

# Interpretable Verification of Distributed Protocols by Inductive Proof Decomposition

**Will Schultz**  
*Northeastern University*

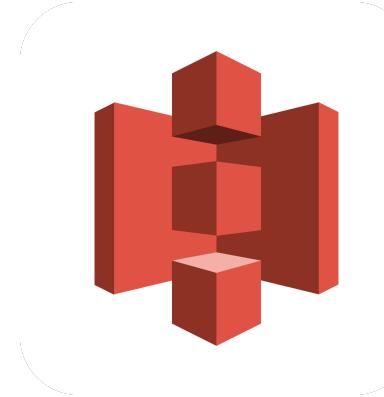
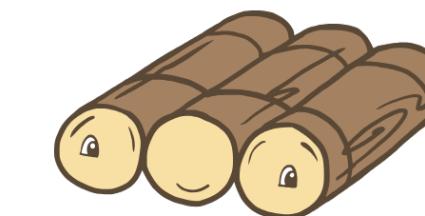
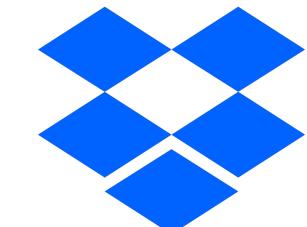


Heidi Howard, Eddy Ashton  
*Microsoft Research*

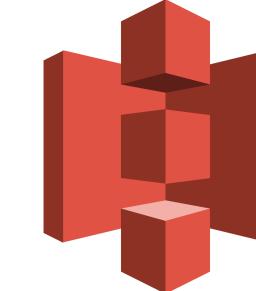
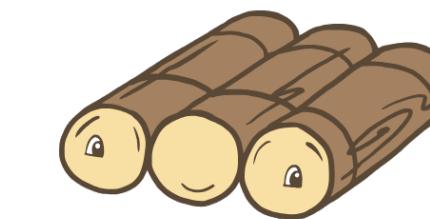
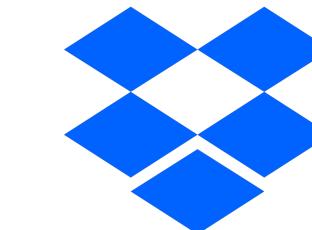


CMU SSSG 2024

# Distributed systems are everywhere.



# Distributed systems are everywhere.



# Distributed systems are hard.

Table of errors			
Protocol	Reference	Violation	Counter-example
PBFT[1]	[Castro and Liskov 1999]	liveness	[Berger et al. 2021]
Chord	[Stoica et al. 2001; Liben-Nowell et al. 2002]	liveness[2]	[Zave 2012; Zave 2017]
Pastry	[Rowstron and Drusel 2001]	safety	[Azmy et al. 2016; Azmy et al. 2018]
Generalised Paxos	[Lamport 2005]	non-triviality[3]	[Sutra and Shapiro 2010]
FaB Paxos	[Martin and Alvisi 2005; Martin and Alvisi 2006]	liveness	[Abraham et al. 2017]
Multi-Paxos[4]	[Chandra et al. 2007]	safety	[Michael et al. 2017]
Zyzzyva	[Kotla et al. 2007; Kotla et al. 2010]	safety	[Abraham et al. 2017]
CRAQ	[Terrace and Freedman 2009]	safety[5]	[Whittaker 2020]
JPaxos	[Kończak et al. 2011]	safety	[Michael et al. 2017]
VR Revisited	[Liskov and Cowling 2012]	safety	[Michael et al. 2017]
EPaxos	[Moraru et al. 2013]	safety	[Sutra 2020]
EPaxos	[Moraru et al. 2013]	safety	[Whittaker 2021]
Raft	[Ongaro and Ousterhout 2014]	liveness[6]	[Hoch 2014]
Raft	[Ongaro 2014]	safety[7]	[Amos and Zhang 2015; Ongaro 2015]
Raft	[Ongaro and Ousterhout 2014; Ongaro 2014]	liveness	[Howard and Abraham 2020; Jensen et al. 2021]
hBFT	[Duan et al. 2015]	safety	[Shrestha et al. 2019]
Tendermint	[Buchman 2016]	liveness	[Cachin and Vukolić 2017]
CAESAR	[Arun et al. 2017]	liveness	[Enes et al. 2021]
DPaxos	[Nawab et al. 2018]	safety	[Whittaker et al. 2021]
Sync HotStuff	[Abraham et al. 2019]	safety & liveness	[Momose and Cruz 2019]
Gasper	[Buterin et al. 2020]	safety & liveness	[Neu et al. 2021]

On the correctness of Egalitarian Paxos

Pierre Sutra

Télécom SudParis  
9, rue Charles Fourier  
91000 Évry, France

**Abstract**

This paper identifies a problem in both the TLA<sup>+</sup> specification and the implementation of the Egalitarian Paxos protocol. It is related to how replicas switch from one ballot to another when computing the dependencies of a command. The problem may lead replicas to diverge and break the linearizability of the replicated service.

bug in single-server membership changes 4974 views

onga...@gmail.com to raft...@googlegroups.com Jul 10, 2021 12:53 AM

Hi raft-dev,

Unfortunately, I need to announce a bug in the dissertation version of membership changes (the single-server changes, not joint consensus). The bug is potentially severe, but the fix I'm proposing is easy to implement.

When Huanchen Zhang and Brandon Amos were working on a class project at CMU to formalize single-server membership changes, Huanchen found the bug and the counter-example below by hand. They contacted me over email on May 14th, and I chose to keep this quiet for a while until we had agreed upon a solution to propose to the list. After several incorrect and/or ugly attempts, I came up with the solution proposed below. I apologize for keeping this information from you for so long.

