A Java Supercompiler and its Application to Verification of Cache-Coherence Protocols

Andrei Klimov

Keldysh Institute of Applied Mathematics, Russian Academy of Sciences

Co-authors of the JScp system

Arkady Klimov

Artem Shvorin

July 17, 2009

Perspectives of System Informatics, June 15-19, Novosibirsk, Russia

Outline

- Theorem proving and program verification by program optimization
- Verification of protocol models by supercompilers
  - Modeling of protocols (due to G.Delzanno)
  - Encoding in Java and applying Java Supercompiler JScp
- Overview of Java Supercompiler JScp
- Discussion and Conclusion

Theorem Proving and Program Verification by Program Optimization

Theorem proving by program optimization

- Given
  - a computable predicate P(x) – a function in some programming language
- To prove or refute
  - ∀x P(x) when P(x) terminates
- Optimize the program P and conclude that
  - the statement is proven if the residual code looks like
    P(x) = true
  - the statement is refuted and a counter example x = A is found if the residual code looks like
    P(x) = if x = A then false else …
- In principle, any program optimizer can be used
  - The class of provable statements depends on the power of the program optimizer
  - A nice test problem to compare specializers and other optimizers

Program verification by program optimization

- Given
  - a program: F(x)
  - a postcondition: P(y) – a total function in the same language
- To prove or refute
  - ∀x P(F(x)) when F(x) terminates
- Write the following program G:
  - G(x) = P(F(x))
- Optimize (specialize, supercompile, etc) G and conclude the program F is verified if the residual code looks like
  - G(x) = true
- More practical:
  - … if the residual code contains return statements (in case of Java) only of form return true (no return false and no return *expression*)

Verification of protocol models by supercompilers

A Class of Verification Problems Soluble by Supercompilers

- A.Nemytykh and A.Lisitsa has found a nice class of verification problems soluble by supercompilers:
  - Verification of models of cache coherence protocols following G.Delzanno and that of other similar parameterized automata
- They performed successful experiments with the Refal Supercompiler SCP4 developed by V.Turchin and A.Nemytykh
- We reproduced the experiments with our Java Supercompiler JScp
- All of the considered protocol models have been either verified, or contain an error, which has been found by the supercompilers
- This suggests ideas that
  - The result is based on the essence of supercompilation rather than particular improvements and tricks
  - The models, pre- and postconditions belong to a class for which it can be proven that the supercompilers successfully verify them

Modeling of Protocols (informally)

- The behavior of a protocol is described by n identical finite automata
  - e.g., in the MOESI cache-coherence protocol the names of states are invalid, exclusive, shared, modified, owned
- Rules define when simultaneous state transition is allowed, e.g. in MOESI:
  - if some automaton is in invalid state
    - this invalid → shared
    - all exclusive → shared
    - all modified → owned
  - if some automata is in exclusive state
    - this exclusive → modified
  - if some automaton is in shared or owned state
    - this shared or owned → exclusive
    - all other → invalid
  - …
- Condition for allowed initial states
  - e.g. in MOESI, all automata initially are in invalid state
- Condition for "unsafe" states that must not be reached, e.g. in MOESI:
  - some automaton is in modified state and some automaton is in exclusive, shared or owned state, or
  - some automaton is in exclusive state and some automaton is in shared or owned state, or
  - 2 automata are in modified state, or 2 automata are in exclusive state

Modeling of Protocols (formally)

Due to G.Delzanno, a protocol model is an Extended Finite State Machine (EFSM)

- The model state is a tuple of natural numbers $(x_1,\dots, x_k)$, where
  - k is the number of automata states
  - x_i is the number of automata in k-th state

  e.g. in MOESI protocol
  - $k = 5$, the model state is (invalid, exclusive, shared, modified, owned) where variables are named after respective automata states

- Transition rules have form
  - if L then R where L is a conjunction of conditions of form $x_i = l_i$ or $\Sigma x_{ij} \geq l_i$ R is a sequence of assignments of form $x_i' = r_i$ or $x_i' = x_i + \Sigma x_{ij} + r_i$

  e.g. in MOESI protocol
  - if invalid ≥ 1 then invalid′ = invalid – 1, exclusive′ = 0, modified′ = 0, shared′ = shared + exclusive + 1, owned′ = owned + modified
  - if exclusive ≥ 1 then exclusive′ = exclusive – 1, modified′ = modified + 1
  - if shared + owned ≥ 1 then …

- Condition for allowed initial states of form $x_i = l_i$ or $x_i \geq l_i$, e.g. in MOESI:
  - invalid ≥ 1, exclusive = 0, shared = 0, modified = 0, owned = 0

- Conditions for "unsafe" states that must not be reached of form $\&(\Sigma x_{ij} \geq l_i)$, e.g.
  - exclusive + shared + owned ≥ 1 and modified ≥ 1, or
  - exclusive ≥ 1 and shared + owned ≥ 1, or
  - modified ≥ 2, or exclusive ≥ 2

Program model in Java of MOESI cache-coherence protocol

```java
public boolean runModel(int[i] actions, int[] pars)
    throws ActionNonapplicableException
{
  // set and check initial state (precondition)
  int invalid   = pars[0], invalid_   = invalid;
  int exclusive = 0,       exclusive_ = exclusive;
  int shared    = 0,       shared_    = shared;
  int modified  = 0,       modified_  = modified;
  int owned     = 0,       owned_     = owned;

  require (invalid >= 1);

  // execute actions
  for (int i = 0; i < actions.length; i++) {

    // execute one action
    switch (action) {
      ...
      default:
        require(false);
    }
    invalid   = invalid_;
    exclusive = exclusive_;
    shared    = shared_;
    modified  = modified_;
    owned     = owned_;
  }

  // check final state (postcondition)
  if (exclusive + shared + owned >= 1 && modified >= 1)
      return false;
  if (exclusive >= 1 && shared + owned >= 1) return false;
  if (modified >= 2) return false;
  if (exclusive >= 2) return false;
  return true;
}
```

To prove: never returns false

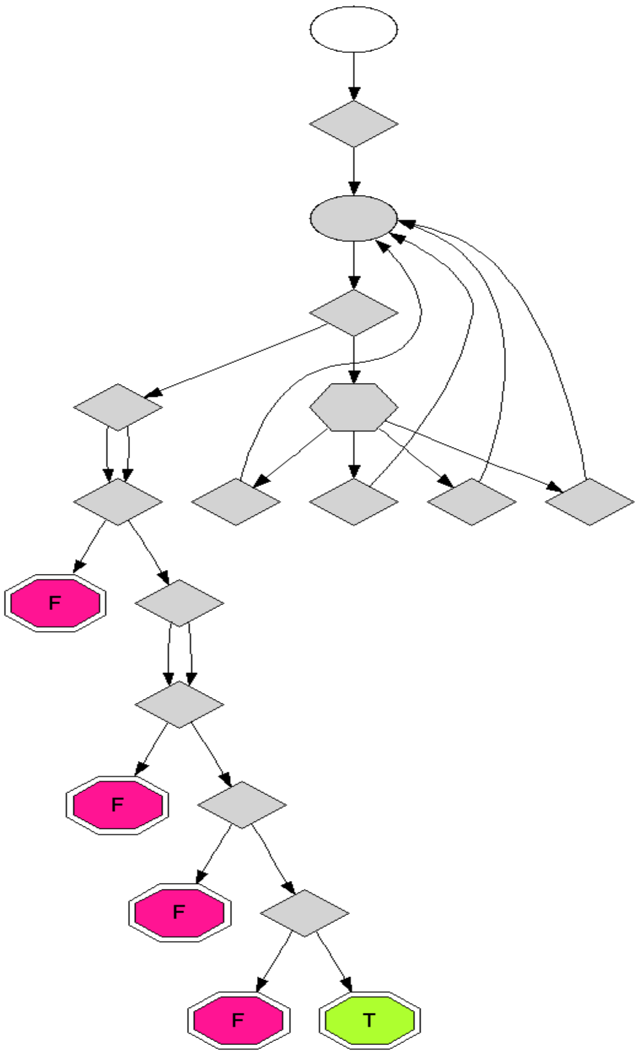
```java
// definition of actions
case rm:
  require (invalid >= 1);
  invalid_   = invalid - 1;
  exclusive_ = 0;
  modified_  = 0;
  shared_    = shared + exclusive + 1;
  owned_     = owned + modified;
  break;

case wh2:
  require (exclusive >= 1);
  exclusive_ = exclusive - 1;
  modified_  = modified + 1;
  break;

case wh3:
  require (shared + owned >= 1);
  shared_    = 0;
  exclusive_ = 1;
  modified_  = 0;
  owned_     = 0;
  invalid_   = invalid + modified +
               exclusive + shared +
               owned - 1;
  break;

case wm:
  require (invalid >= 1);
  shared_    = 0;
  exclusive_ = 1;
  modified_  = 0;
  owned_     = 0;
  invalid_   = invalid + modified +
               exclusive + shared +
               owned - 1;
  break;
```

```java
void require(boolean b) throws ModelException {
    if (!b) throw new ModelException();
}
```

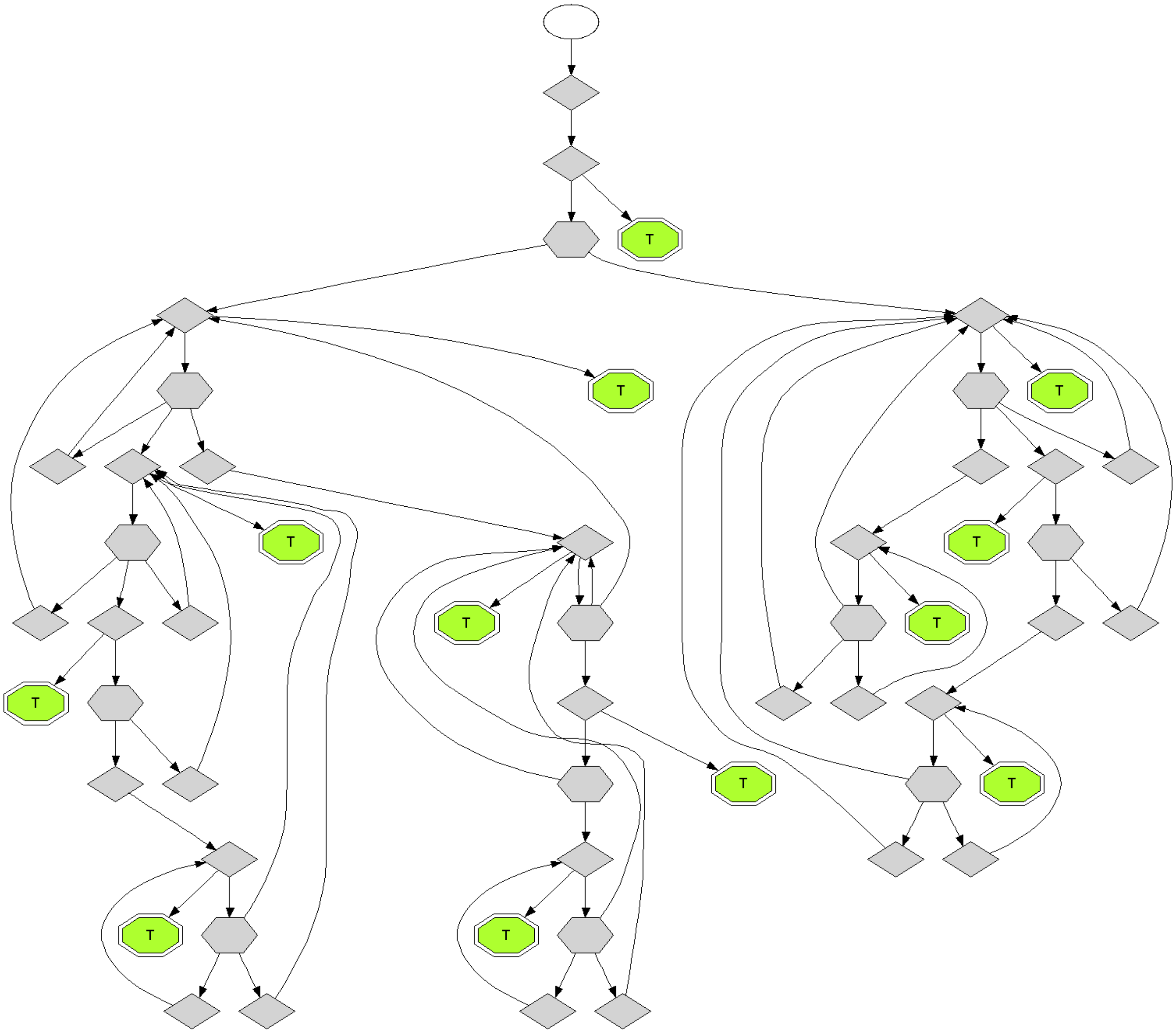
Program model in Java of MOESI cache-coherence protocol

```java
public boolean runModel(int[i] actions, int[] pars)
    throws ActionNonapplicableException
{
  // set and check initial state (precondition)
  int invalid   = pars[0], invalid_   = invalid;
  int exclusive = 0,       exclusive_ = exclusive;
  int shared    = 0,       shared_    = shared;
  int modified  = 0,       modified_  = modified;
  int owned     = 0,       owned_     = owned;

  require (invalid >= 1);

  // execute actions
  for (int i = 0; i < actions.length; i++) {

    // execute one action
    switch (action) {
      ...
      default:
        require(false);
    }
    invalid   = invalid_;
    exclusive = exclusive_;
    shared    = shared_;
    modified  = modified_;
    owned     = owned_;
  }

  // check final state (postcondition)
  if (exclusive + shared + owned >= 1 && modified >= 1)
      return false;
  if (exclusive >= 1 && shared + owned >= 1) return false;
  if (modified >= 2) return false;
  if (exclusive >= 2) return false;
  return true;
}
```

