How To Specify and Verify Cache Coherence Protocols: An Elementary Tutorial

Ching-Tsun Chou

Microprocessor Technology Lab

Corporate Technology Group

Intel Corporation

Overview

- Goals:
  - To give a flavor of how cache coherence protocols are specified and verified via a simple but complete example
  - To share experience with some protocol modeling techniques that have proved useful in practice
  - To introduce a simple method of parameterized verification for arbitrary number of nodes
- Example: The German protocol
  - A simple directory-based message-passing cache coherence protocol devised by Steven German in 2000 as a challenge problem
    - German's challenge was fully automatic parameterized verification, which is ***not*** our goal in this talk
- Caveats:
  - This talk is an elementary introduction and contains very little that is new
    - Possible exception: Parameterized verification for arbitrary number of nodes
  - More advanced topics are discussed in Steven German's and Ganesh Gopalakrishnan's tutorials

Agenda

- 1-address abstraction
- Choosing a model checker
- Overview of the German protocol
- Using tables to specify state transitions
- Generation of executable reference models from formal models
- Parameterized verification for arbitrary number of nodes
- Issues not addressed in this tutorial

Agenda

- 1-address abstraction
- Choosing a model checker
- Overview of the German protocol
- Using tables to specify state transitions
- Generation of executable reference models from formal models
- Parameterized verification for arbitrary number of nodes
- Issues not addressed in this tutorial

1-address abstraction

- Focus on how a cache coherence protocol handles data belonging to a single, arbitrary address
- Why this can be a good idea:
  - By focusing on 1 address, protocol instances with more nodes (or other parameters such as buffer entries) can be model-checked
    - Often in practice, only 1-address models are tractable by model checking
  - Interactions between addresses can often be modeled by nondeterminism
    - Example: "Request to address A causes cache line of address B to be victimized" can be modeled by "cache line of address B is nondeterministically victimized"
  - In some designs, ordering requirements between addresses is enforced conservatively by processors and not exported to the network
- Why this can be a bad idea:
  - Correctness requirements do exist between addresses (known as memory ordering or consistency models)
  - Modeling by nondeterminism can hide real problems which often manifest themselves as liveness problems
  - More aggressive designs may export memory ordering enforcement to the network
- Bottom line: 1-address abstraction defines the minimum problem that cache coherence verification should address
  - We will use 1-address abstraction in this tutorial

Agenda

- 1-address abstraction
- Choosing a model checker
- Overview of the German protocol
- Using tables to specify state transitions
- Generation of executable reference models from formal models
- Parameterized verification for arbitrary number of nodes
- Issues not addressed in this tutorial

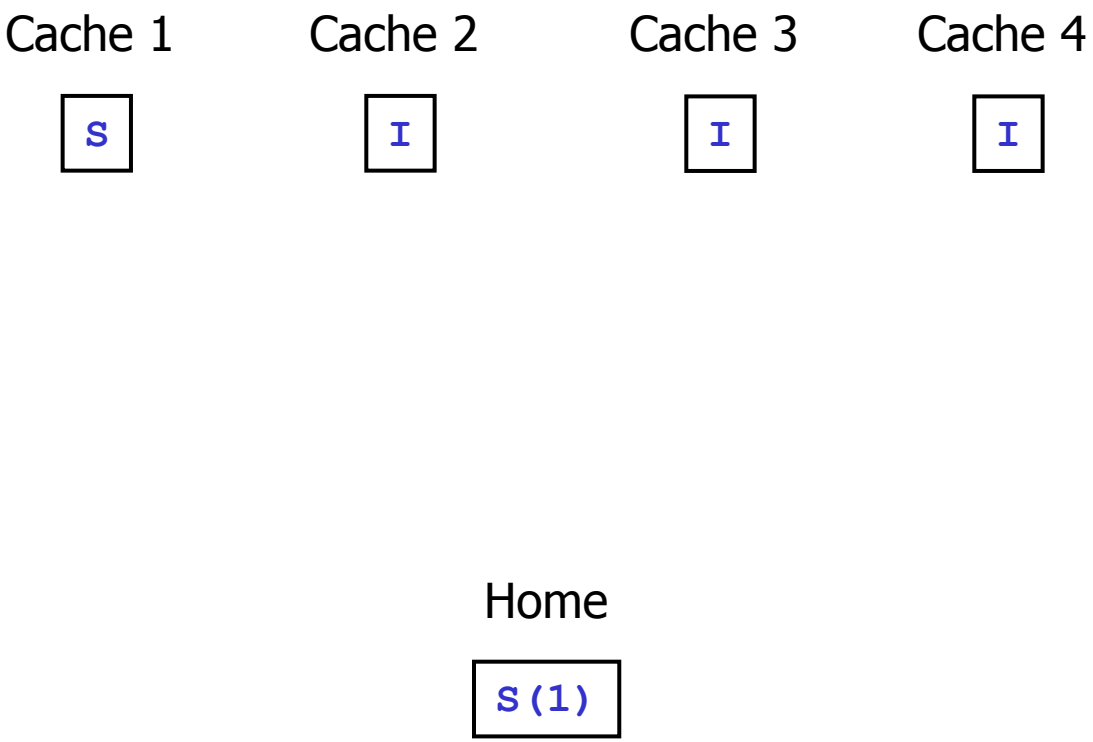
Choosing a model checker

- Explicit state enumeration model checkers
  - Represent unexplored state vectors explicitly and store explored states in hash tables
  - Can use symmetry reduction and disk storage to increase the number of states that can be explored
  - Most widely used model checkers for cache coherence protocols:
    - Murphi (http://verify.stanford.edu/dill/murphi.html)
    - TLC (http://research.microsoft.com/users/lamport/tla/tlc.html)
  - We will use Murphi in this tutorial
- Symbolic model checkers
  - Use symbolic techniques to represent and explore reachable states
    - Ordered binary decision diagrams
    - Boolean satisfiability decision procedures
  - Experience shows that symbolic model checkers are less effective and robust than explicit state enumeration model checkers on cache coherence protocols