# Need for formal models and verification.

DOI:10.1145/2699417

**Engineers use TLA+ to prevent serious but subtle bugs from reaching production.**

BY CHRIS NEWCOMBE, TIM RATH, FAN ZHANG, BOGDAN MUNTEANU, MARC BROOKER, AND MICHAEL DEARDEUFF

## How Amazon Web Services Uses Formal Methods

S3 is just one of many AWS services that store and process data our customers have entrusted to us. To safeguard that data, the core of each service relies on fault-tolerant distributed algorithms for replication, consistency, concurrency control, auto-scaling, load balancing, and other coordination tasks. There are many such algorithms in the literature, but combining them into a cohesive system is a challenge, as the algorithms must usually be modified to interact properly in a real-world system. In addition, we have found it necessary to invent algorithms of our own. We work hard to avoid unnecessary complexity, but the essential complexity of the task remains high.

Complexity increases the probability of human error in design, code, and operations. Errors in the core of the system could cause loss or corruption of data, or violate other interface contracts on which our customers depend. So, before launching a service, we need to reach extremely high con-

**Planning for Change in a Formal Verification of the Raft Consensus Protocol**

Doug Woos James R. Wilcox Steve Anton  
Zachary Tatlock Michael D. Ernst Thomas Anderson  
University of Washington, USA  
{dwoos, jrw12, santon, ztatlock, mernst, tom}@cs.washington.edu

**Abstract**  
We present the first formal verification of the Raft consensus protocol, a critical component of many distributed systems. We connected our proof end-to-end guarantee that our implemen-

**IronFleet: Proving Practical Distributed Systems Correct**

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch,  
Bryan Parno, Michael L. Roberts, Srinath Setty, Brian Zill  
Microsoft Research

**Abstract**  
Distributed systems are notorious for harboring subtle bugs. Verification can, in principle, eliminate these bugs a priori.

This paper presents IronFleet, the first methodology for automated machine-checked verification of the safety and liveness of non-trivial distributed system implementations. The key idea behind IronFleet is to verify the correctness of a distributed system by verifying its implementation against a formal specification of its behavior. The specification is written in a formal language called TLA+, which is designed for specifying distributed systems. The verification process involves two main steps: first, the specification is translated into a form that can be checked by a computer; second, the computer checks the specification against the implementation. The verification process is automated, so it can be used to verify large-scale distributed systems.

**Finding Invariants of Distributed Systems: It's a Small (Enough) World After All**

Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno  
Carnegie Mellon University

**Abstract**  
Today's distributed systems are increasingly complex, leading to subtle bugs that are difficult to detect with standard testing methods. Formal verification can provably rule out such bugs, but it is often prohibitively expensive. To reduce this cost, some work has developed specialized languages that either restrict the kinds of systems that can be encoded [12, 13, 36, 53] (e.g., the protocol must proceed in synchronous rounds), or restrict the language used to describe the systems' properties [42]. In exchange for these restrictions,

# Formal Safety Verification

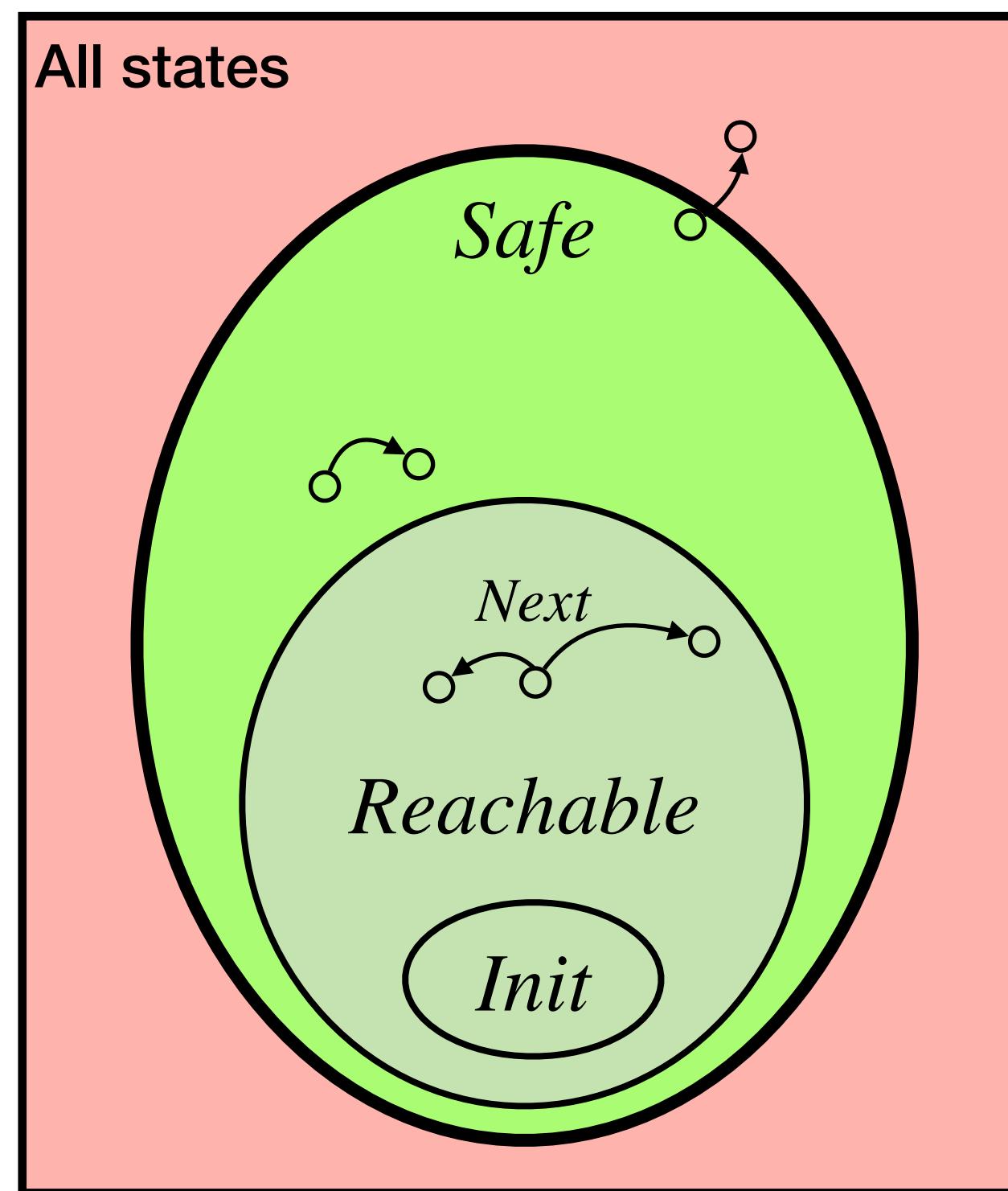
Model protocols as *state transition systems*  $M = (Init, Next)$ .