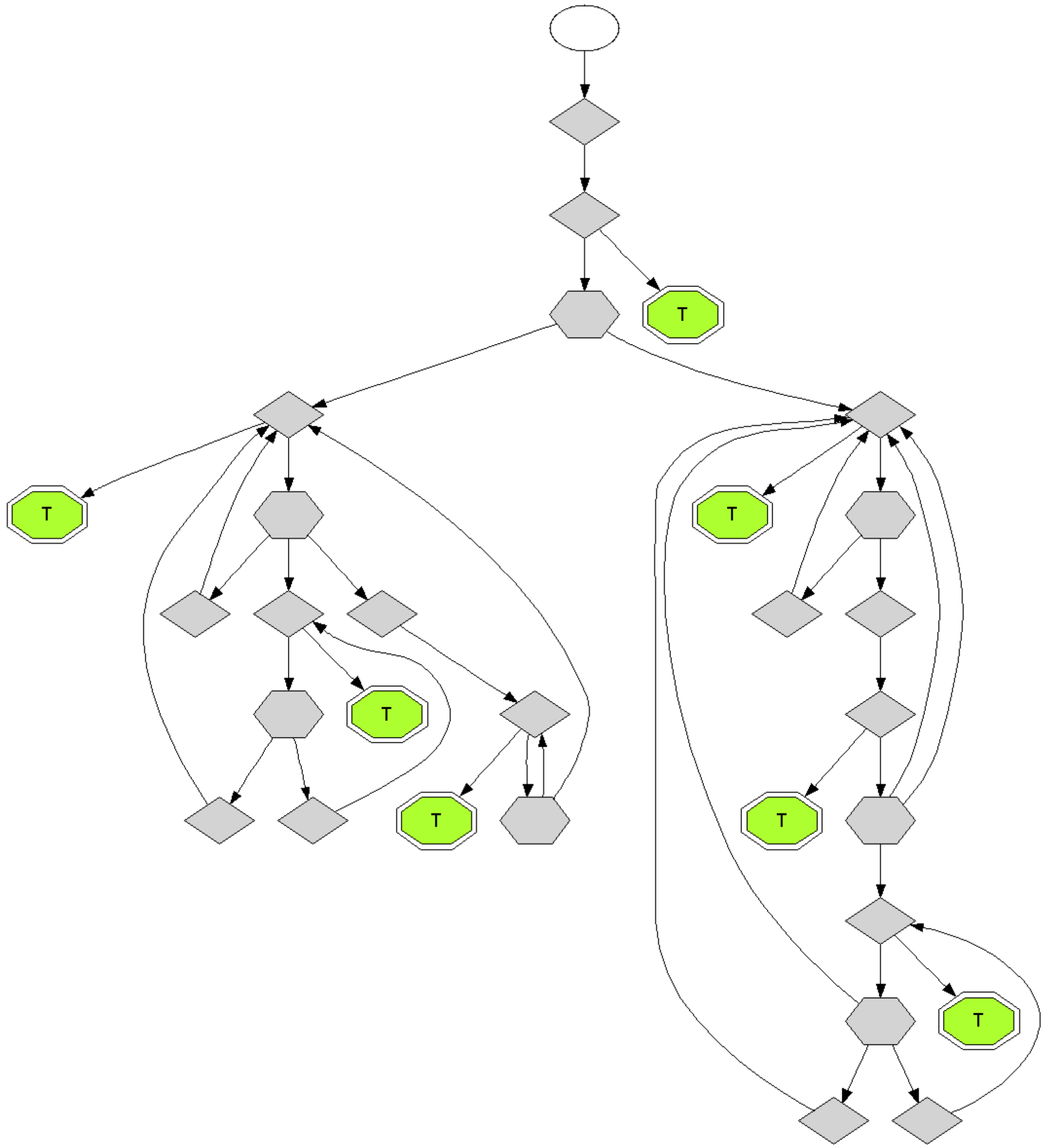
To prove: never returns false



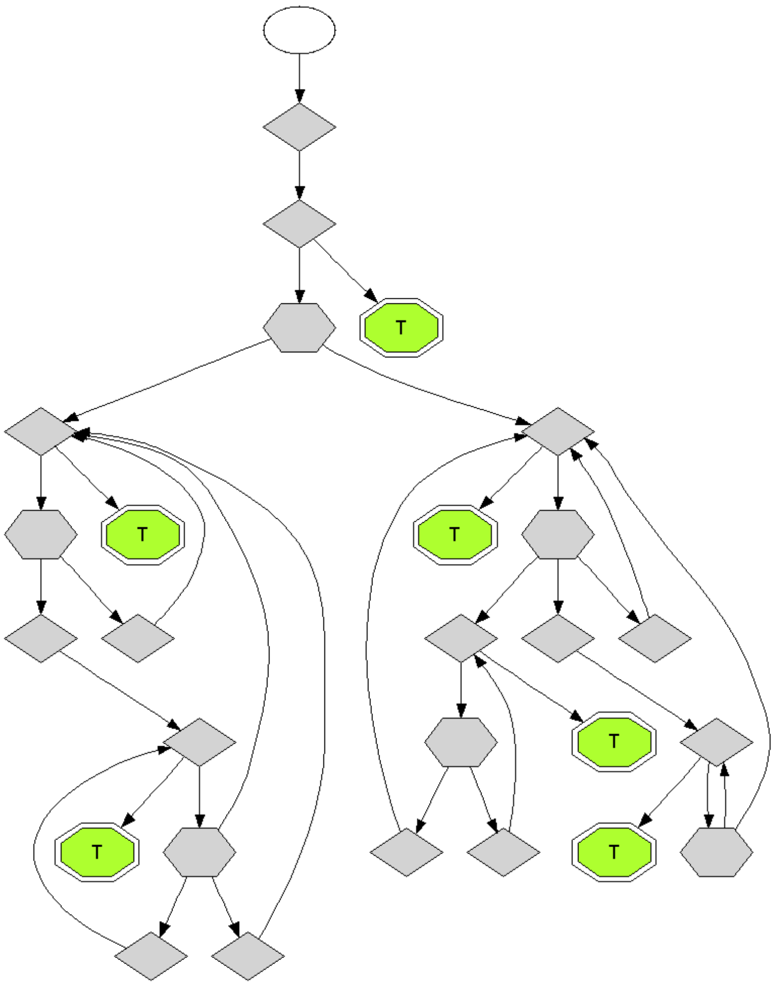
Residual code of MOESI cache-coherence protocol model



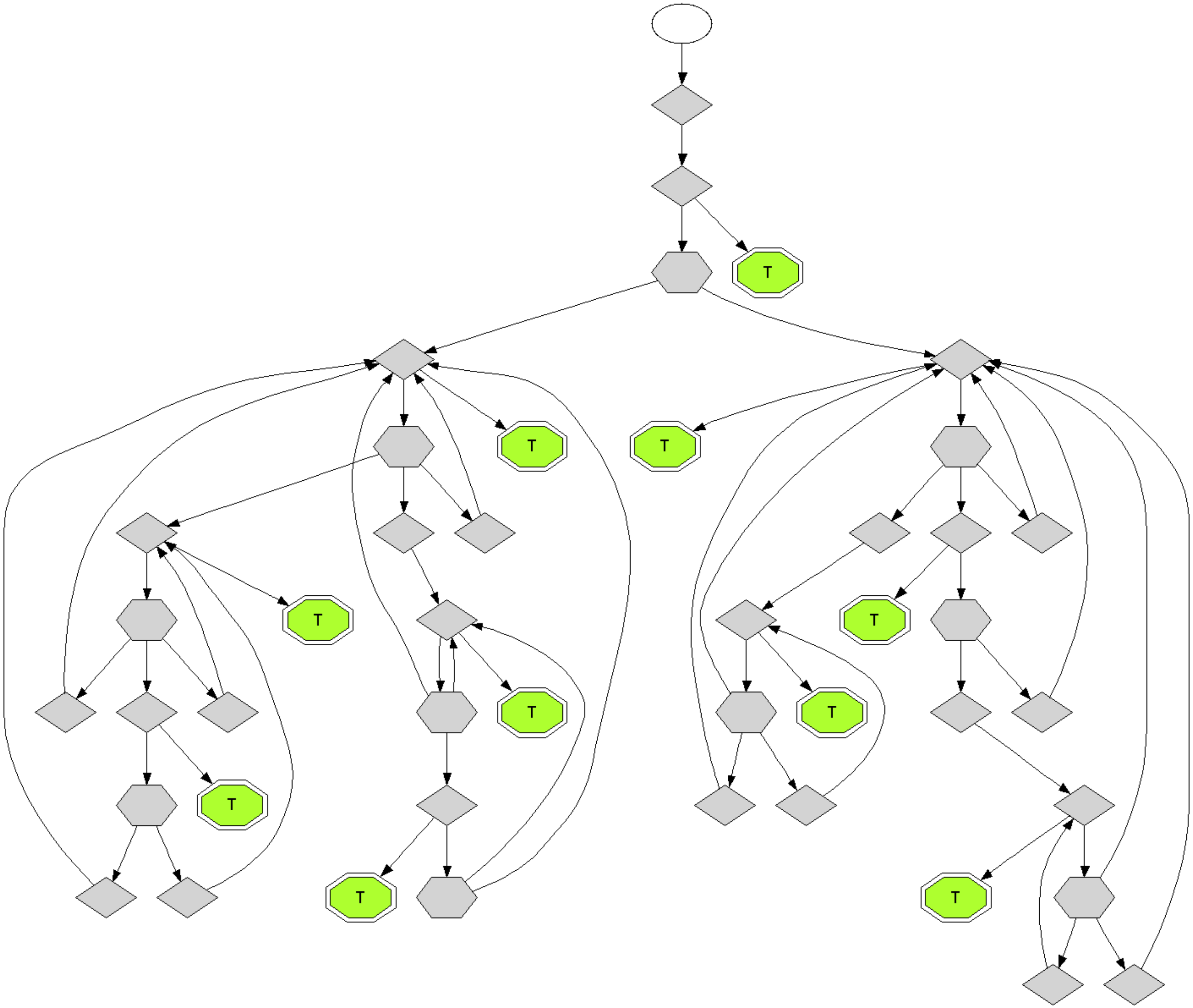
Residual code of Synapse cache-coherence protocol model



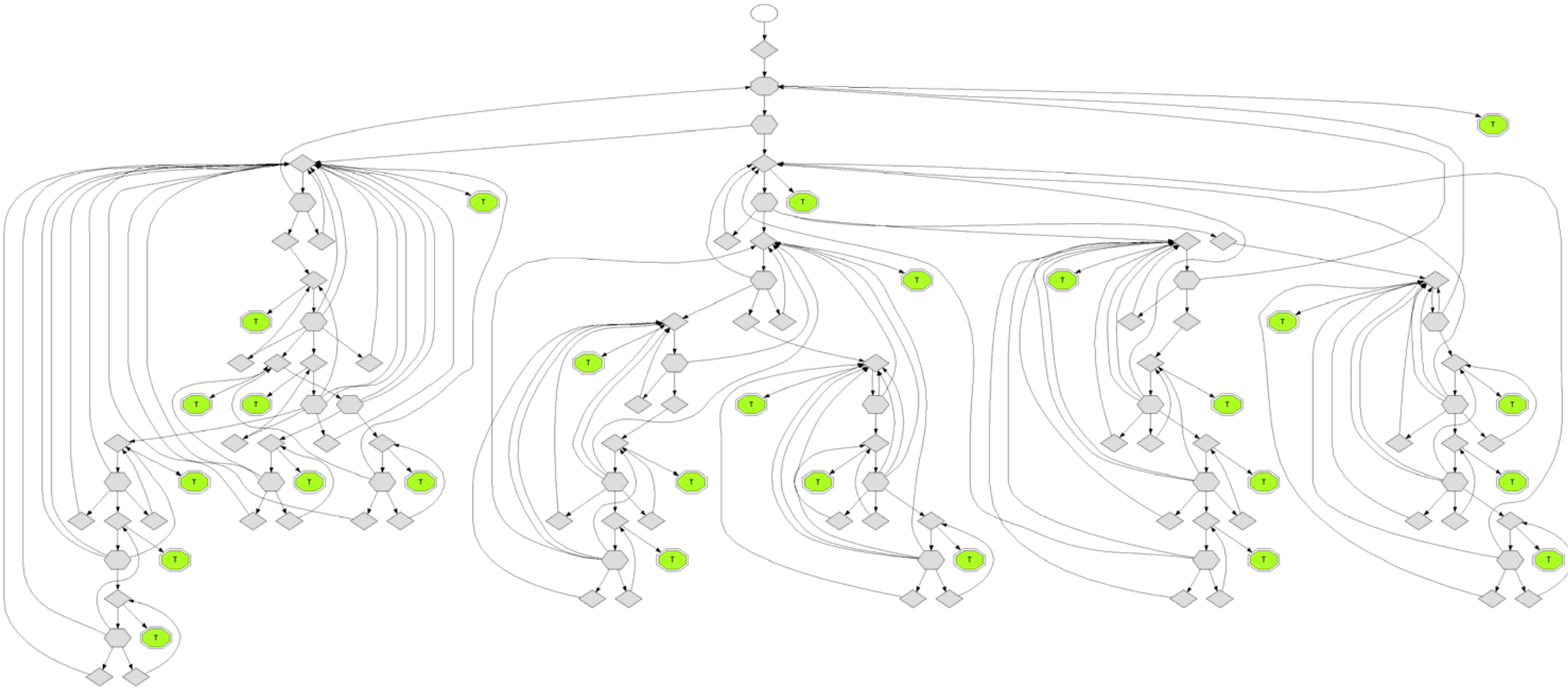
Residual code of MSI cache-coherence protocol model