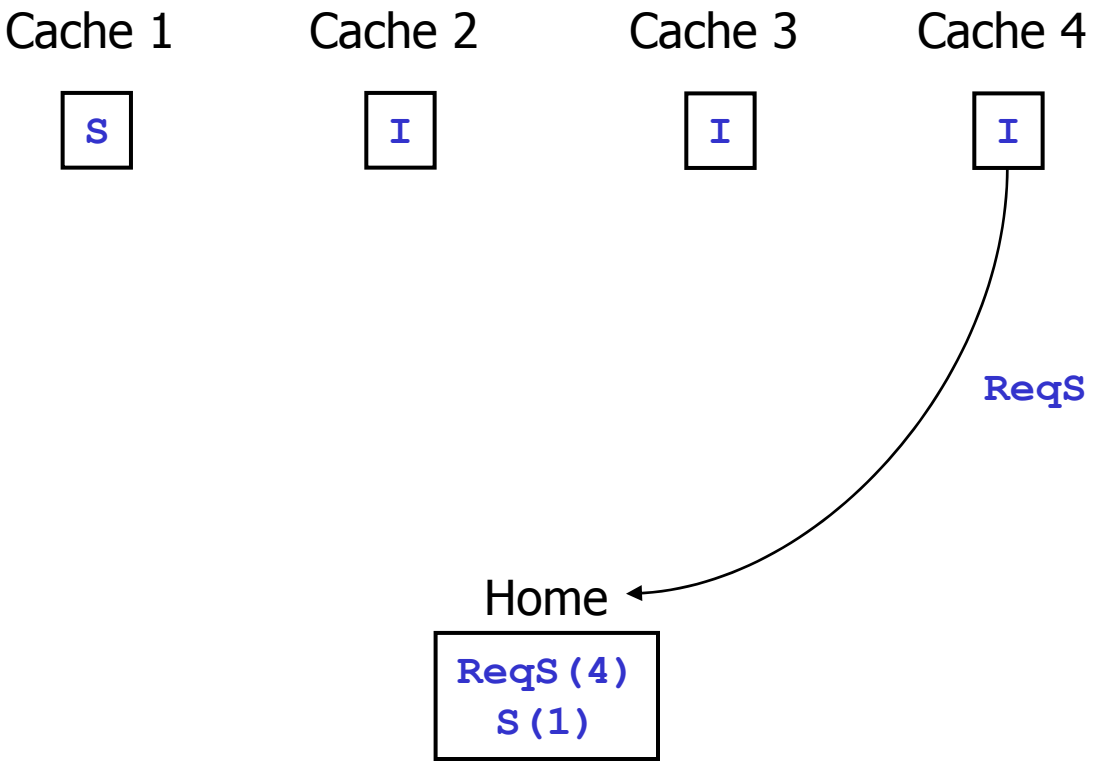
Agenda

- 1-address abstraction
- Choosing a model checker
- Overview of the German protocol
- Using tables to specify state transitions
- Generation of executable reference models from formal models
- Parameterized verification for arbitrary number of nodes
- Issues not addressed in this tutorial