Initial states.

$$\begin{aligned}Init &\triangleq x = 0 \\Next &\triangleq x' = x + 1\end{aligned}$$

Allowed transitions.

**Verification Goal:** prove that *Safe* is an invariant of  $M$ .



# Formal Safety Verification

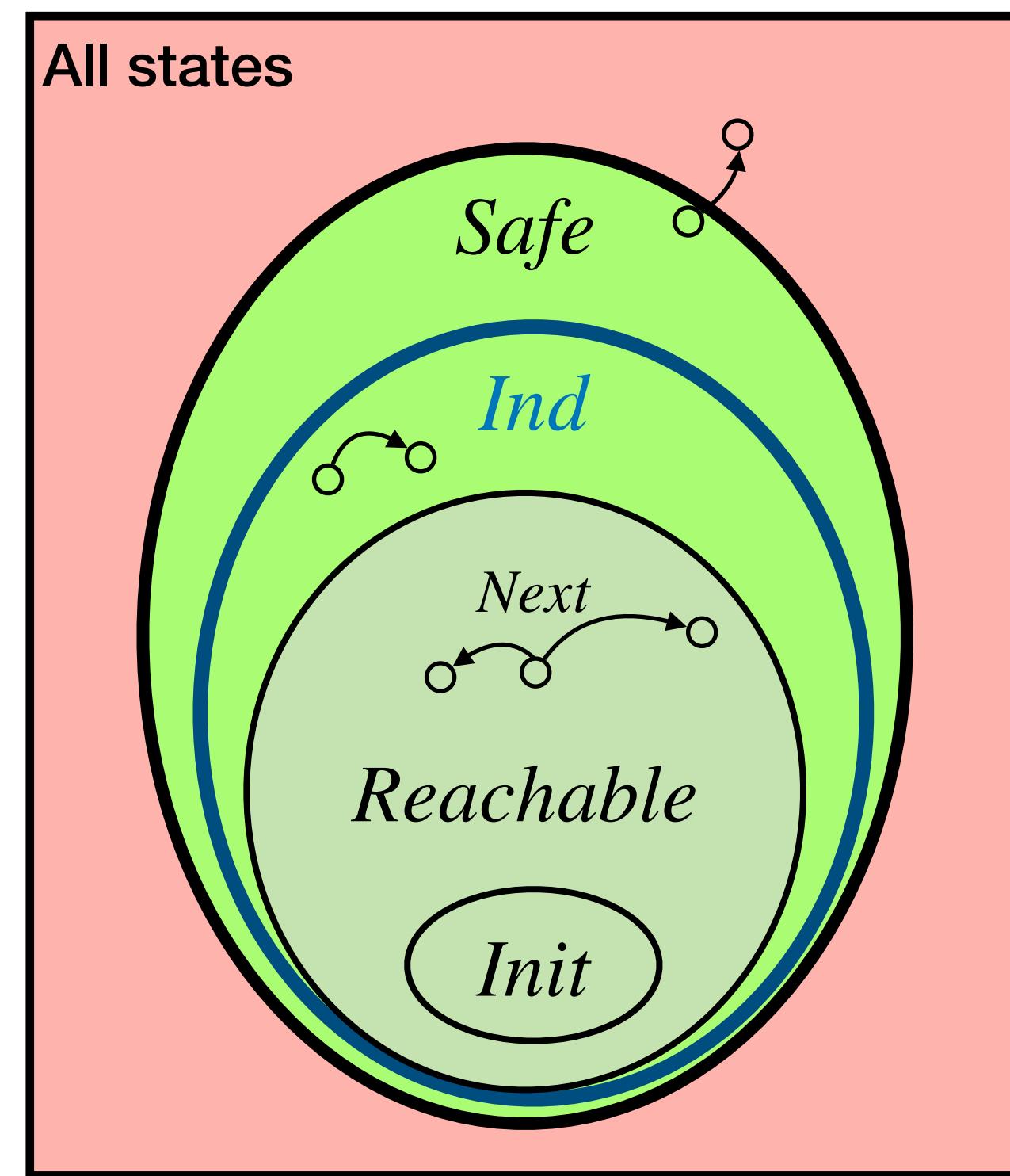
Model protocols as *state transition systems*  $M = (Init, Next)$ .

