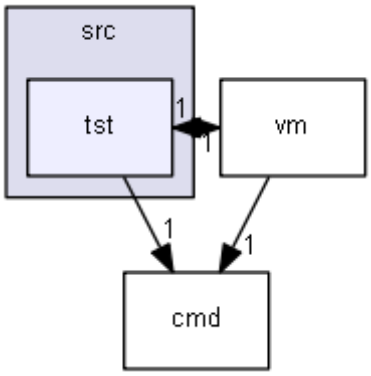

그림 38 tst directory reference

2.37 var

This package contains the data structure for multi-valued variables. For each variable in a BLIF-MV network, this structure is created. The information contained in this structure for a variable is the name, the type (PI,PO,PS,NS etc), the range size, the list of symbolic value names if any, and the encoding of a variable. Any variable can be categorized either to an enumerative variable or a symbolic variable. Enumerative variables are variables which take values from [0,...,n-1], where n is the range of the variable, while symbolic variables are variables which take symbolic values (e.g., [red,blue,green]).

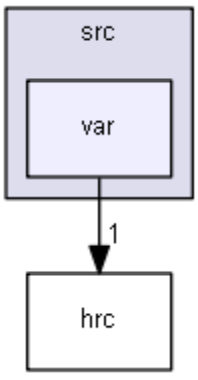

그림 39 var directory reference

	Synopsis
var.h	Multi-valued variables
varInt.h	Internal declarations for the multi-valued variable package
varVariable.c	Routines to access the MV-variable data structure

2.38 vm

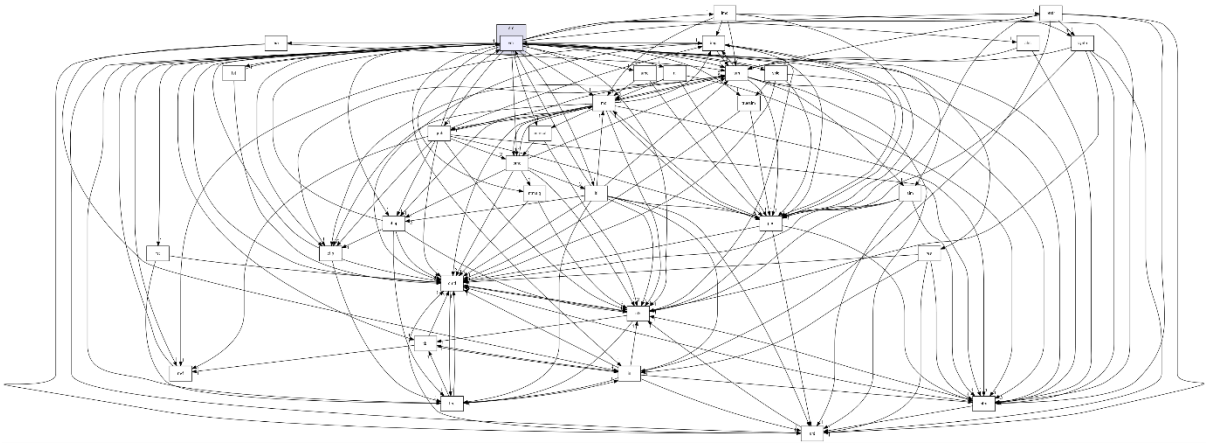

그림 40 vm directory reference

	Synopsis
vm.h	"Main" package of VIS ("vm" = VIS main)
vmInit.c	Initializes and ends VIS
vmInt.h	Internal declarations for the main package
vmMain.c	Main VIS routine. Parses command line at invocation of VIS
vmVers.c	Supplies the compile date and version information