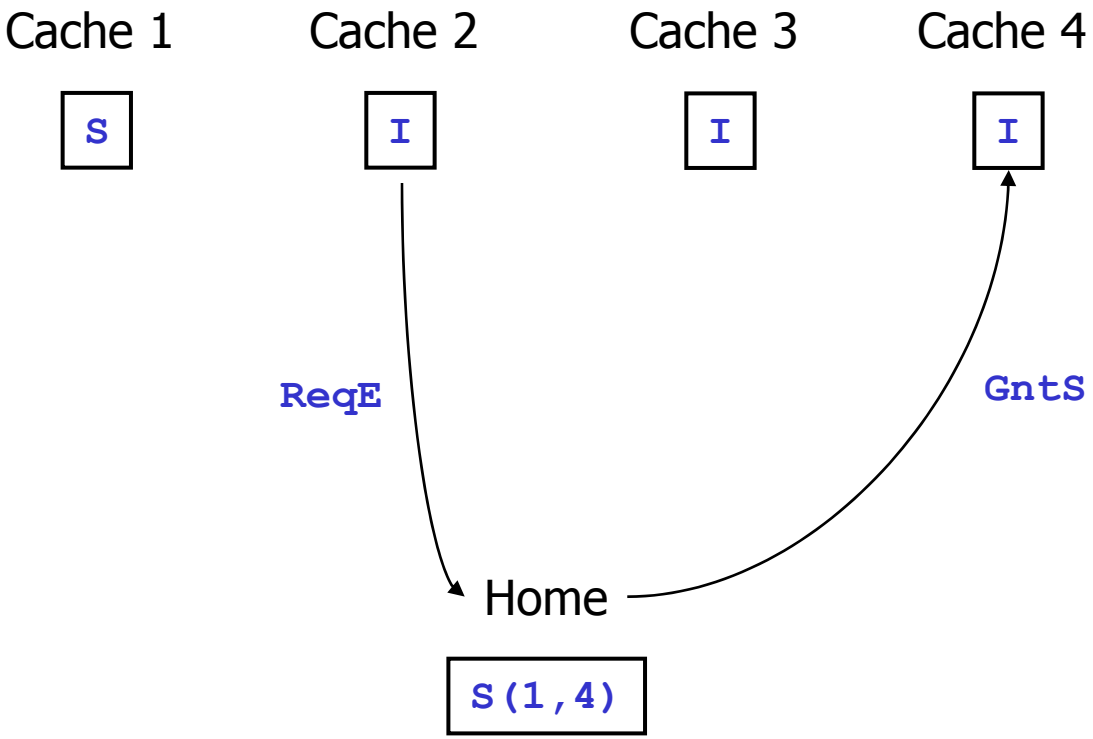
Overview of the German protocol



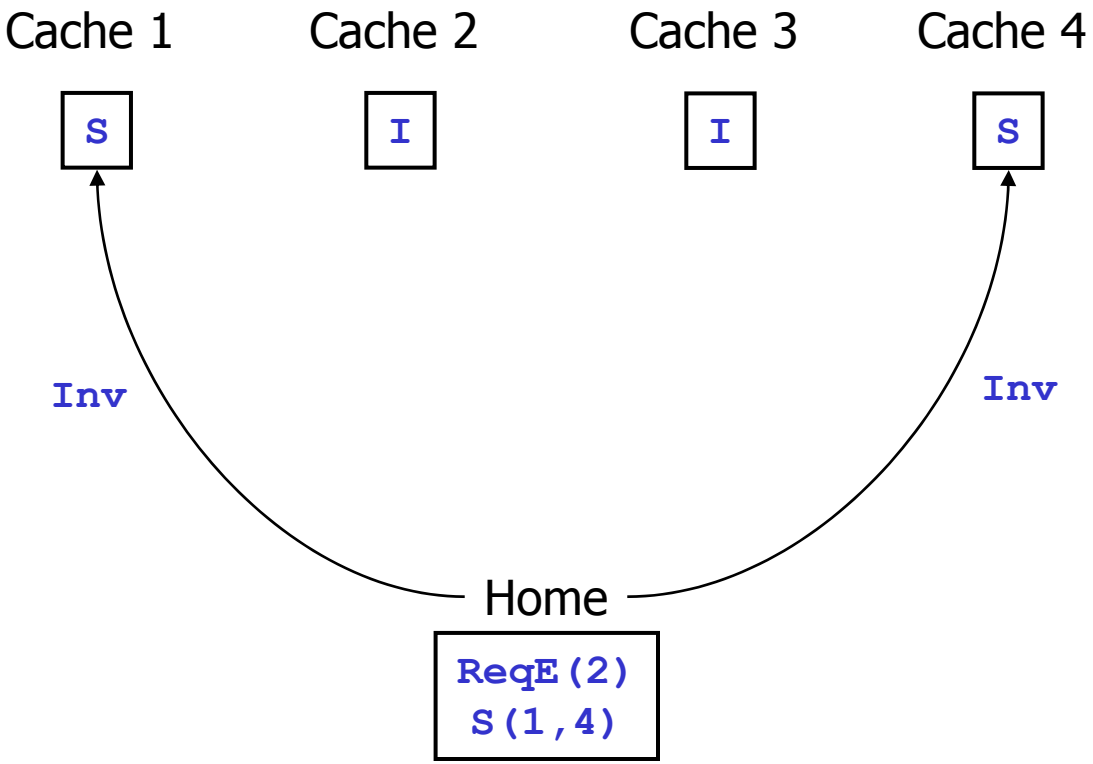
Overview of the German protocol



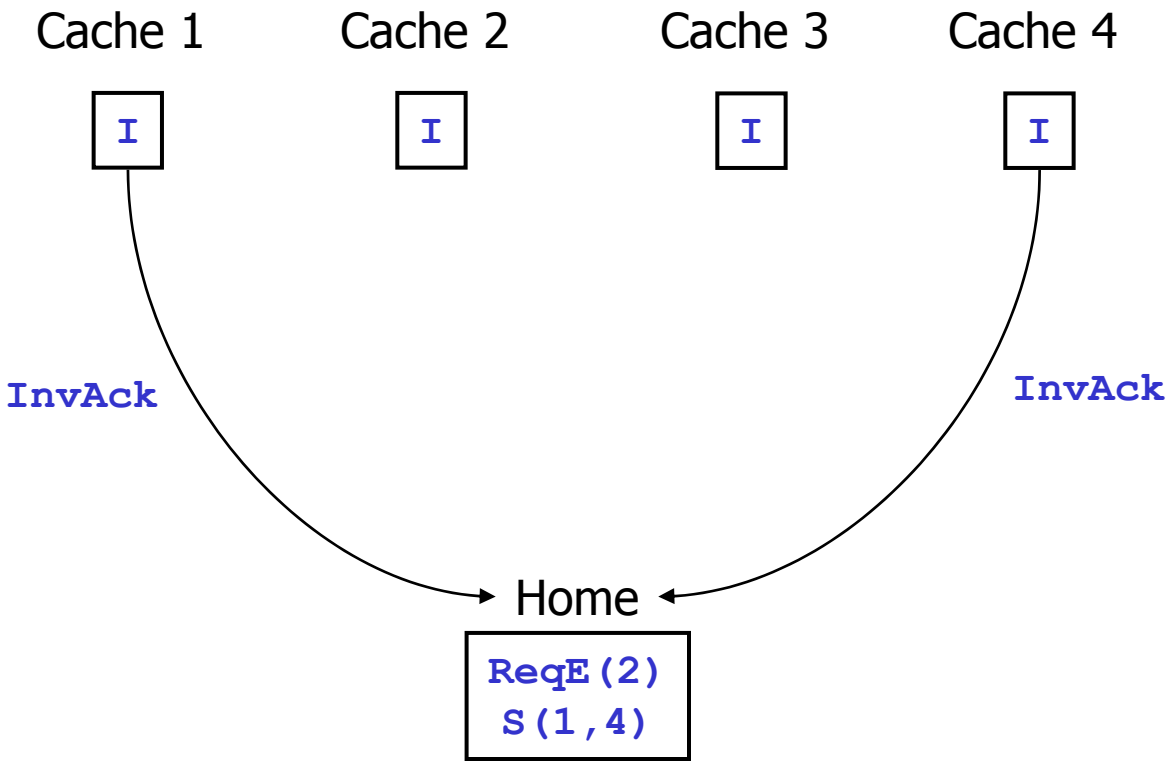
Overview of the German protocol



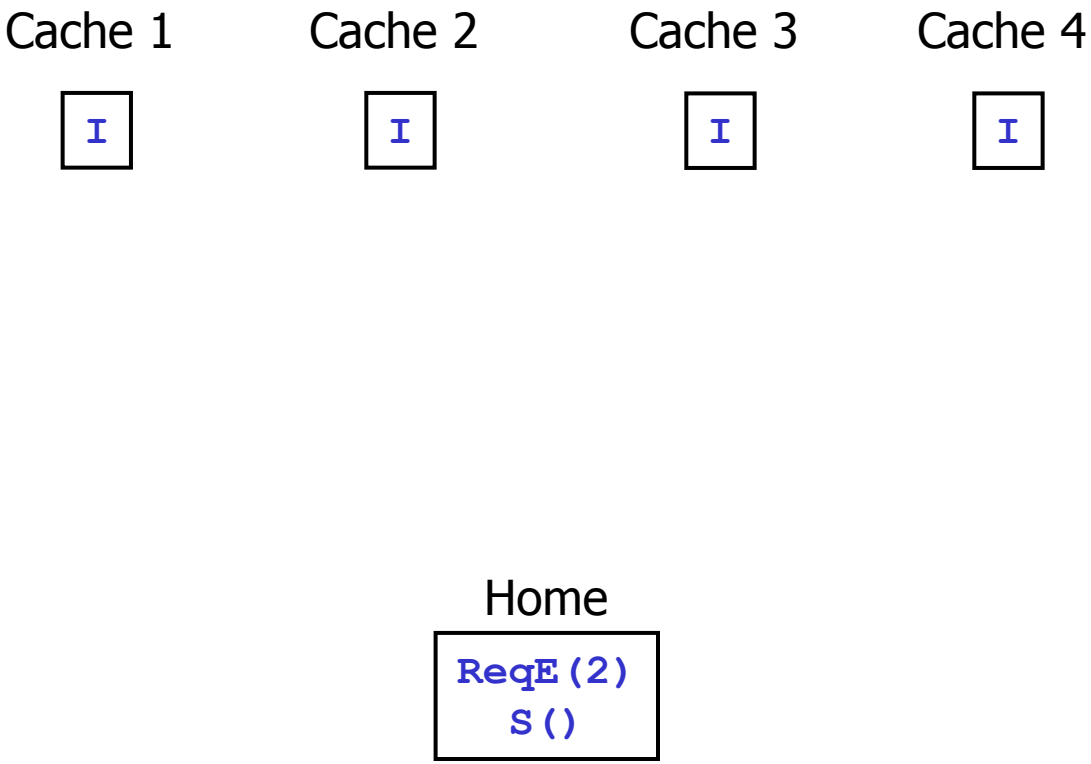
Overview of the German protocol



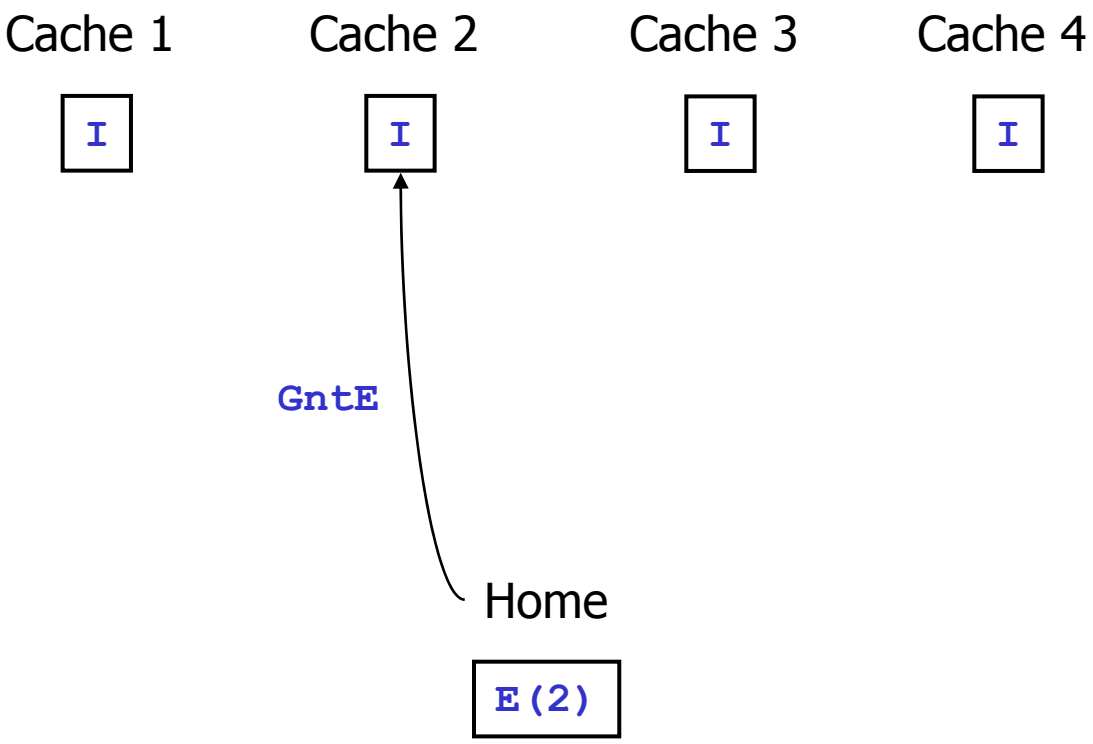
Overview of the German protocol



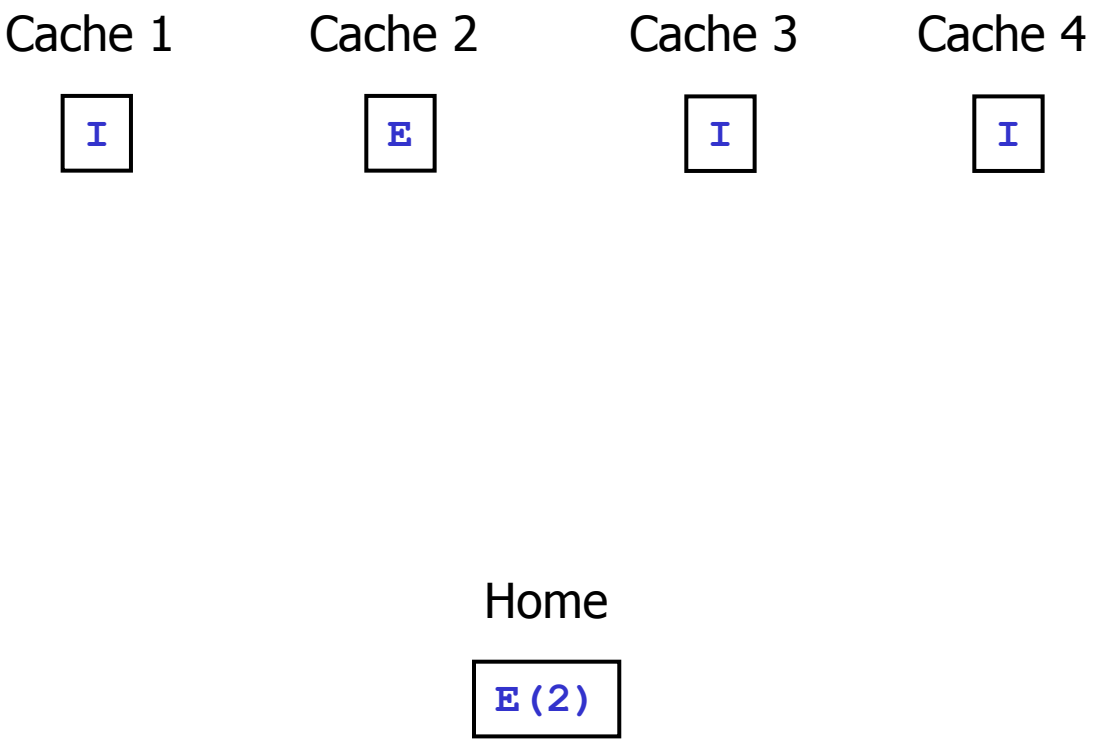
Overview of the German protocol



Overview of the German protocol



Overview of the German protocol



Data structures of the German protocol

```
const  ---- Configuration parameters ----

  NODE_NUM : 5;
  DATA_NUM : 2;

type  ---- Type declarations ----

  NODE : scalarset(NODE_NUM);
  DATA : scalarset(DATA_NUM);

  CACHE_STATE : enum {Invld, Shrd, Excl};
  CACHE : record State : CACHE_STATE; Data : DATA; end;

  MSG_CMD : enum {Empty, ReqS, ReqE, Inv, InvAck, GntS, GntE};
  MSG : record Cmd : MSG_CMD; Data : DATA; end;

  STATE : record
    Cache : array [NODE] of CACHE;      -- Caches
    Chan1 : array [NODE] of MSG;        -- Channels for Req*
    Chan2 : array [NODE] of MSG;        -- Channels for Gnt* and Inv
    Chan3 : array [NODE] of MSG;        -- Channels for InvAck
    InvSet : array [NODE] of boolean;   -- Set of nodes to be invalidated
    ShrSet : array [NODE] of boolean;   -- Set of nodes having valid copies
    ExGntd : boolean;                   -- Excl copy has been granted
    CurCmd : MSG_CMD;                   -- Current request command
    CurPtr : NODE;                      -- Current request node
    MemData : DATA;                     -- Memory data
    AuxData : DATA;                     -- Auxiliary variable for latest data
  end;
```

Data structures of the German protocol