Initial states.

$$\begin{aligned}Init &\triangleq x = 0 \\Next &\triangleq x' = x + 1\end{aligned}$$

Allowed transitions.

**Verification Goal:** prove that *Safe* is an invariant of  $M$ .



To prove that *Safe* is an invariant of  $M$ , find an **inductive invariant**,

$$Ind = Safe \wedge L_1 \wedge \dots \wedge L_k,$$

*Lemma invariants*

satisfying:

$$Init \Rightarrow Ind$$

$$Ind \wedge Next \Rightarrow Ind'$$

*Induction rule*

# Formal Safety Verification

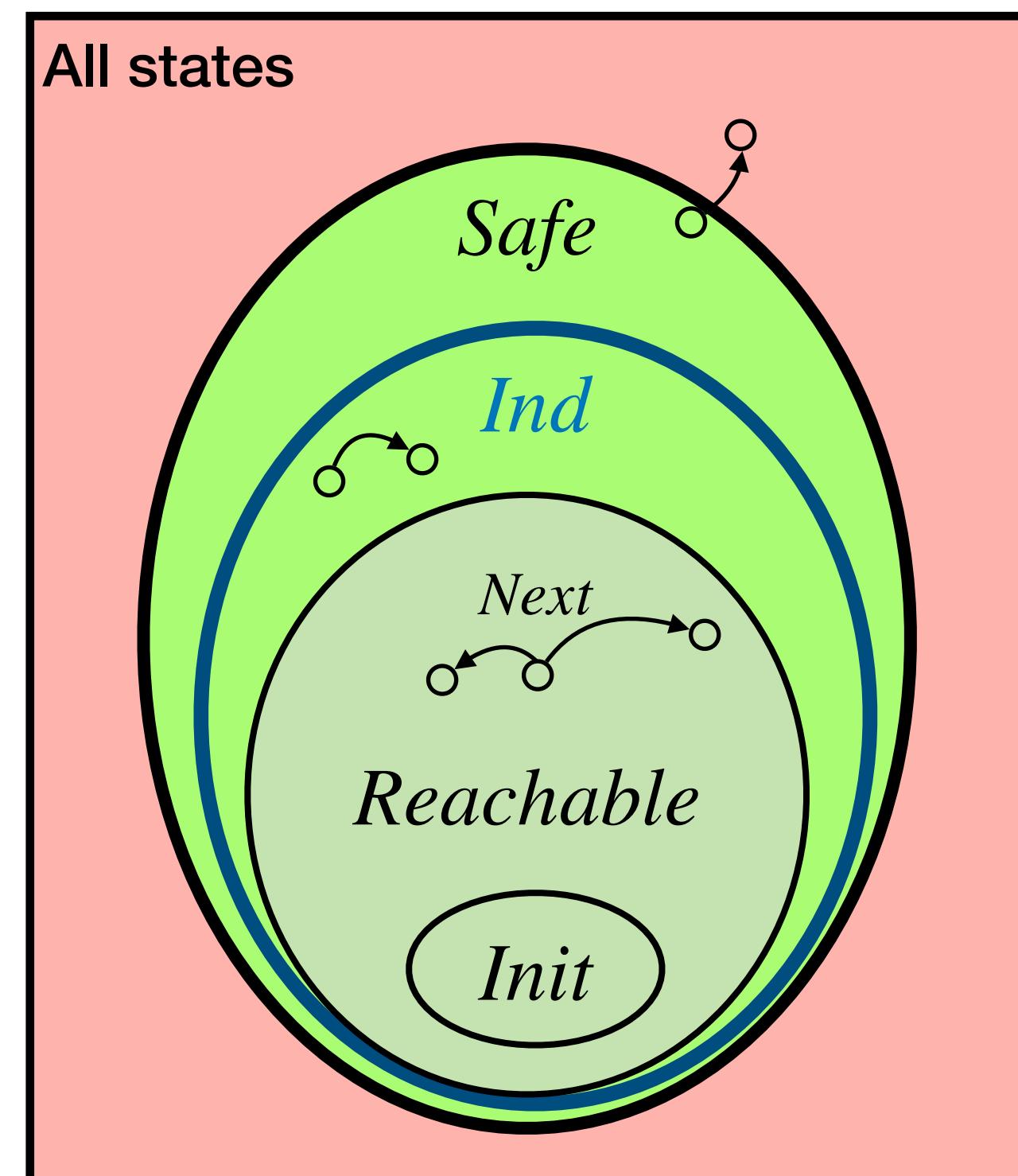
Model protocols as *state transition systems*  $M = (Init, Next)$ .

Initial states.

$$\begin{aligned}Init &\triangleq x = 0 \\Next &\triangleq x' = x + 1\end{aligned}$$

Allowed transitions.

**Verification Goal:** prove that *Safe* is an invariant of  $M$ .



To prove that *Safe* is an invariant of  $M$ , find an **inductive invariant**,

$$Ind = Safe \wedge L_1 \wedge \dots \wedge L_k,$$

*Lemma invariants*

satisfying:

$$Init \Rightarrow Ind$$

$$Ind \wedge Next \Rightarrow Ind'$$

*Induction rule*

But we need to find *Ind*.  
**(HARD)**

# Automatic Safety Verification

Recent protocol verification techniques try to solve this **inductive invariant synthesis** problem automatically.