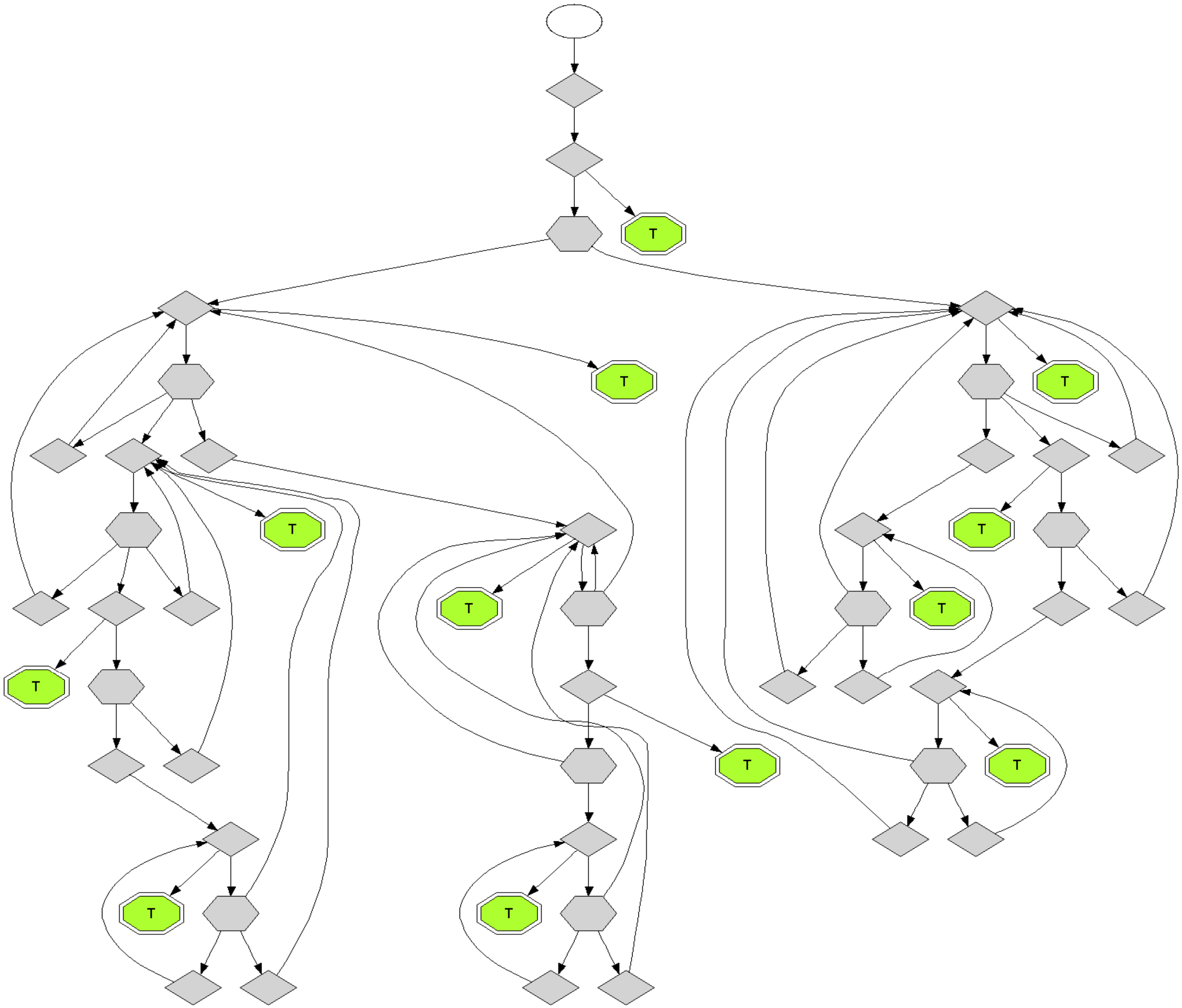
Residual code of MESI cache-coherence protocol model



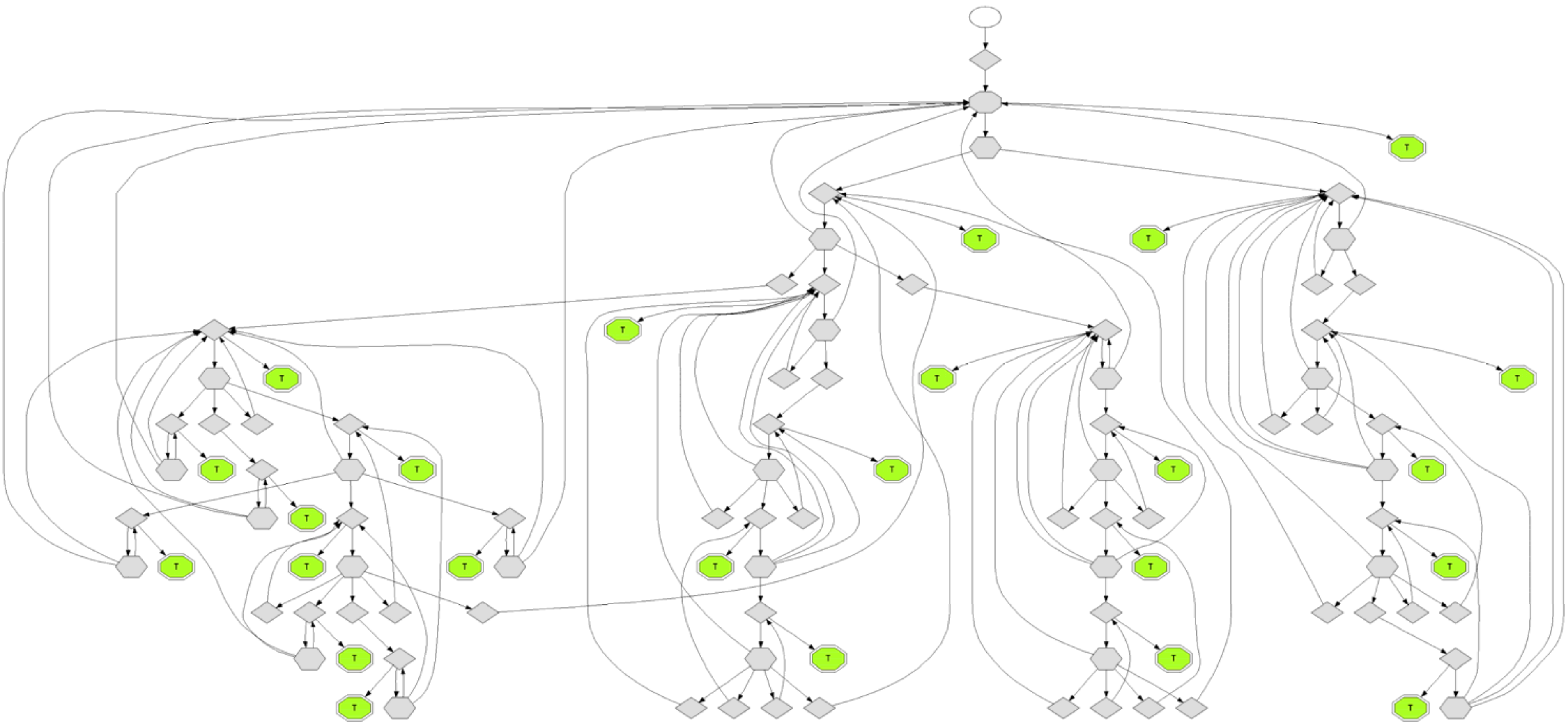
Residual code of MOSI cache-coherence protocol model



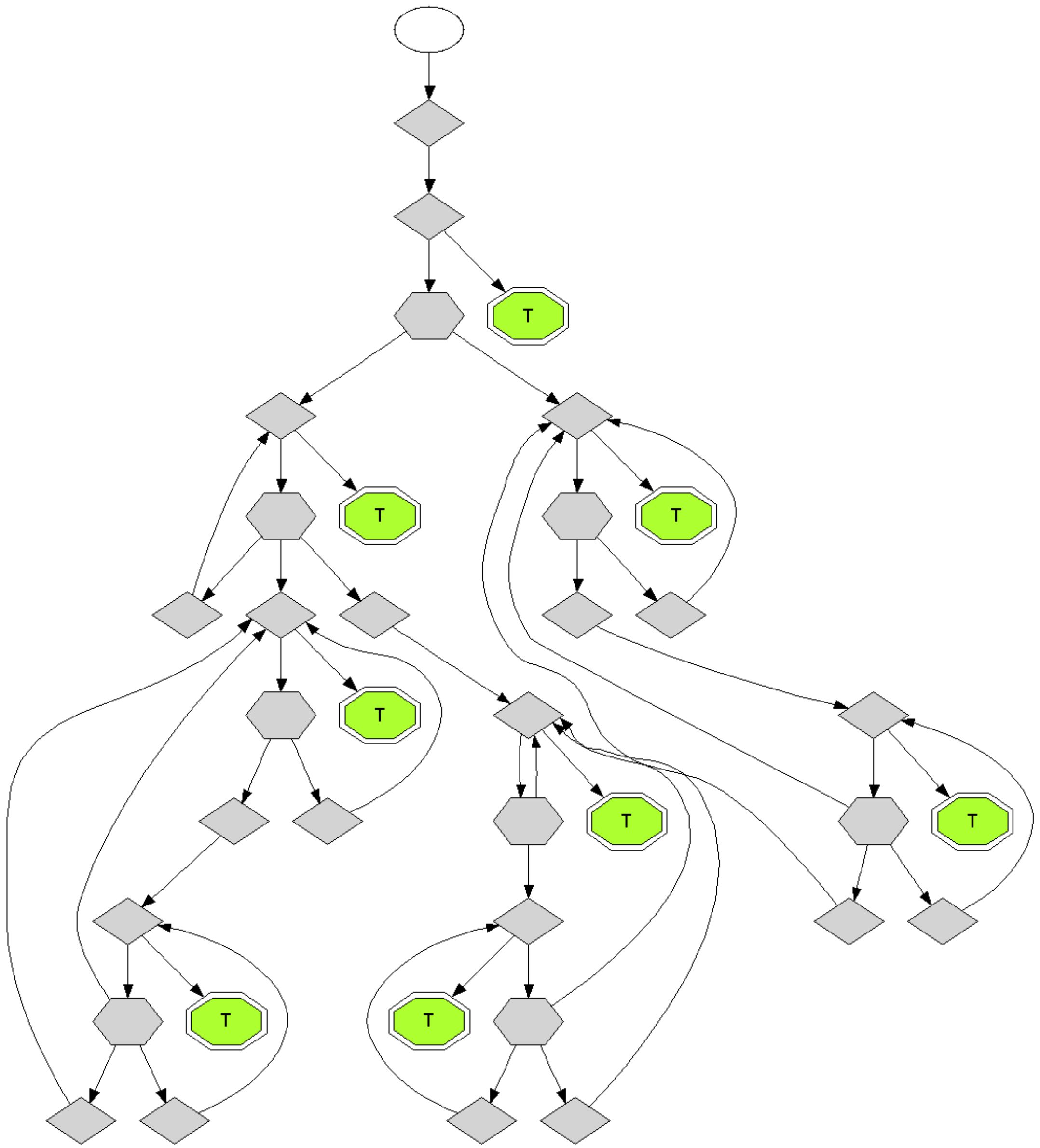
Residual code of MOESI cache-coherence protocol model



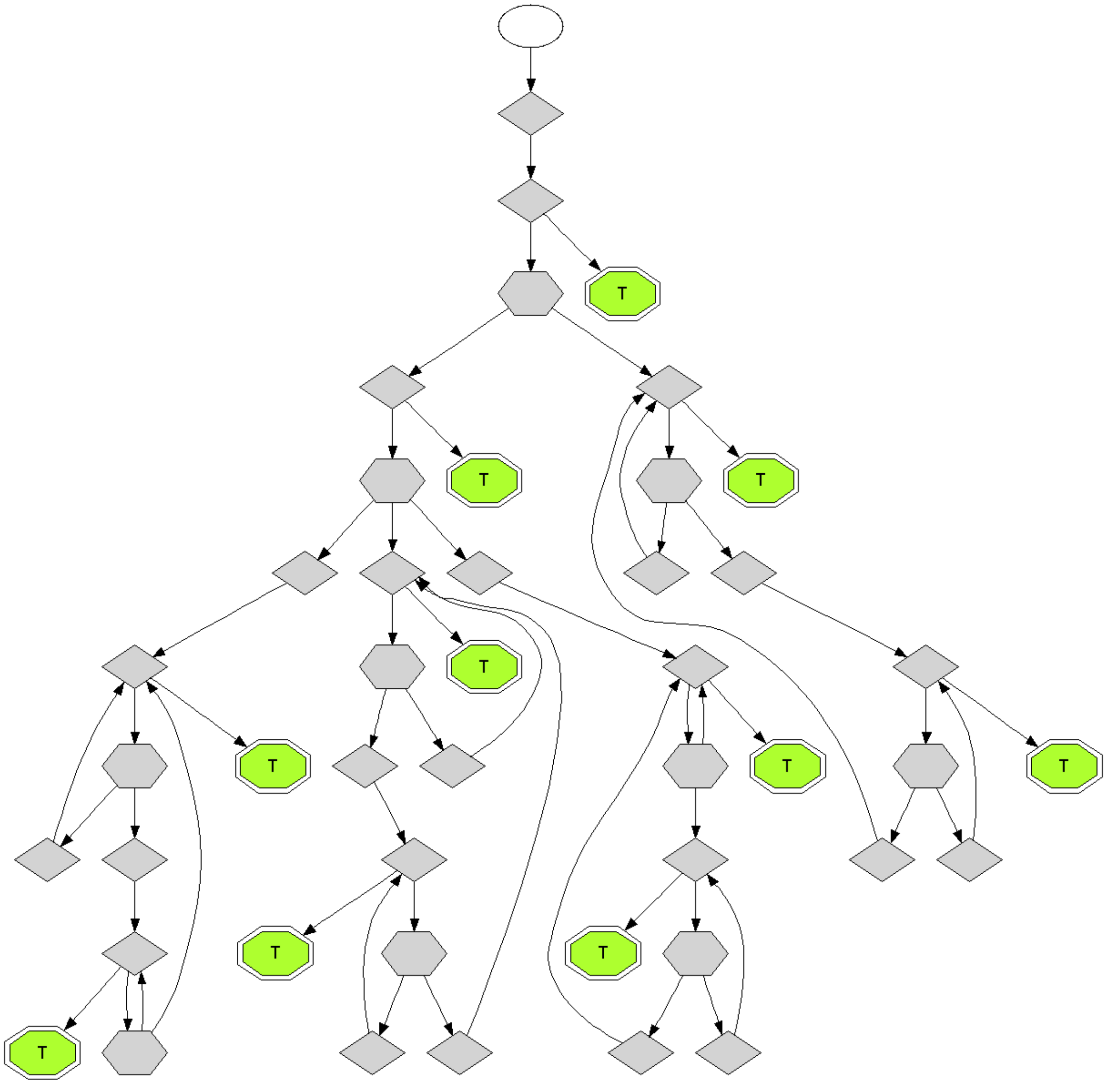
Residual code of Illinois cache-coherence protocol model