The original German protocol and all its incarnations in the research literature have no data path

```
const  ---- Configuration parameters ----

  NODE_NUM : 5;
  DATA_NUM : 2;

type  ---- Type declarations ----

  NODE : scalarset(NODE_NUM);
  DATA : scalarset(DATA_NUM);

  CACHE_STATE : enum {Invld, Shrd, Excl};
  CACHE : record State : CACHE_STATE; Data : DATA; end;

  MSG_CMD : enum {Empty, ReqS, ReqE, Inv, InvAck, GntS, GntE};
  MSG : record Cmd : MSG_CMD; Data : DATA; end;

  STATE : record
    Cache : array [NODE] of CACHE;      -- Caches
    Chan1 : array [NODE] of MSG;        -- Channels for Req*
    Chan2 : array [NODE] of MSG;        -- Channels for Gnt* and Inv
    Chan3 : array [NODE] of MSG;        -- Channels for InvAck
    InvSet : array [NODE] of boolean;   -- Set of nodes to be invalidated
    ShrSet : array [NODE] of boolean;   -- Set of nodes having valid copies
    ExGntd : boolean;                   -- Excl copy has been granted
    CurCmd : MSG_CMD;                   -- Current request command
    CurPtr : NODE;                      -- Current request node
    MemData : DATA;                     -- Memory data
    AuxData : DATA;                     -- Auxiliary variable for latest data
  end;
```

Initial states of the German protocol

```
var   ---- State variables ----

  Sta : STATE;

---- Initial states ----

ruleset d : DATA do
startstate "Init"
  undefine Sta;
  for i : NODE do
    Sta.Cache[i].State := Invld;
    Sta.Chan1[i].Cmd := Empty;
    Sta.Chan2[i].Cmd := Empty;
    Sta.Chan3[i].Cmd := Empty;
    Sta.InvSet[i] := FALSE;
    Sta.ShrSet[i] := FALSE;
  end;
  Sta.ExGntd := FALSE;
  Sta.CurCmd := Empty;
  Sta.MemData := d;
  Sta.AuxData := d;
end; end;
```