**IC3PO** [FMCAD21]

**DistAI** [OSDI21, OSDI22]

**endive** [FMCAD22] (ours)

**SWISS** [NSDI21]

# Automatic Safety Verification

Recent protocol verification techniques try to solve this **inductive invariant synthesis** problem automatically.

Distributed Protocol	Domains Relations	Variables	Refinements	DistAI time(s)	I4 time(s)	FOL-IC3 time(s)
		Literals	Invariants	final	total	✓ default
asynchronous lock server [5]	2 5	3 2	0	12	1.1	generalize fail <sup>s</sup> 6.9 -
chord ring maintenance [24]	1 8	3 4	48	163	52.8 586.1 <sup>t</sup> 594.4	- -
database chain replication [24]	4 13	7 4	158	66	58.8 20.2 <sup>t</sup> 63.1	- Z3 fail
decentralized lock [9]	2 2	4 2	150	16	9.4	generalize fail <sup>s</sup> 37.1 <sup>d</sup> Z3 fail
distributed lock [24]	2 4	4 3	82	45	12.6 152.1 <sup>t</sup> 204.7	1451.3 Z3 fail
hashed sharding [11]	3 3	5 2	0	15	1.1	nondet fail <sup>s</sup> 9.2 -
leader election [24]	2 3	6 3	0	17	1.9 4.9 <sup>t</sup> 4.9	26.3 -
learning switch [24]	2 4	4 3	8	71	27.6 10.5 <sup>t</sup> 12.4	- -
lock server [24]	2 2	2 2	0	1	0.8 0.5 <sup>t</sup> 0.8	0.5 2.1
Paxos [13, 15, 23]	4 9	- -	-	-	- * - * - * - *	- -
permissioned blockchain [17]	4 10	6 3	2	13	4.9 blackbox fail <sup>s</sup>	21.2 -
Ricart-Agrawala [26]	1 3	2 2	0	6	0.9 0.8 0.8	0.7 3.2
simple consensus [11]	3 8	5 3	19	50	23.3 41.8 68.7	- -
two-phase commit [18]	1 7	2 3	3	30	1.9 3.1 <sup>t</sup> 8.0	3.4 7.9

<sup>\*</sup> Time out after 1 week.  
<sup>t</sup> I4 runtimes on our machine are similar (6 out of 7 protocols slightly faster) to those previously reported for I4 [18].  
<sup>s</sup> "generalize fail" means I4's implementation fails to convert invariants from the AVR model checker to generalized universally quantified invariants.  
<sup>d</sup> "nondet fail" means failed on nondeterministic initialization. "blackbox fail" means error triggered on reasoning of blackbox functions.  
<sup>d</sup> FOL-IC3 initially completed in less than a second, but this turned out to be incorrect due to a bug in the mypyv protocol specification used by FOL-IC3, which does not exist in the Ivy protocol specification used by DistAI and I4.

Table 1: Evaluation results on 14 distributed protocols from multiple sources.

**IC3PO** [FMCAD21]

**DistAI** [OSDI21, OSDI22]

**endive** [FMCAD22] (ours)

**SWISS** [NSDI21]



Protocol	S.A.	Time (seconds)						Inv	SMT
		IC3PO	SWISS	fol-ic3	DistAI	I4	UPDR		
epr-paxos	∅	<b>568</b>	15950 <sup>*</sup>	timeout	error	memout	timeout	6	11
epr-flexible_paxos	∅	<b>561</b>	18232 <sup>*</sup>	timeout	error	memout	failure	6	11
epr-multi_paxos	∅	timeout	timeout	timeout	error	memout	timeout	-	12
Voting	∅	<b>64</b>	timeout	timeout	error	memout	timeout	3	3
SimplePaxos	$A_{1-2}$	<b>51</b>	timeout	timeout	error	failure	timeout	5	5
ImplicitPaxos	$A_{1-6}$	<b>2008</b>	timeout	timeout	error	failure	timeout	7	7
Paxos	$A_{1-8}$	<b>98</b>	timeout	timeout	error	failure	timeout	10	10
MultiPaxos	$A_{1-11}$	<b>340</b>	timeout	timeout	error	timeout	timeout	10	10
FlexiblePaxos	$A_{1-11}$	<b>1408</b>	timeout	timeout	error	failure	timeout	10	10

TABLE II: Comparison of IC3PO against other state-of-the-art verifiers  
ORIGINAL problems employ hierarchical strengthening (as detailed in Section VI), while EPR problems do not.

While introducing *SimplePaxos* and *ImplicitPaxos* to get the four-level Paxos hierarchy was quite easy, these intermediate levels were still added manually. It is appealing to explore

**DistAI** [Yao2021]

Towards an Automatic Proof of Lamport's Paxos [Goel2022]

**Problem:** Jagged boundary between *automation* and *interactivity*.

# Our Work

Decompose inductive invariants into **inductive proof graphs**, and use this to guide their development.