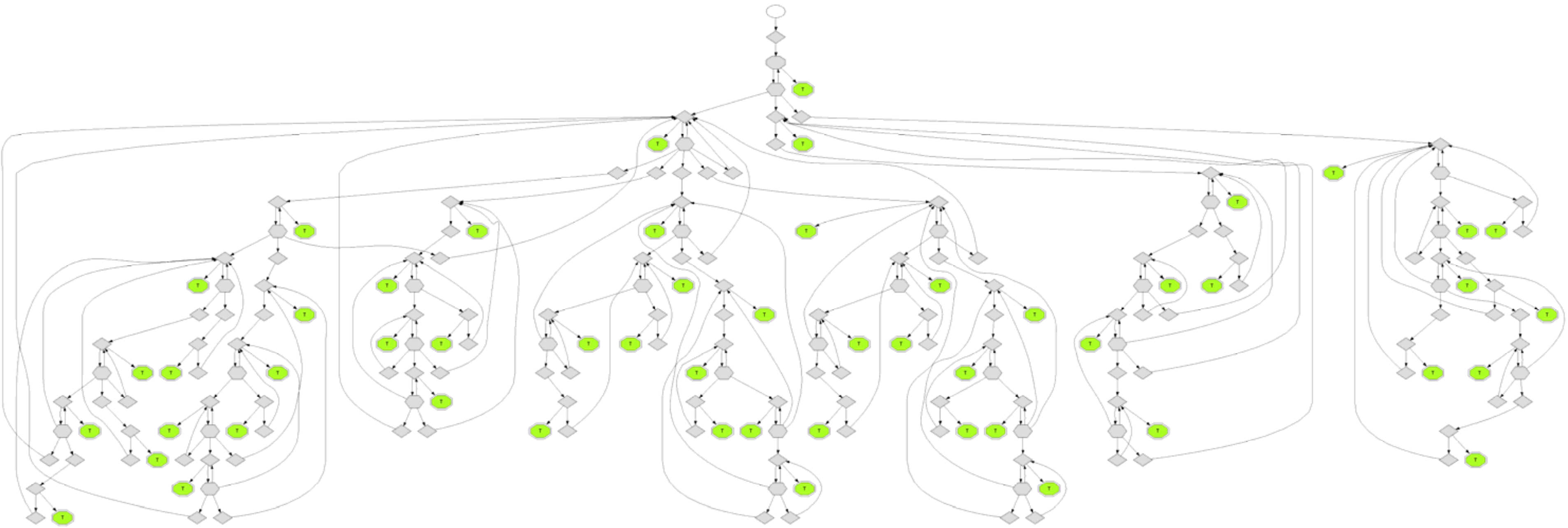
Residual code of Berkley cache-coherence protocol model



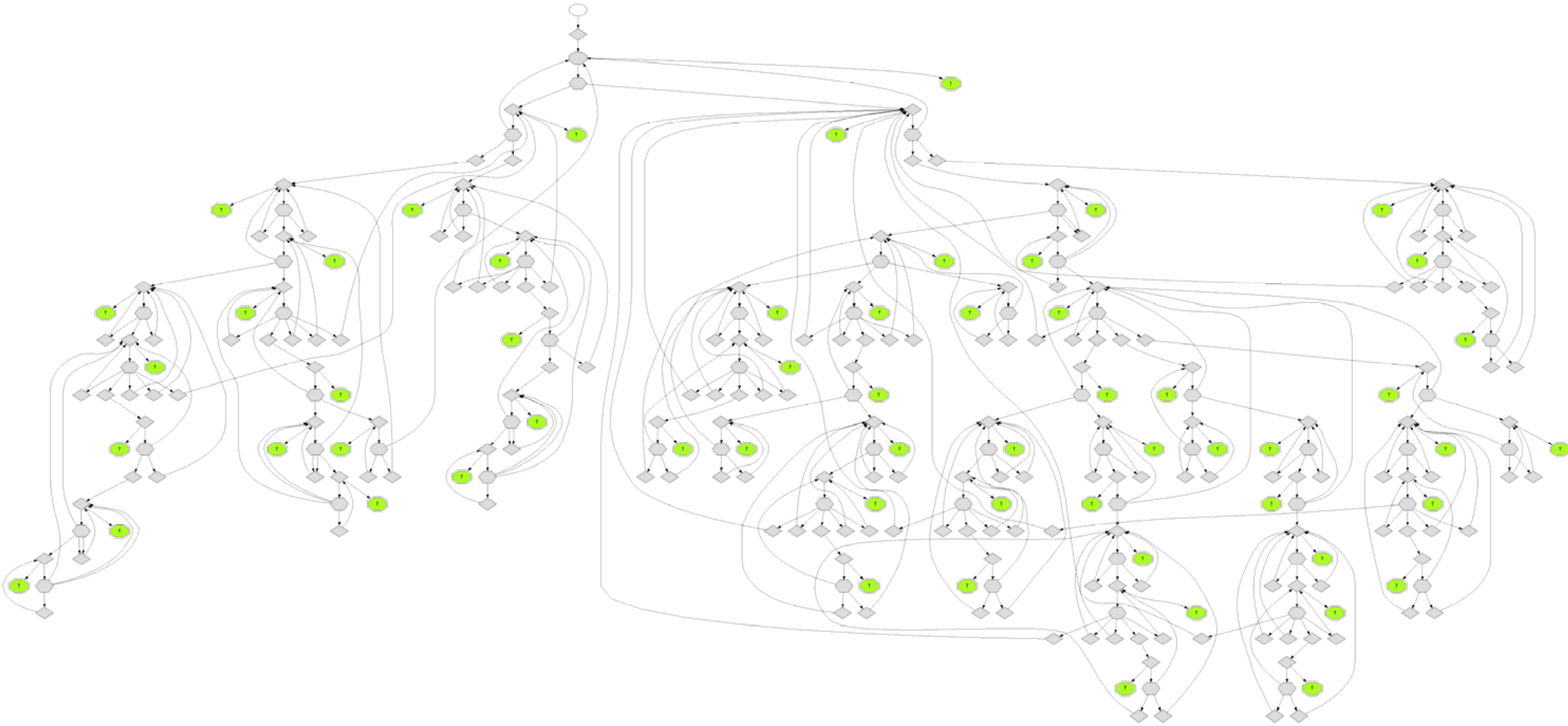
Residual code of Firefly cache-coherence protocol model



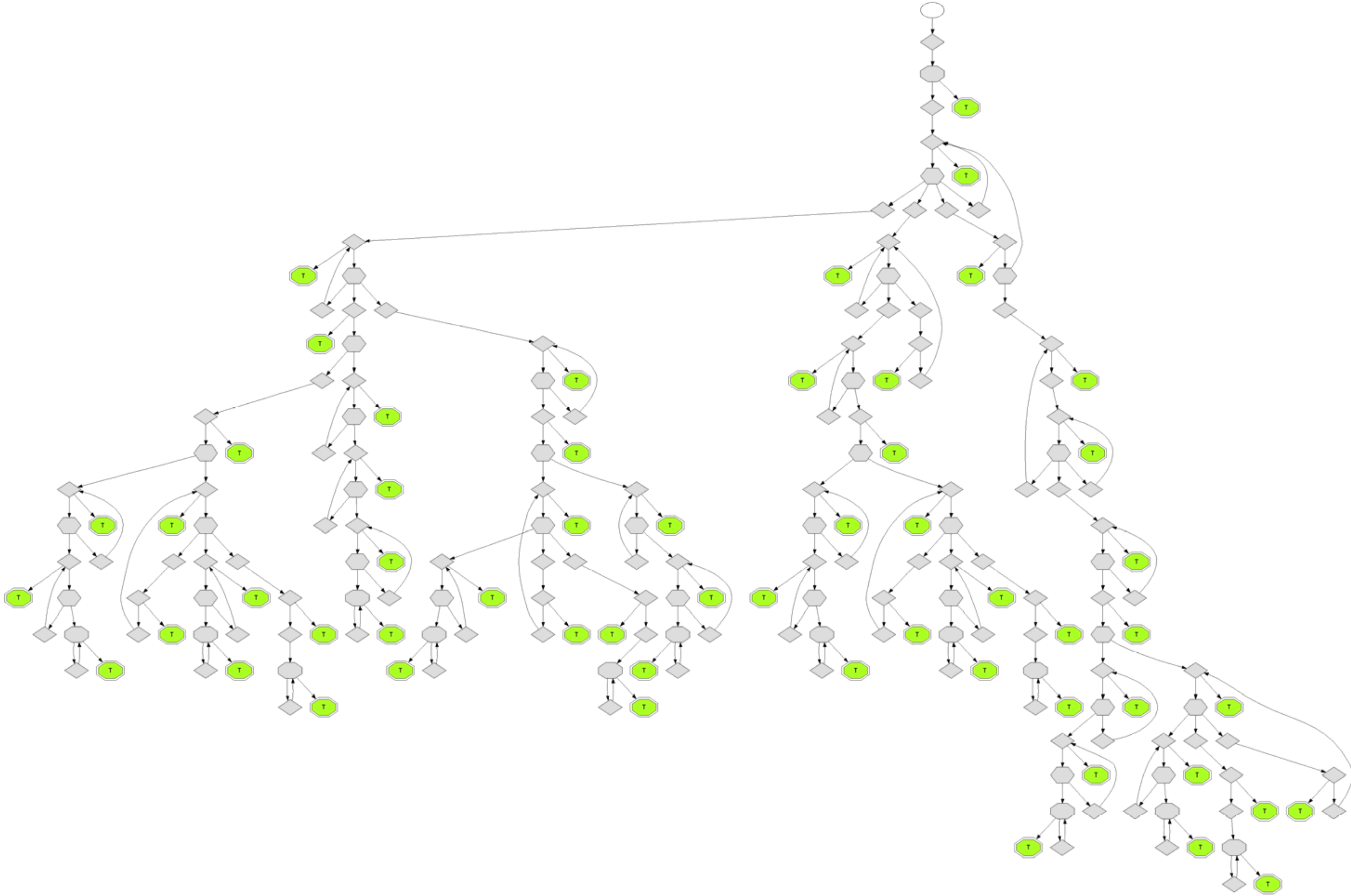
Residual code of Futurebus cache-coherence protocol model



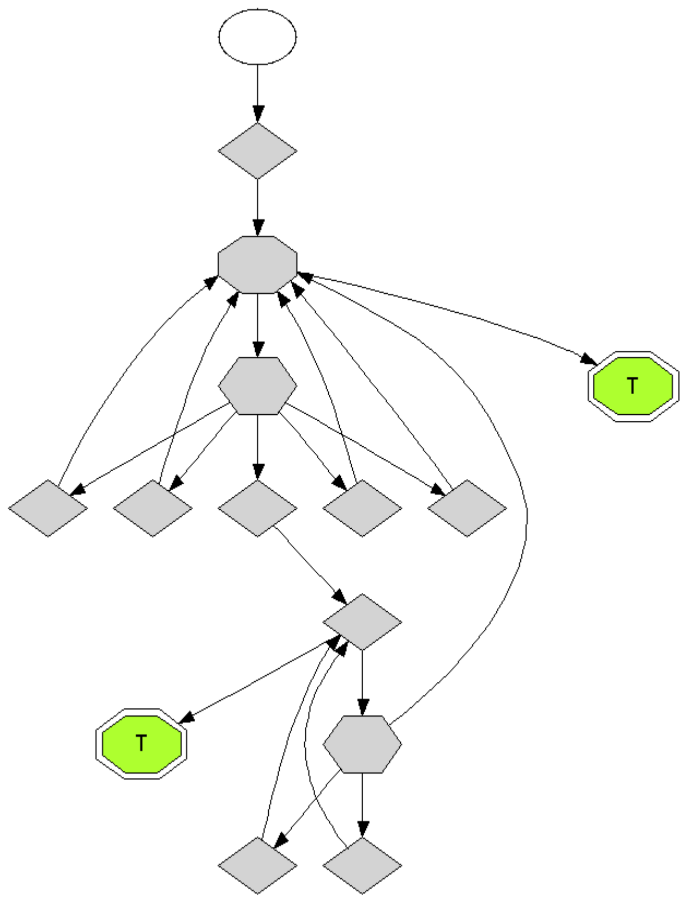
Residual code of Dragon cache-coherence protocol model