Desired properties of the German protocol

```
---- Invariant properties ----

invariant "CtrlProp"
  forall i : NODE do forall j : NODE do
    i != j ->
    (Sta.Cache[i].State = Excl -> Sta.Cache[j].State = Invld) &
    (Sta.Cache[i].State = Shrd -> Sta.Cache[j].State = Invld |
                                  Sta.Cache[j].State = Shrd)
  end end;

invariant "DataProp"
  (Sta.ExGntd = FALSE -> Sta.MemData = Sta.AuxData) &
  forall i : NODE do
    Sta.Cache[i].State != Invld -> Sta.Cache[i].Data = Sta.AuxData
  end;
```

Agenda

- 1-address abstraction
- Choosing a model checker
- Overview of the German protocol
- Using tables to specify state transitions
- Generation of executable reference models from formal models
- Parameterized verification for arbitrary number of nodes
- Issues not addressed in this tutorial

Using tables to specify state transitions

- Cache node actions:

Action			Current State				Next State							
Cmd	Node	Data	Cache[i]	Chan1[i]	Chan2[i]	Chan3[i]	Cache[i]		Chan1[i]	Chan2[i]		Chan3[i]		AuxData
			State	Cmd	Cmd	Cmd	State	Data	Cmd	Cmd	Data	Cmd	Data	
Store	i	d	Excl					d						d
SendReqS	i		Invld	Empty					ReqS					
SendReqE	i		!= Excl	Empty					ReqE					
RecvInvS	i		!= Excl		Inv	Empty	Invld	Undef		Empty		InvAck		
RecvInvE	i		Excl		Inv	Empty	Invld	Undef		Empty		InvAck	Cache[i].Data	
RecvGntS	i				GntS		Shrd	Chan2[i].Data		Empty	Undef			
RecvGntE	i				GntE		Excl	Chan2[i].Data		Empty	Undef			

- Home node actions:

Action		Current State								Next State										
Cmd	Node	CurCmd	CurPtr	InvSet	ShrSet	ExGntd	Chan1[i]	Chan2[i]	Chan3[i]	CurCmd	CurPtr	InvSet	ShrSet	ExGntd	MemData	Chan1[i]	Chan2[i]		Chan3[i]	
							Cmd	Cmd	Cmd							Cmd	Cmd	Data	Cmd	Data
RecvReqS	i	Empty					ReqS			ReqS	i	ShrSet				Empty				
RecvReqE	i	Empty					ReqE			ReqE	i	ShrSet				Empty				
SendInvReqS	i	ReqS		>= {i}		TRUE		Empty				\ {i}					Inv			
SendInvReqE	i	ReqE		>= {i}				Empty				\ {i}					Inv			
RecvInvAckS	i	!= Empty				FALSE			InvAck				\ {i}						Empty	
RecvInvAckE	i	!= Empty				TRUE			InvAck				\ {i}	FALSE	Chan3[i].Data				Empty	Undef
SendGntS	i	ReqS	i			FALSE		Empty		Empty	Undef		+ {i}				GntS	MemData		
SendGntE	i	ReqE	i		= {}	FALSE		Empty		Empty	Undef		+ {i}	TRUE			GntE	MemData		

Assigning semantics to tables

Action			Current State				Next State							
Cmd	Node	Data	Cache[i]	Chan1[i]	Chan2[i]	Chan3[i]	Cache[i]		Chan1[i]	Chan2[i]		Chan3[i]		AuxData
			State	Cmd	Cmd	Cmd	State	Data	Cmd	Cmd	Data	Cmd	Data	
Store	i	d	Excl					d						d
SendReqS	i		Invld	Empty					ReqS					
SendReqE	i		!= Excl	Empty					ReqE					
RecvInvS	i		!= Excl		Inv	Empty	Invld	Undef		Empty		InvAck		
RecvInvE	i		Excl		Inv	Empty	Invld	Undef		Empty		InvAck	Cache[i].Data	
RecvGntS	i				GntS		Shrd	Chan2[i].Data		Empty	Undef			
RecvGntE	i				GntE		Excl	Chan2[i].Data		Empty	Undef			

```
COLUMN |Current State|Cache[i]|State|
    |Invld| => Sta.Cache[i].State = Invld
    |Excl| => Sta.Cache[i].State = Excl
    |!= Excl| => Sta.Cache[i].State != Excl

  COLUMN |Current State|Chan1[i]|Cmd|
    |Empty| => Sta.Chan1[i].Cmd = Empty

  COLUMN |Current State|Chan2[i]|Cmd|
    |Inv| => Sta.Chan2[i].Cmd = Inv
    |GntS| => Sta.Chan2[i].Cmd = GntS
    |GntE| => Sta.Chan2[i].Cmd = GntE

  COLUMN |Current State|Chan3[i]|Cmd|
    |Empty| => Sta.Chan3[i].Cmd = Empty

  COLUMN |Next State|Cache[i]|State|
    |Invld| => NxtSta.Cache[i].State := Invld
    |Shrd| => NxtSta.Cache[i].State := Shrd
    |Excl| => NxtSta.Cache[i].State := Excl
```

```
COLUMN |Next State|Cache[i]|Data|
  |d| => NxtSta.Cache[i].Data := d
  |Chan2[i].Data| => NxtSta.Cache[i].Data :=
                     Chan2[i].Data
  |Undef| => undefine NxtSta.Cache[i].Data

COLUMN |Next State|Chan1[i]|Cmd|
  |ReqS| => NxtSta.Chan1[i].Cmd := ReqS
  |ReqE| => NxtSta.Chan1[i].Cmd := ReqE

COLUMN |Next State|Chan2[i]|Cmd|
  |Empty| => NxtSta.Chan2[i].Cmd := Empty

COLUMN |Next State|Chan2[i]|Data|
  |Undef| => undef NxtSta.Chan2[i].Data

COLUMN |Next State|Chan3[i]|Cmd|
  |InvAck| => NxtSta.Chan3[i].Cmd := InvAck

COLUMN |Next State|Chan3[i]|Data|
  |Cache[i].Data| => NxtSta.Chan3[i].Data :=
                    Cache[i].Data

COLUMN |Next State|AuxData|
  |d| => NxtSta.AuxData := d
```

An example action

Action			Current State				Next State							
Cmd	Node	Data	Cache[i]	Chan1[i]	Chan2[i]	Chan3[i]	Cache[i]		Chan1[i]	Chan2[i]		Chan3[i]		AuxData
			State	Cmd	Cmd	Cmd	State	Data	Cmd	Cmd	Data	Cmd	Data	
Store	i	d	Excl					d						d
SendReqS	i		Invld	Empty					ReqS					
SendReqE	i		!= Excl	Empty					ReqE					
RecvInvS	i		!= Excl		Inv	Empty	Invld	Undef		Empty		InvAck		
RecvInvE	i		Excl		Inv	Empty	Invld	Undef		Empty		InvAck	Cache[i].Data	
RecvGntS	i				GntS		Shrd	Chan2[i].Data		Empty	Undef			
RecvGntE	i				GntE		Excl	Chan2[i].Data		Empty	Undef			

```
ruleset i : NODE do
rule "RecvInvE"
  Sta.Cache[i].State = Excl &
  Sta.Chan2[i].Cmd = Inv &
  Sta.Chan3[i].Cmd = Empty
==>
var NxtSta : STATE;
begin
  NxtSta := Sta;
--
  NxtSta.Cache[i].State := Invld;
  undefine NxtSta.Cache[i].Data;
  NxtSta.Chan2[i].Cmd := Empty;
  NxtSta.Chan3[i].Cmd := InvAck;
  NxtSta.Chan3[i].Data := Sta.Cache[i].Data;
--
  Sta := NxtSta;
end; end;
```