## Inductive Proof Decomposition

Safety  $\triangleq \forall rm_i, rm_j \in RM : \neg(\text{rmState}[rm_i] = \text{"aborted"} \wedge \text{rmState}[rm_j] = \text{"committed"})$

$Inv73 \triangleq \forall rm_i \in RM : \forall rm_j \in RM : \neg(\text{rmState}[rm_i] = \text{"committed"}) \vee \neg(\text{rmState}[rm_j] = \text{"working"})$

$Inv11 \triangleq \forall rm_j \in RM : \neg([\text{type} \mapsto \text{"Abort"}] \in \text{msgsAbort}) \vee \neg(\text{rmState}[rm_j] = \text{"committed"})$

$Inv23 \triangleq \forall rm_i \in RM : \neg([\text{type} \mapsto \text{"Commit"}] \in \text{msgsCommit}) \vee \neg(\text{rmState}[rm_i] = \text{"aborted"})$

$Inv2 \triangleq \forall rm_i \in RM : \neg([\text{type} \mapsto \text{"Commit"}] \in \text{msgsCommit}) \vee \neg(\text{rmState}[rm_i] = \text{"working"})$

$Inv1 \triangleq \neg([\text{type} \mapsto \text{"Abort"}] \in \text{msgsAbort}) \vee \neg([\text{type} \mapsto \text{"Commit"}] \in \text{msgsCommit})$

$Inv53 \triangleq \forall rm_i \in RM : \neg(\text{rmState}[rm_i] = \text{"committed"}) \vee \neg(\text{tmState} = \text{"init"})$

$Inv1140 \triangleq \forall rm_i \in RM : (\text{rmState}[rm_i] = \text{"prepared"}) \vee \neg(\text{tmPrepared} = RM) \vee \neg(\text{tmState} = \text{"init"})$

$Inv16 \triangleq \forall rm_i \in RM : \neg(\text{rmState}[rm_i] = \text{"working"}) \vee \neg(\text{tmPrepared} = RM)$

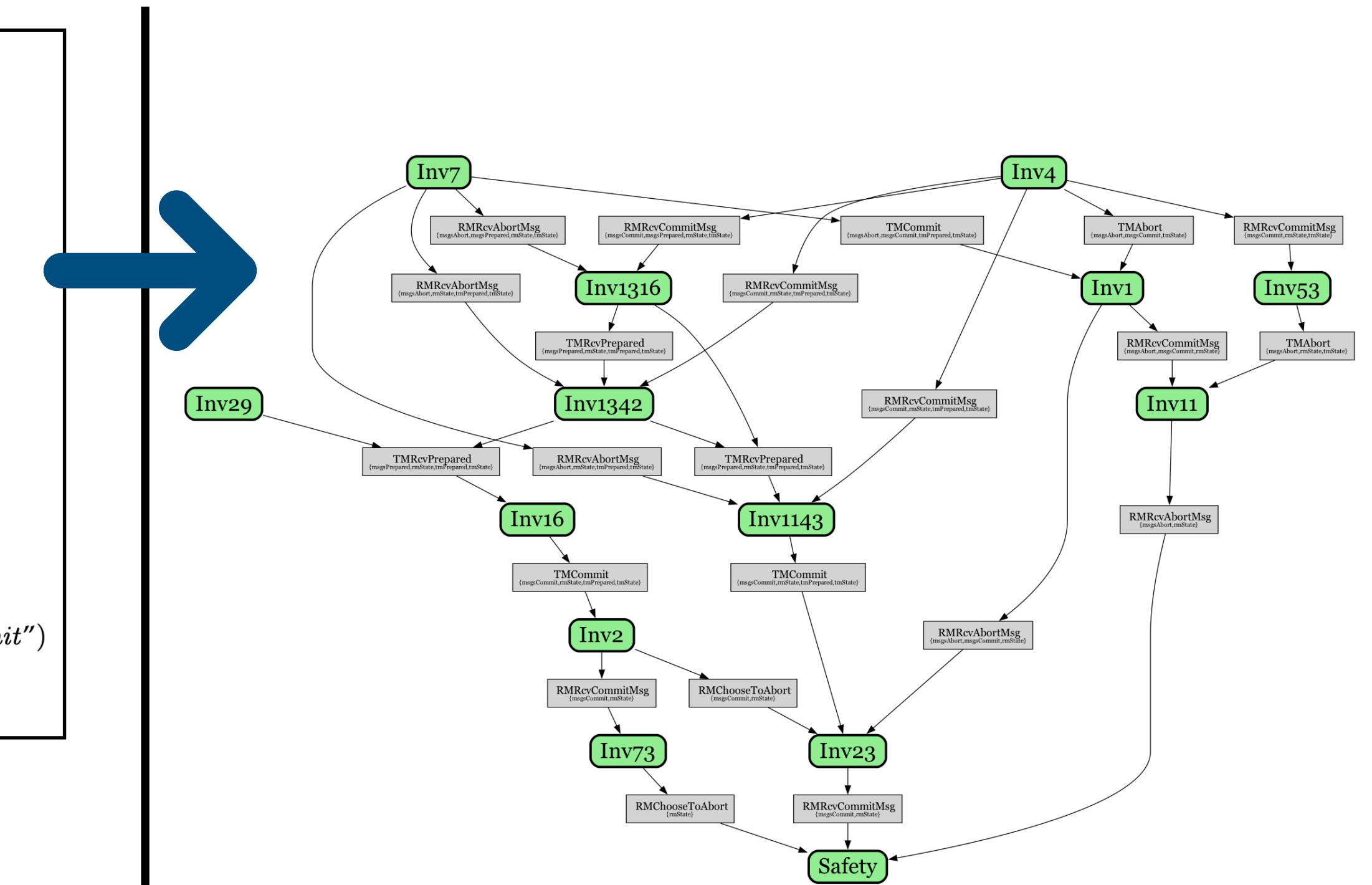
$Inv4 \triangleq \neg([\text{type} \mapsto \text{"Commit"}] \in \text{msgsCommit}) \vee \neg(\text{tmState} = \text{"init"})$