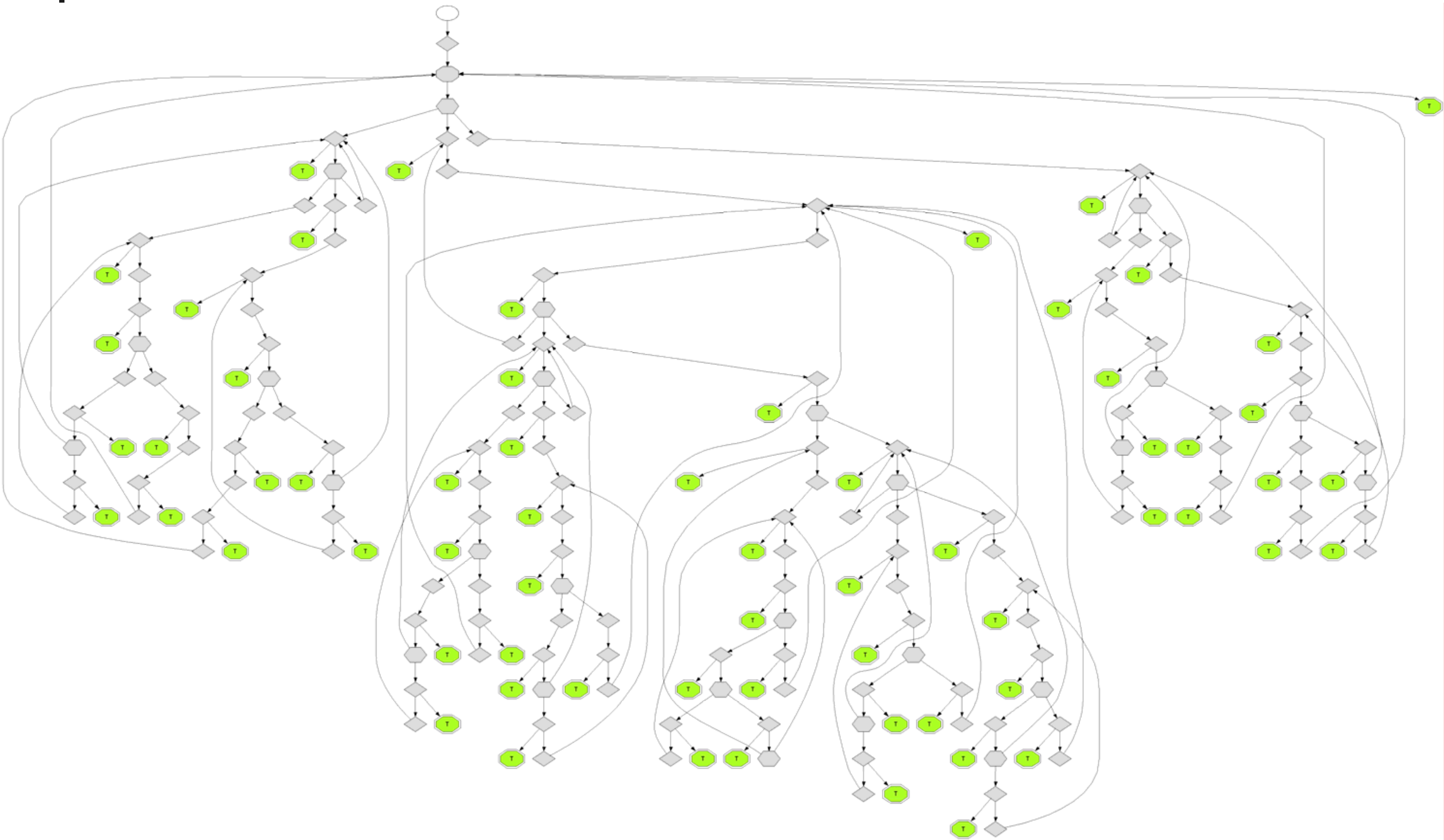
Residual code of JavaMetaLocking cache-coherence protocol



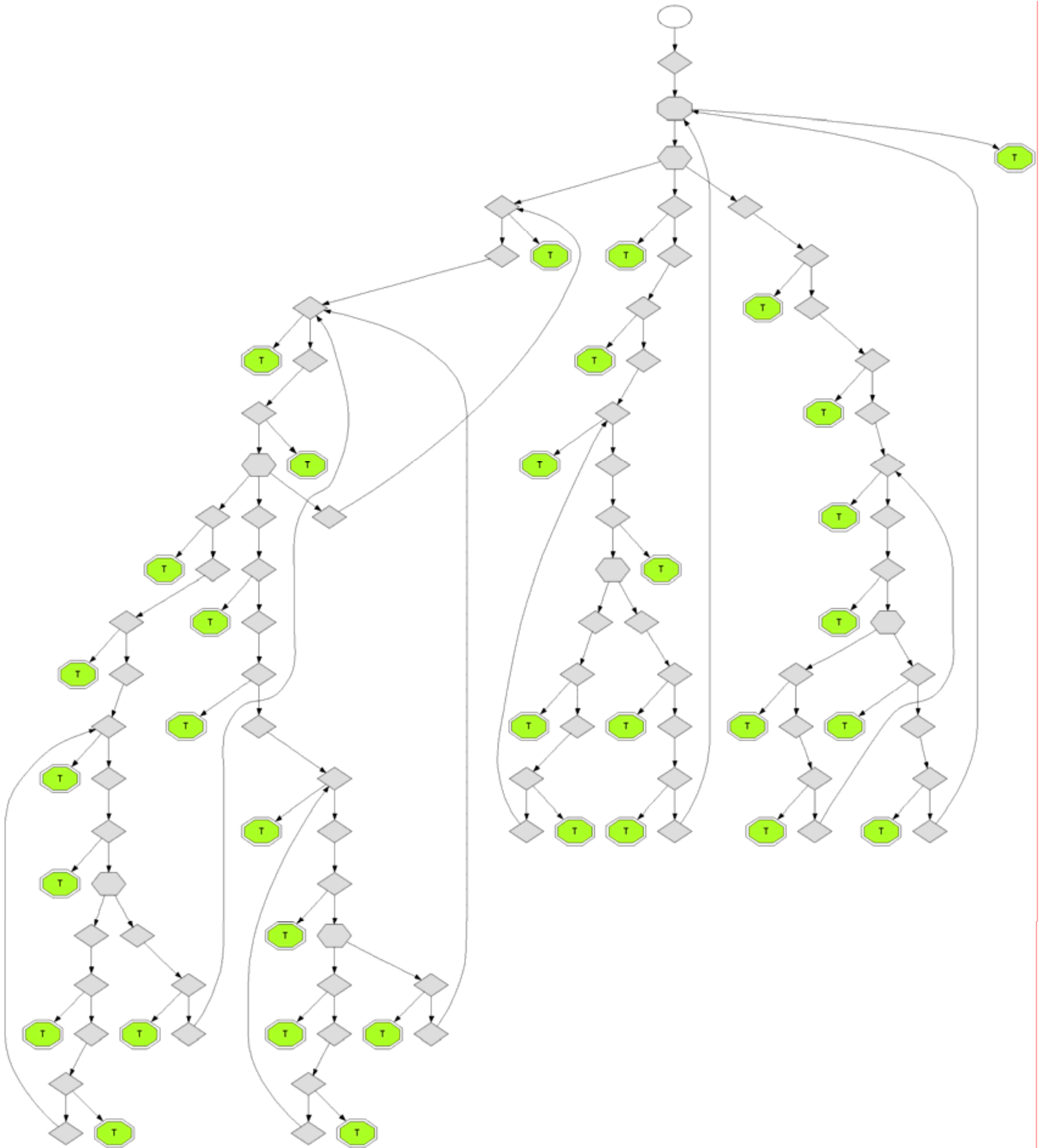
Residual code of ReaderWriter cache-coherence protocol



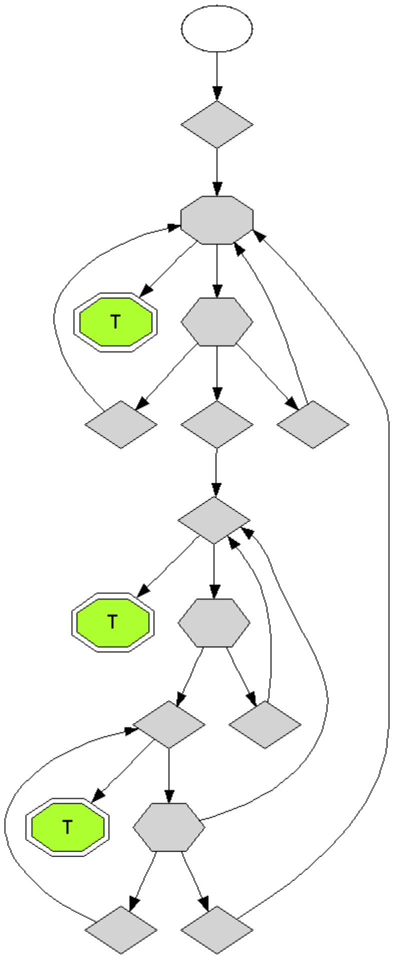
Residual code of German I cache-coherence protocol model



Residual code of German B cache-coherence protocol model



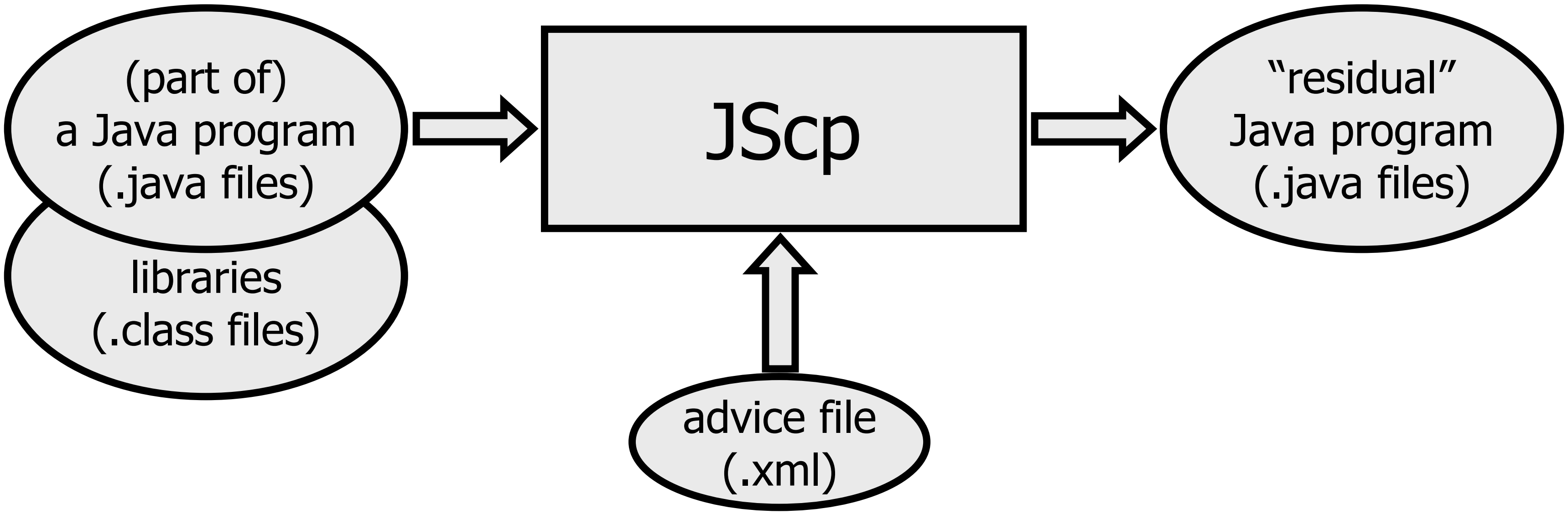
Residual code of DataRaceFreeSynchro cache-coherence protocol model