By having a separate NxtSta, the order of assignments does not matter any more

Advantages of table-based specification

- Tables provide an abstract summary
  - Once one becomes familiar with what table entries mean, one can work almost exclusively at the table level of abstraction
- Table format is flexible
  - There is no restriction on what texts can appear in table entries and what code fragments can be assigned to table entries
  - Lots of room for experimentation
- Experience shows that even complex protocols can typically be summarized using a small number of tables printable on a few pages
  - It is much easier to comprehend and reason about a protocol by staring at a few pages of descriptions than by wading thru 1000's of lines of code
  - Regularities among actions can be more easily observed in tables than in code
- Tables are widely used in industry
  - Example: Eiriksson & McMillan (CAV 1995)

FLASH in tables (1/2)

- The Murphi description of FLASH contains >1000 lines of code

Action		Current State						Next State											
Name	Parameters	Cch(p)			Dir			Cch(p)				Dir					Mem	Forall	Net(p, q)
	p	State	Wait	InvM	Pend	Dirty	Pres	State	Data	Wait	InvM	Pend	Dirty	Local	Pres	InvCnt	Data	q	
PI.Remote.Get	!= Home	I	None							Get								= Home	Get
PI.Local.Get.Else	= Home	I	None		False	True				Get		True						= Dir.Pres	Get
PI.Local.Get.Put	= Home	I	None	False	False	False		S	Mem.Data	None	False			True					
				True				I											
PI.Remote.GetX	!= Home	I	None							GetX								= Home	GetX
PI.Local.GetX.Else	= Home	I, S	None		False	True				GetX		True						= Dir.Pres	GetX
PI.Local.GetX.PutX	= Home	I, S	None		False	False	= {}	E	Mem.Data	None	False		True	True					
							> {}					True			{}	#(Dir.Pres)		in Dir.Pres	Inv
PI.Remote.PutX	!= Home	E	None					I										= Home	Wb(Cch(p).Data)
PI.Local.PutX	= Home	E	None		True			I					False				Cch(p).Data		
					False									False					
PI.Remote.Replace	!= Home	S	None					I										= Home	Replace
PI.Local.Replace	= Home	S	None					I						False					

FLASH in tables (2/2)

Action			Current State									Next State									
Name	Parameters		Net(p, q)	Cch(q)			Dir					Cch(q)				Dir					Mem
	p	q		State	Wait	InvM	Pend	Dirty	Local	Pres	InvCnt	State	Data	Wait	InvM	Pend	Dirty	Local	Pres	InvCnt	Data
NI.Nak			Nak											None	False						
NI.Nakc			Nakc													False					
NI.Local.Get.Else	!= Home	= Home	Get, no Replace				True														
				I, S			False	True	True												
									False	= {p}											
										!= {p}						True					
NI.Local.Get.Put	!= Home	= Home	Get, no Replace			False	False	False											+ {p}		
NI.Local.Get.Put.Ex	!= Home	= Home	Get, no Replace	E		False	False	True	True			S					False		+ {p}		Cch(q).Data
NI.Local.Get.Put.Inv	!= Home	= Home	Get, no Replace			True	False	False											+ {p}		
NI.Local.Get.Put.Ex.Inv	!= Home	= Home	Get, no Replace	E		True	False	True	True			S					False		+ {p}		Cch(q).Data
NI.Remote.Get.Else	!= Home	!= Home	Get	I, S																	
NI.Remote.Get.Put	= Home	!= Home	Get	E		False						S									
	!= Home																				
NI.Remote.Get.Put.Inv	= Home	!= Home	Get	E		True						S									
	!= Home																				
NI.Local.GetX.Else	!= Home	= Home	GetX				True														
				I, S			False	True	True												
									False	= {p}											
										!= {p}						True					
NI.Local.GetX.PutX	!= Home	= Home	GetX		!= Get		False	False		<= {p}		I					True	False	{p}		
					= Get										True						
					!= Get					> {p}						True				#(Dir.Pres - {p})	
					= Get										True						
NI.Local.GetX.PutX.Ex	!= Home	= Home	GetX	E			False	True	True			I					True	False	{p}		
NI.Remote.GetX.Else		!= Home	GetX	I, S																	
NI.Remote.GetX.PutX	= Home	!= Home	GetX	E								I									
	!= Home																				
NI.Local.Put		= Home	Put			False						S	Put.Data	None		False	False	True			Put.Data
						True						I			False						
NI.Remote.Put		!= Home	Put			False						S	Put.Data	None							
						True						I			False						
NI.Local.PutXAcksDone		= Home	PutX									E	PutX.Data	None	False	False		True	- {p}		
NI.Remote.PutX		!= Home	PutX									E	PutX.Data	None	False						
NI.Inv		!= Home	Inv		!= Get							I									
					= Get										True						
NI.InvAck	!= Home		InvAck								> 1									- 1	
											= 1					False		& Dirty			
NI.Wb			Wb														False		{}		Wb.Data
NI.Fack			Fack													False			{p}		
NI.ShWb			ShWb													False	False		+ {p}		ShWb.Data
NI.Replace			Replace																- {p}		

Agenda

- 1-address abstraction
- Choosing a model checker
- Overview of the German protocol
- Using tables to specify state transitions
- Generation of executable reference models from formal models
- Parameterized verification for arbitrary number of nodes
- Issues not addressed in this tutorial

Connecting formal model to RTL

- In the end, what is implemented in silicon is the RTL, not the formal model
- How do we know that the RTL implements the formal model?
  - A very large and deep research problem
- A partial solution: Use the formal model to check RTL simulation
- Ingredients:
  1. A version of the formal model, called an executable reference model (ERM), that can be used as a checker in RTL simulation
  2. A program that monitors RTL simulation and extracts protocol actions to drive ERM
- Experience shows that this is an effective way to leverage results from formal protocol verification in RTL verification
  - Once automated, the cost of generating ERM is zero
  - ERM has been verified by FV, at least for small configurations

Generation of ERMs from formal models

- Restrictions on the formal model:
  - No hidden nondeterminism
    - All nondeterministic choices are made thru the selection of actions and their parameters (in Murphi jargon: rulesets and their parameters)
  - Simple data structures
    - Enumerated types
    - Finite integer ranges
    - Records of previous defined types
    - Arrays over finite integer ranges of previous defined types
- Main API of ERM:

```
VOID Step(INPUT *Inp, OUTPUT *Out, STATE *Sta, STATE *NxtSta);
```

  - Attempts to execute an action (with parameters) specified by *Inp from the state *Sta
  - If the action is enabled, then *NxtSta is the next state
  - If the action is not enabled, then something is wrong
    - Since the action is extracted from RTL simulation, its being disabled in ERM means that RTL and ERM have diverged

Agenda

- 1-address abstraction
- Choosing a model checker
- Overview of the German protocol
- Using tables to specify state transitions
- Generation of executable reference models from formal models
- Parameterized verification for arbitrary number of nodes
- Issues not addressed in this tutorial