$Inv7 \triangleq \neg([\text{type} \mapsto \text{"Abort"}] \in \text{msgsAbort}) \vee \neg(\text{tmState} = \text{"init"})$

$Inv1325 \triangleq \forall rm_j \in RM : (\text{rmState}[rm_j] = \text{"prepared"}) \vee \neg(\text{tmPrepared} = \text{tmPrepared} \cup \{rm_j\}) \vee \neg(\text{tmState} = \text{"init"})$

$Inv1291 \triangleq \forall rm_j \in RM : (\text{rmState}[rm_j] = \text{"prepared"}) \vee \neg([\text{type} \mapsto \text{"Prepared"}, rm \mapsto rm_j] \in \text{msgsPrepared}) \vee \neg(\text{tmState} = \text{"init"})$

$Inv29 \triangleq \forall rm_i \in RM : \neg([\text{type} \mapsto \text{"Prepared"}, rm \mapsto rm_i] \in \text{msgsPrepared}) \vee \neg(\text{rmState}[rm_i] = \text{"working"})$



Standard, monolithic inductive invariant for  
two-phase commit.

# Decomposing Inductive Invariants

$\text{Safety} \triangleq \forall rm_i, rm_j \in RM : \neg(\text{rmState}[rm_i] = \text{"aborted"} \wedge \text{rmState}[rm_j] = \text{"committed"})$

$\text{Inv73} \triangleq \forall rm_i \in RM : \forall rm_j \in RM : \neg(\text{rmState}[rm_i] = \text{"committed"}) \vee \neg(\text{rmState}[rm_j] = \text{"working"})$

$\text{Inv11} \triangleq \forall rm_j \in RM : \neg([\text{type} \mapsto \text{"Abort"}] \in \text{msgsAbort}) \vee \neg(\text{rmState}[rm_j] = \text{"committed"})$

$\text{Inv23} \triangleq \forall rm_i \in RM : \neg([\text{type} \mapsto \text{"Commit"}] \in \text{msgsCommit}) \vee \neg(\text{rmState}[rm_i] = \text{"aborted"})$

$\text{Inv2} \triangleq \forall rm_i \in RM : \neg([\text{type} \mapsto \text{"Commit"}] \in \text{msgsCommit}) \vee \neg(\text{rmState}[rm_i] = \text{"working"})$

$\text{Inv1} \triangleq \neg([\text{type} \mapsto \text{"Abort"}] \in \text{msgsAbort}) \vee \neg([\text{type} \mapsto \text{"Commit"}] \in \text{msgsCommit})$

$\text{Inv53} \triangleq \forall rm_i \in RM : \neg(\text{rmState}[rm_i] = \text{"committed"}) \vee \neg(\text{tmState} = \text{"init"})$

$\text{Inv1140} \triangleq \forall rm_i \in RM : (\text{rmState}[rm_i] = \text{"prepared"}) \vee \neg(\text{tmPrepared} = RM) \vee \neg(\text{tmState} = \text{"init"})$

$\text{Inv16} \triangleq \forall rm_i \in RM : \neg(\text{rmState}[rm_i] = \text{"working"}) \vee \neg(\text{tmPrepared} = RM)$

$\text{Inv4} \triangleq \neg([\text{type} \mapsto \text{"Commit"}] \in \text{msgsCommit}) \vee \neg(\text{tmState} = \text{"init"})$

$\text{Inv7} \triangleq \neg([\text{type} \mapsto \text{"Abort"}] \in \text{msgsAbort}) \vee \neg(\text{tmState} = \text{"init"})$

$\text{Inv1325} \triangleq \forall rm_j \in RM : (\text{rmState}[rm_j] = \text{"prepared"}) \vee \neg(\text{tmPrepared} = \text{tmPrepared} \cup \{rm_j\}) \vee \neg(\text{tmState} = \text{"init"})$

$\text{Inv1291} \triangleq \forall rm_j \in RM : (\text{rmState}[rm_j] = \text{"prepared"}) \vee \neg([\text{type} \mapsto \text{"Prepared"}, rm \mapsto rm_j] \in \text{msgsPrepared}) \vee \neg(\text{tmState} = \text{"init"})$

$\text{Inv29} \triangleq \forall rm_i \in RM : \neg([\text{type} \mapsto \text{"Prepared"}, rm \mapsto rm_i] \in \text{msgsPrepared}) \vee \neg(\text{rmState}[rm_i] = \text{"working"})$

$\text{Next} \triangleq$

$\vee \text{TMCommitAction}$

$\vee \text{TMAbortAction}$

$\vee \text{TMRcvPreparedAction}$

$\vee \text{RMPrepareAction}$

$\vee \text{RMChooseToAbortAction}$

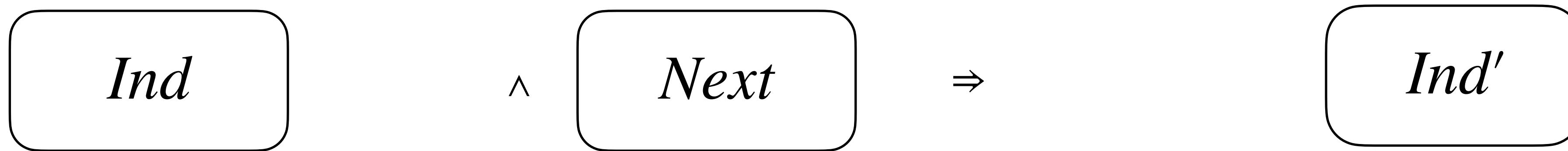
$\vee \text{RMRcvCommitMsgAction}$

$\vee \text{RMRcvAbortMsgAction}$

Inductive invariant and transition relation for two-phase commit protocol.