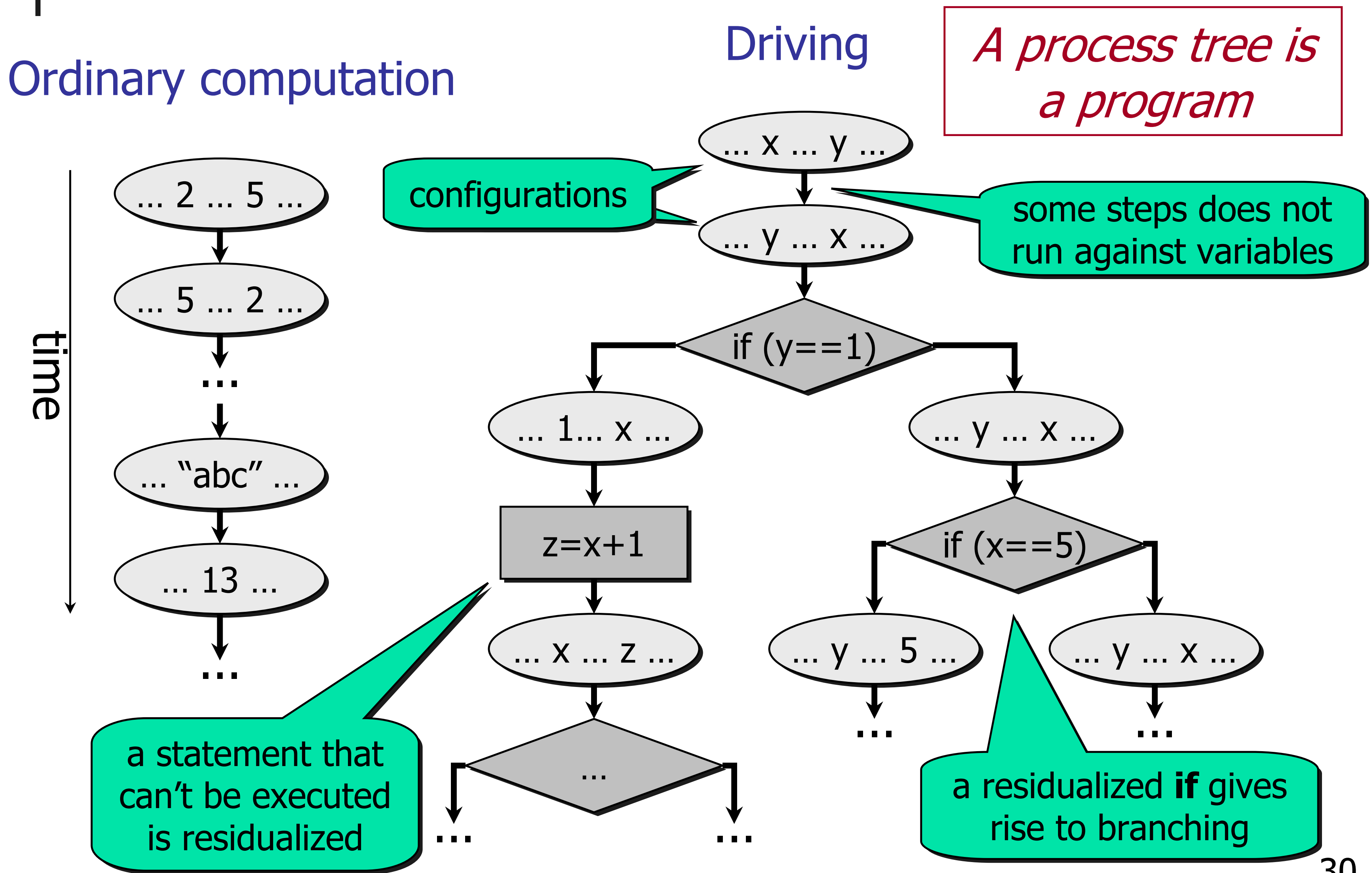
Overview of Features
of the Java Supercompiler JScp

What is the Java Supercompiler?

JScp is a source-to-source program optimizer



Driving: building process tree



The main notions of supercompilation

- Configuration
  - a set of states = a generalized program state = a state with variables
- Driving
  - building a potentially infinite process tree
- Configuration analysis
  - multiple transformations of a process graph (starting with a tree) until in becomes finite
    - by reducing a configuration to an equivalent or wider one
    - by generalizing a configuration to a wider one
    - by cutting a configuration into parts

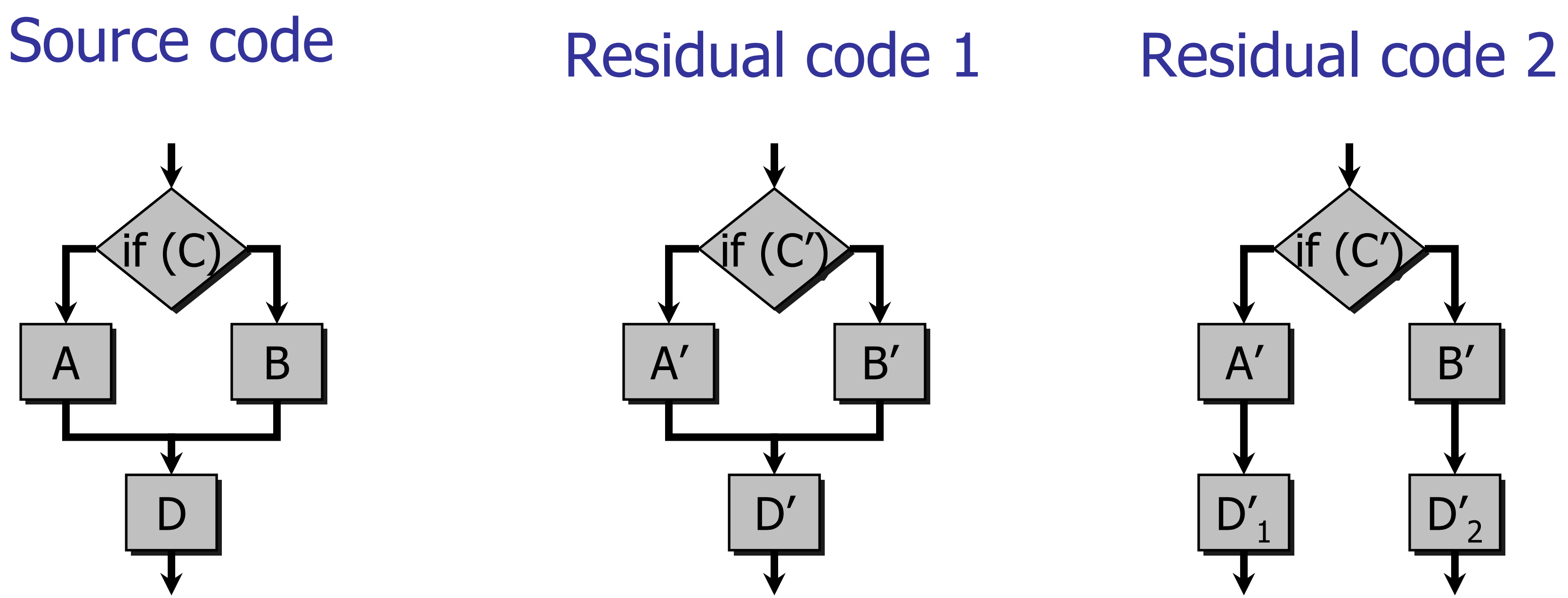
The notion of configuration for Java

- Configuration
  - Stack of frames, each:
    - Control point
    - Operand stack
    - Local environment
  - Heap
    - mapping of reference variables to object "abstractions"
  - Classes
    - static non-final variables
      - always unknown
    - static final variables
      - known after initialization
- Wherever a ground value is allowed, a configuration variable may occur
- Note: one thread now; many threads in future

- Configuration variable
  - is
    - a parameter of a configuration
    - a residual local variable
  - has
    - identity (a unique number)
    - type
    - restriction (now: $i \geq k$)
  - reference variable is
    - a key to the heap
    - was it produced by **new** at supercompilation time?
- "Abstract" object in heap
  - fields
  - type
  - is type exact or a super class?
  - is it unique or may be aliased?
  - may the reference be null?
  - etc

Configuration analysis of conditional statements

- 2 alternatives to continue after statements with multiple exits

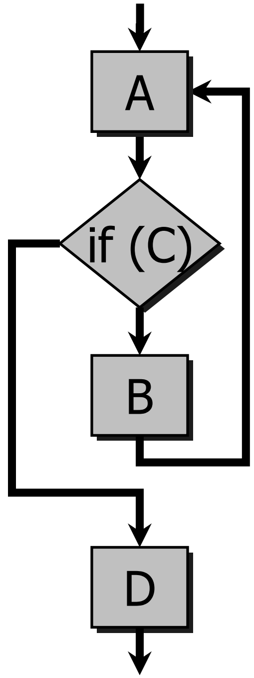


The choice is made by the human

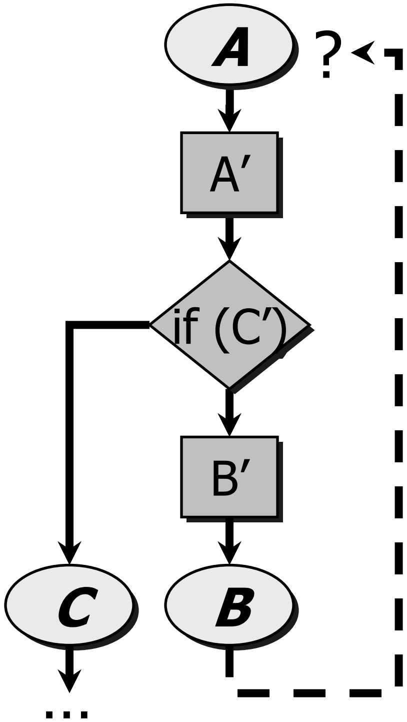
Note the possibility of exponential growth of the residual program

Configuration analysis of loops (1)

Source code



Driving…



How do configurations *A* and *B* relate?

- ***B*** ⊆ ***A*** as sets, that is
  B **= Δ** ***A***, where **Δ** is a substitution
  then loop-back with **Δ** as an assignment

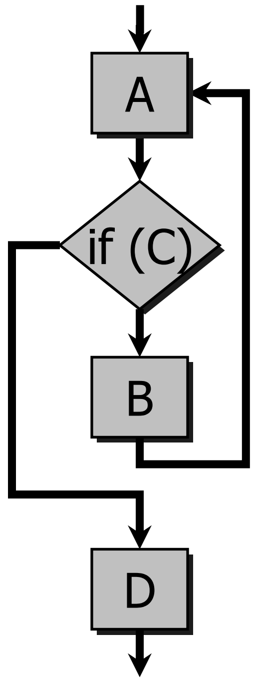
otherwise

- either
  - continue driving from ***B*** forward
- or
  - generalize ***A*** to some ***A′*** such that
    A **= Δ** ***A′***, where **Δ** is a substitution
  - residualize **Δ** as assignments between configurations ***A*** and ***A′***, and
  - continue driving from ***A′***

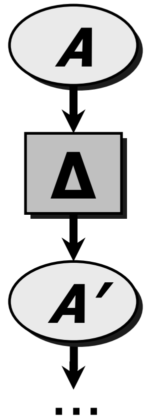
Note the possibility of exponential time to construct the residual program

Configuration analysis of loops (2)

Source code



Driving…



How do configurations *A* and *B* relate?

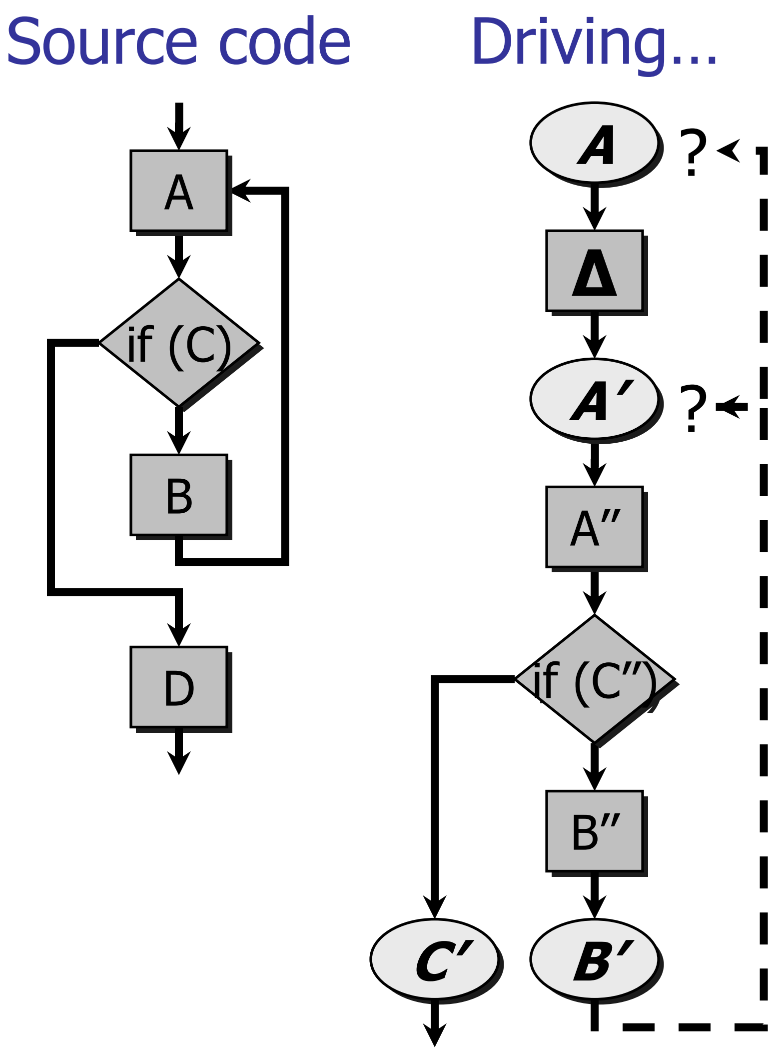
- ***B*** ⊆ ***A*** as sets, that is
  B* = Δ *A, where **Δ** is a substitution
  then loop-back with **Δ** as an assignment

otherwise

- either
  - continue driving from ***B*** forward
- or
  - generalize ***A*** to some ***A′*** such that
    A* = Δ *A′, where **Δ** is a substitution
  - residualize **Δ** as assignments
    between configurations ***A*** and ***A′***,
    and
  - continue driving from ***A′***

Note the possibility of exponential time to construct the residual program

Configuration analysis of loops (3)



How do configurations *A* and *B* relate?

- ***B*** ⊆ ***A*** as sets, that is
  B = **Δ** ***A***, where **Δ** is a substitution
  then loop-back with **Δ** as an assignment

otherwise

- either
  - continue driving from ***B*** forward
- or
  - generalize ***A*** to some ***A′*** such that ***A*** = **Δ** ***A′***, where **Δ** is a substitution
  - residualize **Δ** as assignments between configurations ***A*** and ***A′***, and
  - continue driving from ***A′***

Note the possibility of exponential time to construct the residual program

When to terminate loop unrolling?