Parameterized verification for arbitrary # of nodes

- Typical industrial cache coherence protocols can be model-checked for at most 3~4 (cache) nodes, but they may be deployed in systems with many more nodes
  - Protocol designers often have intuitions about why 3~4 nodes suffice to exhibit all "interesting" scenarios, but such intuitions are typically not formalized
- So how do we know they will continue to work in large systems? Can there be unanticipated error scenarios that take more than 3~4 nodes to manifest themselves?
- Parameterized verification seeks to formally verify the protocol for arbitrary # of nodes
- The method presented below is an alternative formulation of McMillan's compositional model checking method (CHARME 2001)
  - We explain our method by applying it to the German protocol
  - For theoretical justification of the apparently circular reasoning, please see our paper in FMCAD 2004

Basic idea

- Choose any 2 nodes, which WLOG can be taken to be nodes n1 and n2 (because all nodes are symmetric w.r.t. each other)
  - Why 2? The intuitive reason is that any basic interaction in German involves at most 2 nodes: {requesting node, invalidated node}
    - Note that the home node is not indexed by NODE and always included
    - The technical reason is that all quantifications (properties, lemmas, rulesets, etc) in German are nested at most 2 deep
- Our goal is to construct an abstraction, called AbsGerman, that contains the home node, n1, n2, and a fictitious node "Other" representing all other nodes, such that:
  - AbsGerman permits all possible behaviors in German that n1, n2, and the home node can exhibit (including what "Other" can do to them)
  - The behaviors of AbsGerman are sufficiently constrained that they satisfy the desired properties CtrlProp and DataProp
- If successful, any safety property satisfied by AbsGerman should be satisfied by German as well

General strategy

- Start with an AbsGerman that is obviously more permissive than German
- Counterexample-guided discovery of *noninterference lemmas*:
  1. Try to prove the desired properties and all noninterference lemmas discovered so far on the current AbsGerman by model checking
  2. If all properties pass, we are done and all properties and lemmas are true in AbsGerman and hence in German as well
  3. Otherwise, analyze the counterexample to identify an offending action and formulate a new noninterference lemma to "fix" it
     - This step requires human ingenuity and understanding of the protocol
  4. Instantiate the new noninterference lemma to strengthen the precondition of the offending action in the abstract model
  5. Go back to step 1.

Data structures of AbsGerman

```
const  ---- Configuration parameters ----

  NODE_NUM : 2;
  DATA_NUM : 2;

type  ---- Type declarations ----

  NODE : scalarset(NODE_NUM);

  ABS_NODE : union {NODE, enum{Other}};

-- … some type declarations omitted

  STATE : record
    Cache : array [NODE] of CACHE;      -- Caches
    Chan1 : array [NODE] of MSG;        -- Channels for Req*
    Chan2 : array [NODE] of MSG;        -- Channels for Gnt* and Inv
    Chan3 : array [NODE] of MSG;        -- Channels for InvAck
    InvSet : array [NODE] of boolean;   -- Set of nodes to be invalidated
    ShrSet : array [NODE] of boolean;   -- Set of nodes having valid copies
    ExGntd : boolean;                   -- Excl copy has been granted
    CurCmd : MSG_CMD;                   -- Current request command
    CurPtr : ABS_NODE;                  -- Current request node
    MemData : DATA;                     -- Memory data
    AuxData : DATA;                     -- Auxiliary variable for latest data
  end;
```

Action abstraction when i ∈ NODE

- Concrete action:

```
ruleset i : NODE; d : DATA do
rule "Store"
  Sta.Cache[i].State = Excl
==>
var NxtSta : STATE;
begin
  NxtSta := Sta;
--
  NxtSta.Cache[i].Data := d;
  NxtSta.AuxData := d;
--
  Sta := NxtSta;
```

- Abstract action:

```
ruleset i : NODE; d : DATA do
rule "Store"
  Sta.Cache[i].State = Excl
==>
var NxtSta : STATE;
begin
  NxtSta := Sta;
--
  NxtSta.Cache[i].Data := d;
  NxtSta.AuxData := d;
--
  Sta := NxtSta;
```

Action abstraction when i ∈ NODE

- Concrete action:

```
ruleset i : NODE do
rule "SendGntE"
  Sta.CurCmd = ReqE &
  Sta.CurPtr = i &
  forall j : NODE do
    Sta.ShrSet[j] = FALSE
  end &
  Sta.ExGntd = FALSE &
  Sta.Chan2[i].Cmd = Empty
==>
var NxtSta : STATE;
begin
  NxtSta := Sta;
--
  NxtSta.CurCmd := Empty;
  undefine NxtSta.CurPtr;
  NxtSta.ShrSet[i] := TRUE;
  NxtSta.ExGntd := TRUE;
  NxtSta.Chan2[i].Cmd := GntE;
  NxtSta.Chan2[i].Data := Sta.MemData;
--
  Sta := NxtSta;
end end;
```

NODE = {1, 2, …, N}

- Abstract action:

```
ruleset i : NODE do
rule "SendGntE"
  Sta.CurCmd = ReqE &
  Sta.CurPtr = i &
  forall j : NODE do
    Sta.ShrSet[j] = FALSE
  end &
  Sta.ExGntd = FALSE &
  Sta.Chan2[i].Cmd = Empty
==>
var NxtSta : STATE;
begin
  NxtSta := Sta;
--
  NxtSta.CurCmd := Empty;
  undefine NxtSta.CurPtr;
  NxtSta.ShrSet[i] := TRUE;
  NxtSta.ExGntd := TRUE;
  NxtSta.Chan2[i].Cmd := GntE;
  NxtSta.Chan2[i].Data := Sta.MemData;
--
  Sta := NxtSta;
end end;
```

NODE = {1, 2}

Action abstraction when i ∈ NODE

- Concrete action:

```
ruleset i : NODE do
rule "RecvReqS"
  Sta.CurCmd = Empty &
  Sta.Chan1[i].Cmd = ReqS
==>
var NxtSta : STATE;
begin
  NxtSta := Sta;
--
  NxtSta.CurCmd := ReqS;
  NxtSta.CurPtr := i;
  for j : NODE do
    NxtSta.InvSet[j] :=
        Sta.ShrSet[j]
  end;
  NxtSta.Chan1[i].Cmd := Empty;
--
  Sta := NxtSta;
end end;
```

NODE = {1, 2, …, N}

- Abstract action:

```
ruleset i : NODE do
rule "RecvReqS"
  Sta.CurCmd = Empty &
  Sta.Chan1[i].Cmd = ReqS
==>
var NxtSta : STATE;
begin
  NxtSta := Sta;
--
  NxtSta.CurCmd := ReqS;
  NxtSta.CurPtr := i;
  for j : NODE do
    NxtSta.InvSet[j] :=
        Sta.ShrSet[j]
  end;
  NxtSta.Chan1[i].Cmd := Empty;
--
  Sta := NxtSta;
end end;
```

NODE = {1, 2}

Action abstraction when i = Other

- Concrete action:

```
ruleset i : NODE do
rule "SendReqS"
  Sta.Cache[i].State = Invld &
  Sta.Chan1[i].Cmd = Empty
==>
var NxtSta : STATE;
begin
  NxtSta := Sta;
--
  NxtSta.Chan1[i].Cmd := ReqS;
--
  Sta := NxtSta;
end end;
```

- Abstract action:

```
rule "ABS_Stutter" end;
```

Action abstraction when i = Other

- Concrete action:

```
ruleset i : NODE; d : DATA do
rule "Store"
  Sta.Cache[i].State = Excl
==>
var NxtSta : STATE;
begin
  NxtSta := Sta;
--
  NxtSta.Cache[i].Data := d;
  NxtSta.AuxData := d;
--
  Sta := NxtSta;
```

- Abstract action:

```
ruleset d : DATA do
rule "ABS_Store"
  TRUE
==>
var NxtSta : STATE;
begin
  NxtSta := Sta;
--

  NxtSta.AuxData := d;
--
  Sta := NxtSta;
end end;
```

Action abstraction when i = Other

- Concrete action:

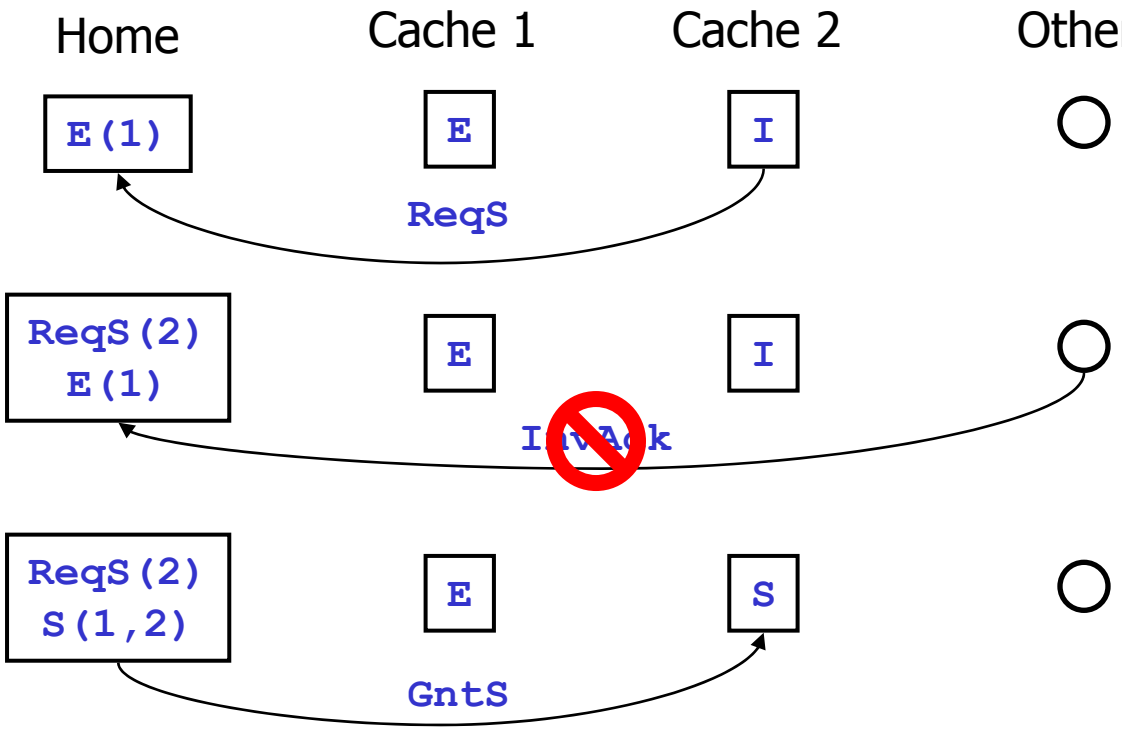
```
ruleset i : NODE do
rule "RecvInvAckE"
  Sta.CurCmd != Empty &
  Sta.ExGntd = TRUE &
  Sta.Chan3[i].Cmd = InvAck
==>
var NxtSta : STATE;
begin
  NxtSta := Sta;
--
  NxtSta.ShrSet[i] := FALSE;
  NxtSta.ExGntd := FALSE;
  NxtSta.MemData :=
     Sta.Chan3[i].Data;
  NxtSta.Chan3[i].Cmd := Empty;
  undefine NxtSta.Chan3[i].Data;
--
  Sta := NxtSta;
end end;
```

- Abstract action:

```
rule "ABS_RecvInvAckE"
  Sta.CurCmd != Empty &
  Sta.ExGntd = TRUE

==>
var NxtSta : STATE;
begin
  NxtSta := Sta;
--

  NxtSta.ExGntd := FALSE;
  undefine NxtSta.MemData;



--
  Sta := NxtSta;
end;
```

The first counterexample

- We first comment out DataProp and focus on proving CtrlProp
  - Because data consistency depends on the correctness of control logic, but not the other way around
- The first version of AbsGerman produces the following counterexample to CtrlProp:



The first noninterference lemma

- Goal: Outlaw the bogus InvAck from Other
  - Why is this particular InvAck from Other bad?
  - Because there is an Exclusive copy at a node that is not the sender of InvAck
- Noninterference lemma:

```
invariant "Lemma_1"
  forall i : NODE do
    Sta.Chan3[i].Cmd = InvAck & Sta.CurCmd != Empty & Sta.ExGntd = true ->
    forall j : NODE do j != i -> Sta.Cache[j].State != Excl end
  end;
```

- Instantiating the noninterference lemma with i = Other:

```
rule "ABS_RecvInvAckE"
  Sta.CurCmd != Empty & Sta.ExGntd = true &
  forall j : NODE do Sta.Cache[j].State != Excl end
==> …
```

- But how do we justify the noninterference lemma?
  - We prove it in the same abstract model where we have used the lemma to strengthen the precondition of ABS_RecvInvAck!
- Why no circularity? See our FMCAD paper

The rest of the proof

- CtrlProp is proved after 2 more iterations
- DataProp is proved after 4 further iterations
  - There are nontrivial control logic properties that are needed to prove DataProp, but are not needed for CtrlProp
  - Curiously, none of the papers in literature that uses German as an example considered DataProp
- See handout for intermediate steps and the final AbsGerman and noninterference lemmas
- Even FLASH can be proved correct for arbitrary # of nodes in this manner, using only a small set of noninterference lemmas
  - See our FMCAD paper

Agenda

- 1-address abstraction
- Choosing a model checker
- Overview of the German protocol
- Using tables to specify state transitions
- Generation of executable reference models from formal models
- Parameterized verification for arbitrary number of nodes
- Issues not addressed in this tutorial

Issues not addressed in this tutorial

- Liveness
  - Bad things that can happen: Deadlock, livelock, starvation
  - Tricky because:
    - Liveness problems are often caused by low-level implementation artifacts that are hard to model in a high-level model
    - Liveness properties are more expensive to model-check than safety properties and requires fairness assumptions
- Memory ordering
  - Will be addressed in Ganesh's tutorial
  - Involves the interactions between multiple addresses and hence is very expensive to model-check
- Bridging the abstraction gap between formal models and RTL
  - Formal protocol models typically have large atomic actions: an agent receives a message, performs local state updates, and (possibly) sends out more messages in one atomic step
  - RTL has small atomic actions (each protocol action is performed over multiple cycles) and pipelined (multiple protocol actions can be active at the same time in different pipeline stages)
  - Good target for pipeline verification research
    - Protocol engines should be simpler than processor cores
  - Prior work: Eiriksson (FMCAD 1998)