- Supercompilers (like many other formal system transformers) usually use well-quasi-orders (WQO) of configurations to terminate and forcedly generalize configurations
  - A pre-order $\trianglelefteq$ (reflexive transitive relation) is a well-quasi-order if in any infinite sequence $\{x_i\}$ there exist x_i and x_j, $i < j$, such that $x_i \trianglelefteq x_j$
- The author of a supercompiler has to chose some reasonable WQO on configurations and generalize one of configurations C_i and C_j (found on one path in graph) such that $C_i \trianglelefteq C_j$
  - Most popular WQO – homeomorphic embedding of terms: roughly, $t_1 \trianglelefteq t_2$ if the text representation of t_1 can be obtained from that of t_2 by cleaning some of its parts
- In JScp
  - for integers: $i_1 \trianglelefteq i_2$ if $i_1 < i_2$
  - for restrictions on integer configuration variables: $(v_1 \geq i_1) \trianglelefteq (v_2 \geq i_2)$ if $i_1 < i_2$

Discussion and conclusion

- The main reason why the supercompilers verify the considered protocol models is that the transition rules are monotonic with respect to the WQO:
  - for integers: $i_1 \trianglelefteq i_2$ if $i_1 < i_2$
  - for restrictions on integer configuration variables: $(v_1 \geq i_1) \trianglelefteq (v_2 \geq i_2)$ if $i_1 < i_2$
- Based on results on decidability of the reachability problem by P.Abdulla and K.Čerāns for similar class of systems
  - Systems with monotonic (with respect to a WQO) transition rules are referred to as well-structured
- G.Delzanno and others used backward analysis (from postcondition to precondition), while supercompilers use forward analysis (prom precondition to postcondition)
  - The backward analysis solves the reachability problem for a larger class of well-structured systems than backward analysis
  - Subtleties lie in pre- and postconditions
- The main difference between ours and Delzanno’s work is that he used a special-purpose verification system, while supercompilers are universal tools that can do much more than verify this particular class of programs

The end

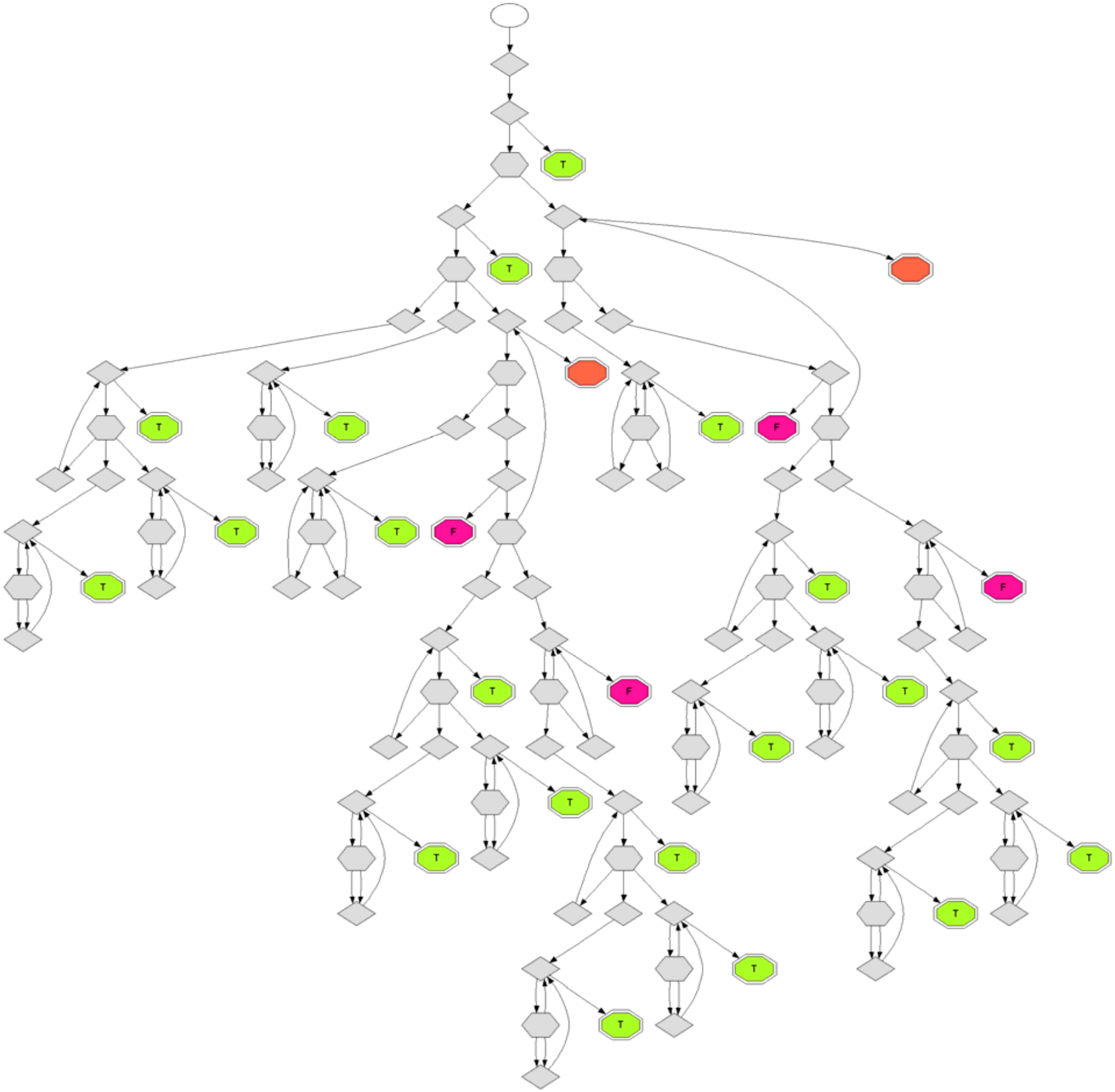
Thank you!
Questions?

Andrei Klimov
klimov@keldysh.ru

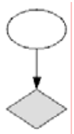
Spare slides

Finding a counter example for an erroneous protocol model

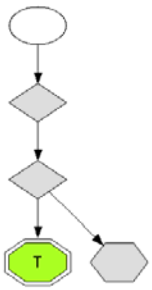
Residual Code of Erroneous Version of Dragon protocol



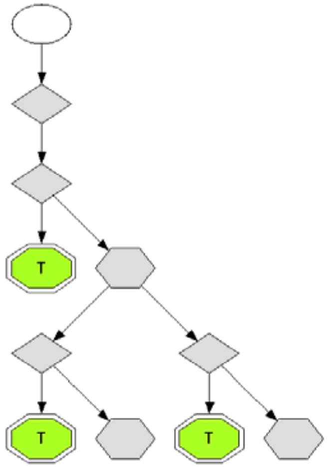
Protocol Dragon Incorrect (-nolca -bol -l0)



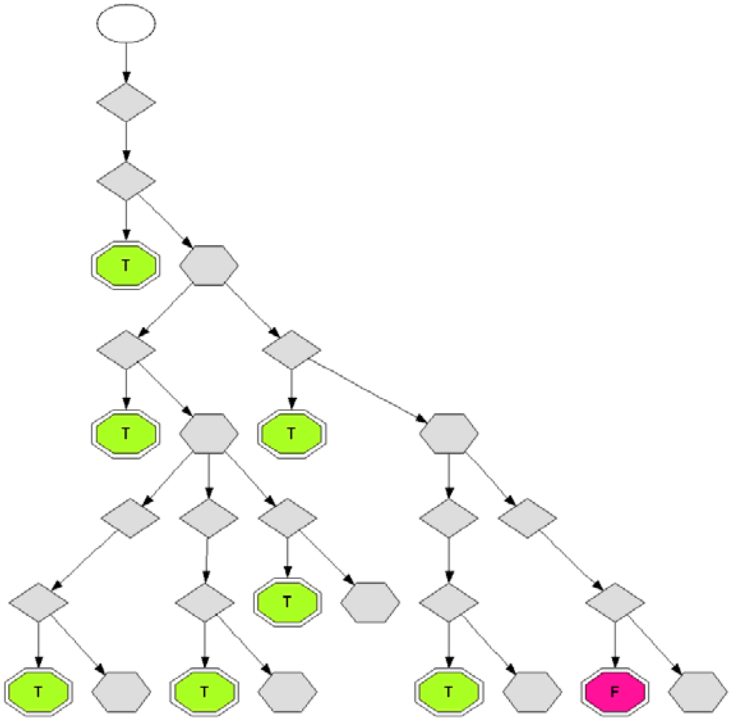
Protocol Dragon Incorrect (-nolca -bol -l1)



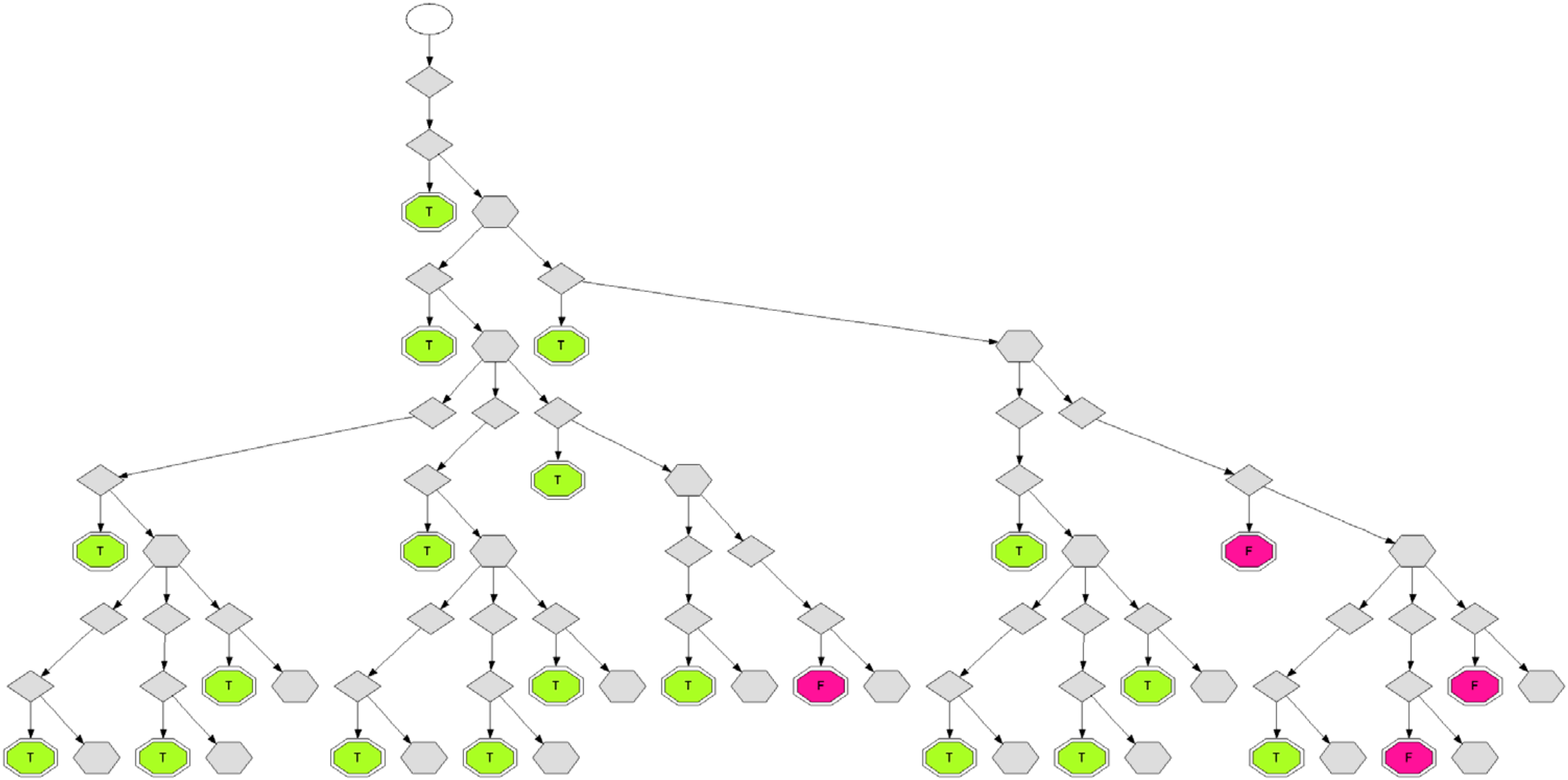
Protocol Dragon Incorrect (-nolca -bol -l2)



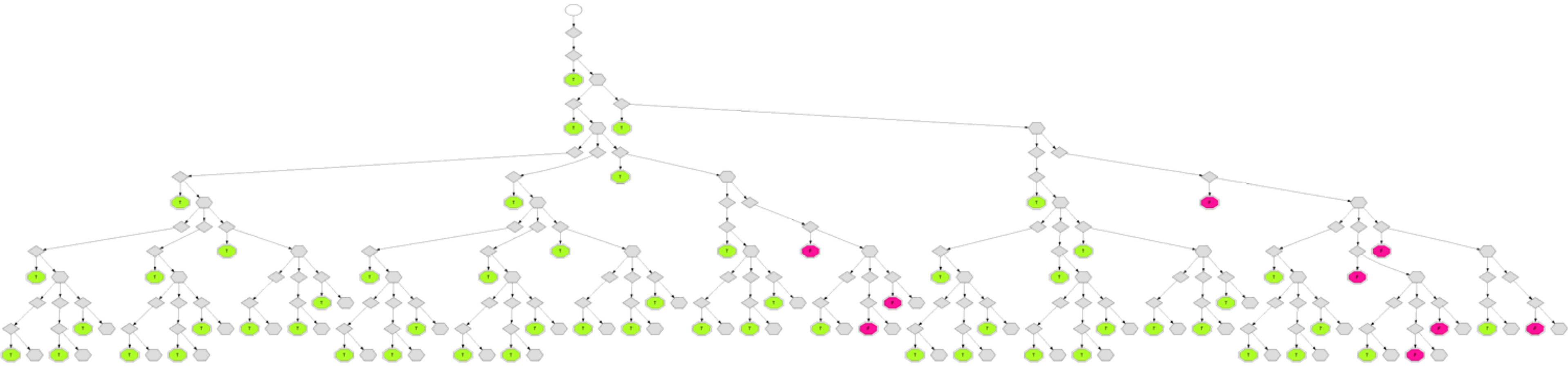
Protocol Dragon Incorrect (-nolca -bol -l3)



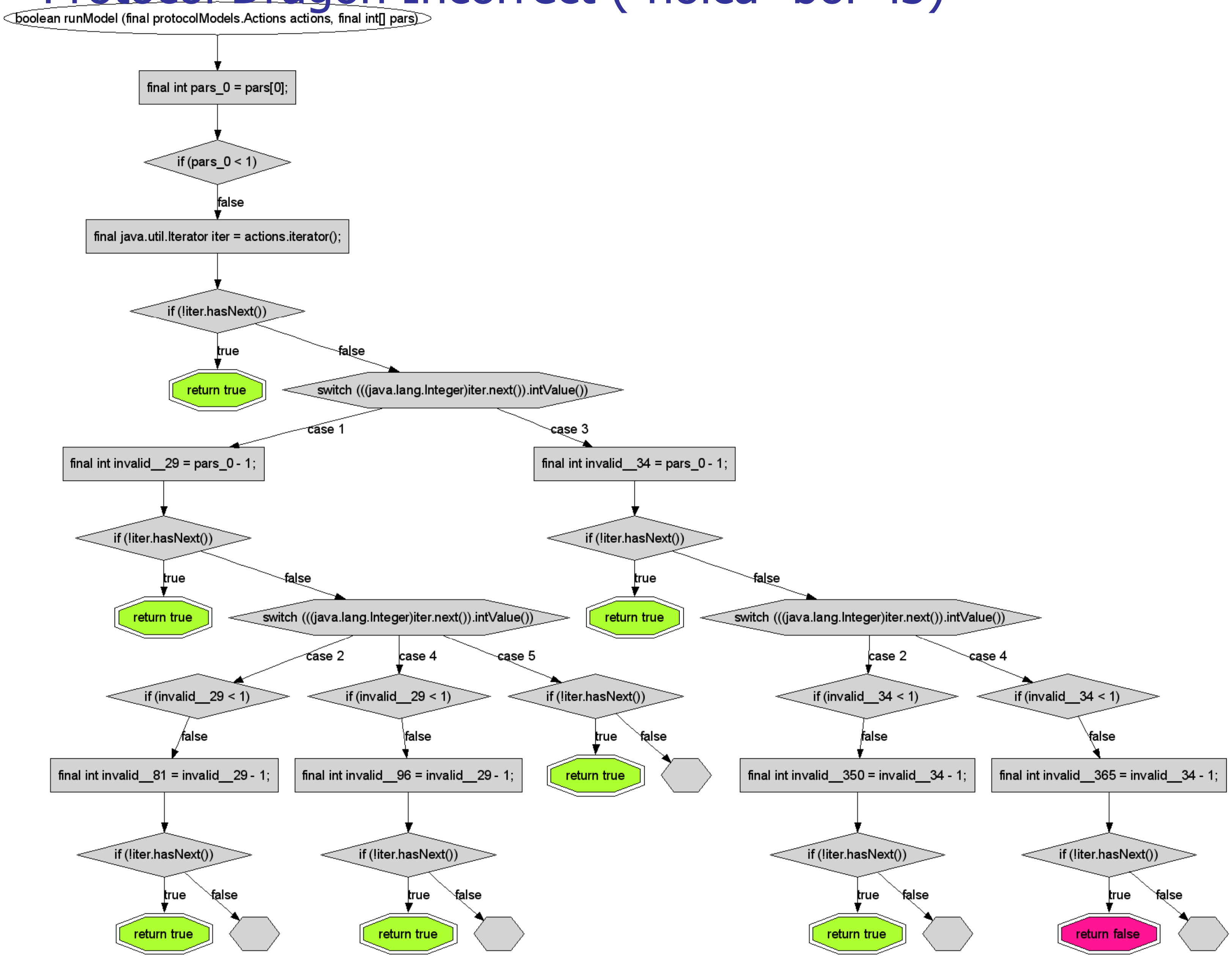
Protocol Dragon Incorrect (-nolca -bol -l4)



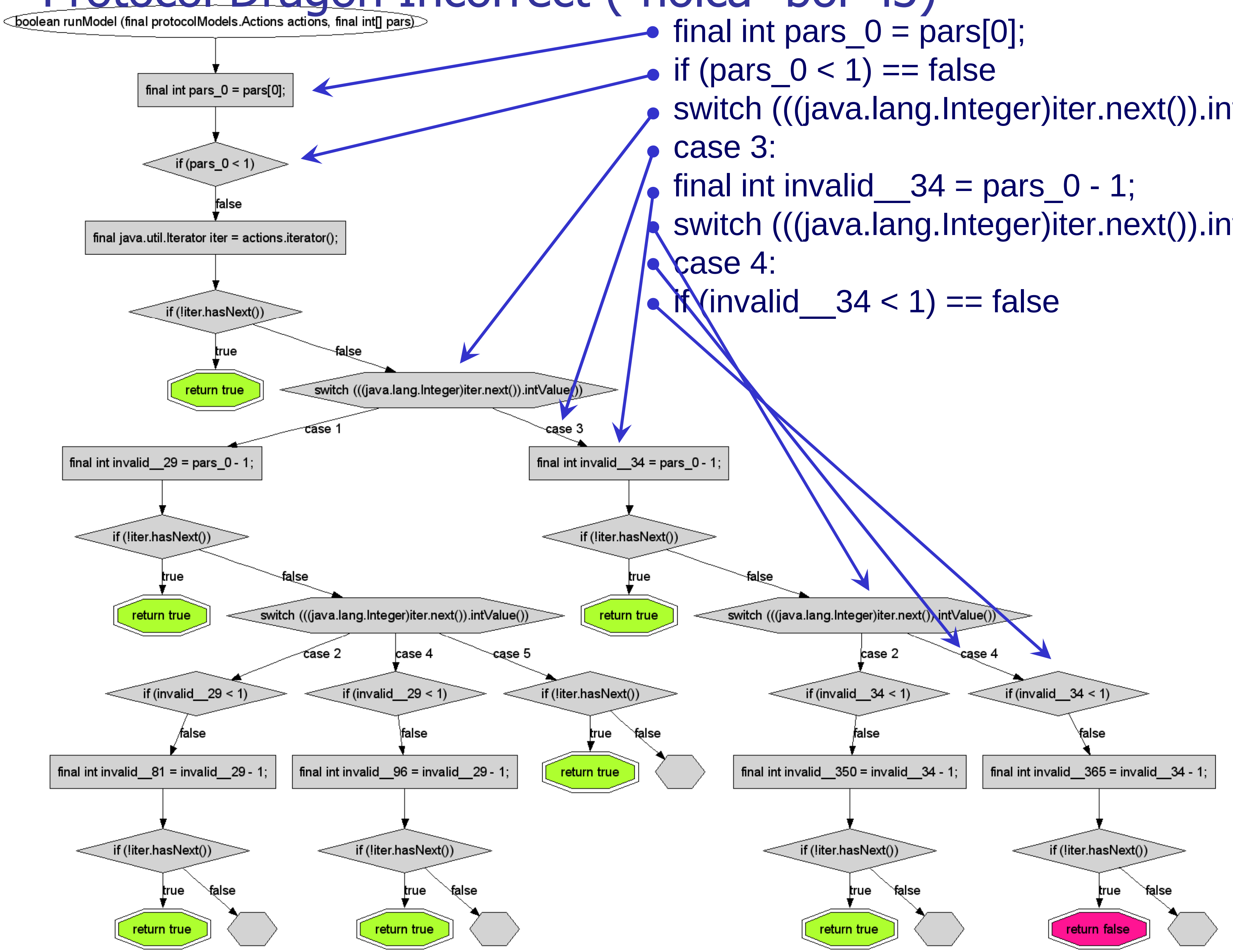
Protocol Dragon Incorrect (-nolca -bol -l5)



Protocol Dragon Incorrect (-nolca -bol -l3)



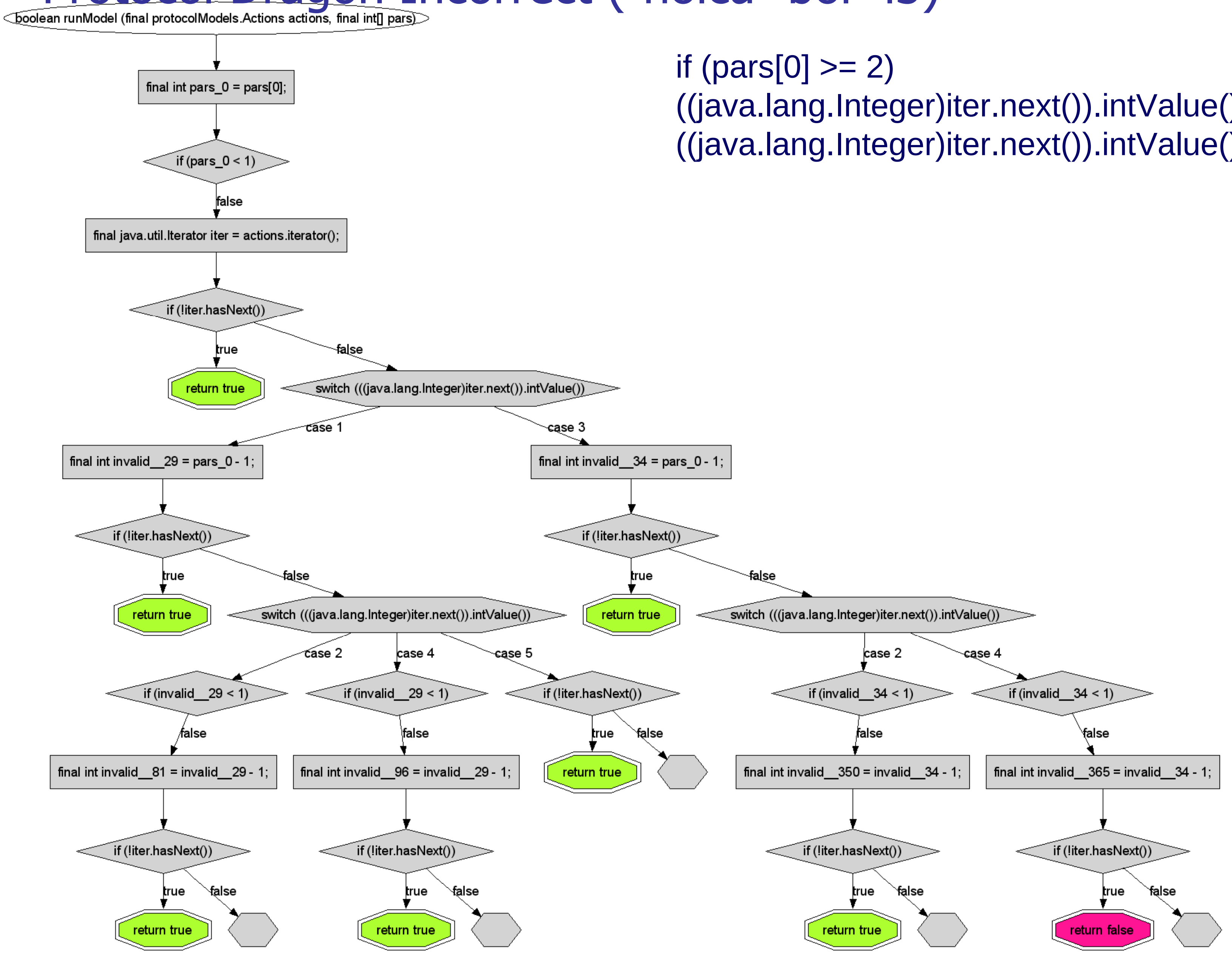
Protocol Dragon Incorrect (-nolca -bol -l3)



- final int pars_0 = pars[0];
- if (pars_0 < 1) == false
- switch (((java.lang.Integer)iter.next()).intValue())
- case 3:
- final int invalid__34 = pars_0 - 1;
- switch (((java.lang.Integer)iter.next()).intValue())
- case 4:
- if (invalid__34 < 1) == false

Protocol Dragon Incorrect (-nolca -bol -l3)

if (pars[0] >= 2)
((java.lang.Integer)iter.next()).intValue() == 3
((java.lang.Integer)iter.next()).intValue() == 4



Short History of Supercompilation

1974	Valentin Turchin presented supercompilation to a group of students at seminars in Moscow
1980s	Valentin Turchin developed first supercompilers for the functional language Refal (CUNY, New York)
1980s – – 1990s	A series of papers by Valentin Turchin on supercompilation of Refal
1990s	Works on supercompilation for simplified languages in Copenhagen University by Robert Glück and Morten Sørensen in collaboration with us
1993 – – 2000s	Andrei Nemytykh (IPS RAS, Pereslavl-Zalessky) continued work on Turchin's supercompiler and completed it
1998 – – 2000s	Java Supercompiler by Andrei Klimov, Arkady Klimov and Artem Shvorin (Keldysh Institute of Applied Mathematics, RAS, Moscow)

Java Supercompiler Project Sites

- Supercompilers, LLC
  - http://supercompilers.com
- JScp Working site
  - http://supercompilers.ru
- JVer Project:
  Verification of Java Programs by means of the JScp Supercompiler
  - http://pat.keldysh.ru/jver