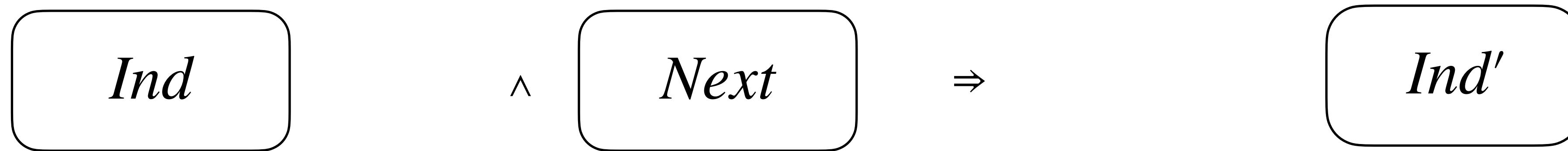
# Decomposing Inductive Invariants

Basic induction verification step.



# Decomposing Inductive Invariants

Basic induction verification step.



(1) Decompose by lemma conjunct.

# Decomposing Inductive Invariants

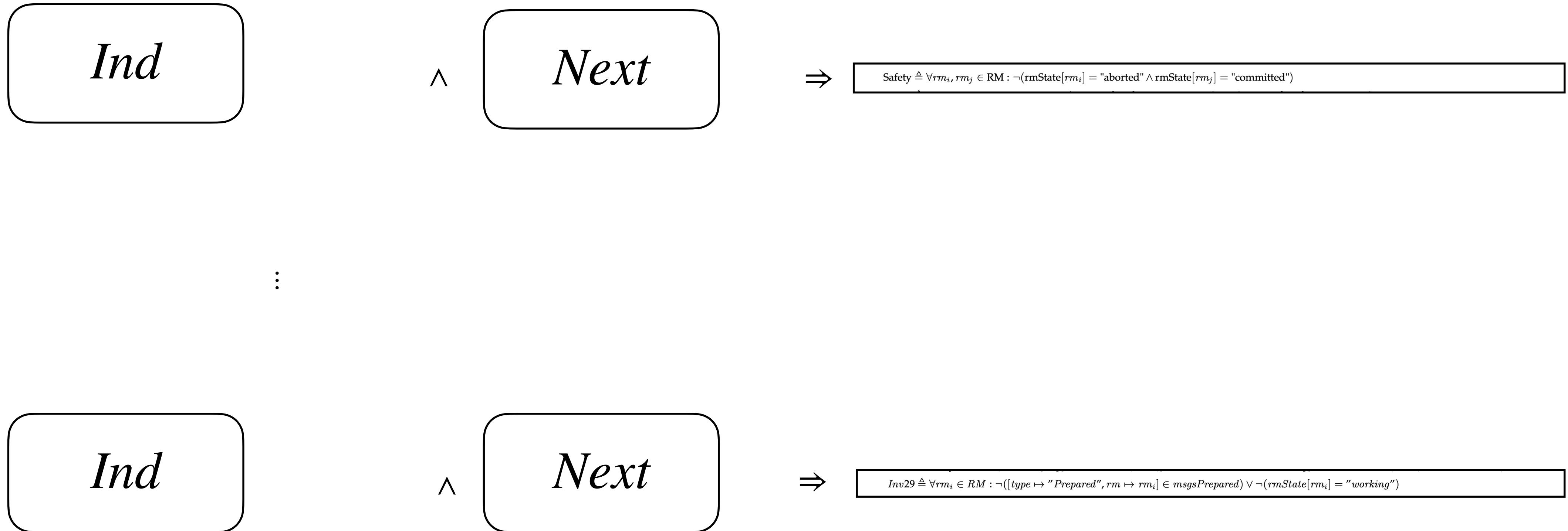
Basic induction verification step.



Safety  $\triangleq \forall rm_i, rm_j \in RM : \neg(\text{rmState}[rm_i] = "aborted" \wedge \text{rmState}[rm_j] = "committed")$   
Inv73  $\triangleq \forall rm_i \in RM : \forall rm_j \in RM : \neg(\text{rmState}[rm_i] = "committed") \vee \neg(\text{rmState}[rm_j] = "working")$   
Inv11  $\triangleq \forall rm_j \in RM : \neg([\text{type} \mapsto "Abort"] \in \text{msgsAbort}) \vee \neg(\text{rmState}[rm_j] = "committed")$   
Inv23  $\triangleq \forall rm_i \in RM : \neg([\text{type} \mapsto "Commit"] \in \text{msgsCommit}) \vee \neg(\text{rmState}[rm_i] = "aborted")$   
Inv2  $\triangleq \forall rm_i \in RM : \neg([\text{type} \mapsto "Commit"] \in \text{msgsCommit}) \vee \neg(\text{rmState}[rm_i] = "working")$   
Inv1  $\triangleq \neg([\text{type} \mapsto "Abort"] \in \text{msgsAbort}) \vee \neg([\text{type} \mapsto "Commit"] \in \text{msgsCommit})$   
Inv53  $\triangleq \forall rm_i \in RM : \neg(\text{rmState}[rm_i] = "committed") \vee \neg(\text{tmState} = "init")$   
Inv1140  $\triangleq \forall rm_i \in RM : (\text{rmState}[rm_i] = "prepared") \vee \neg(\text{tmPrepared} = RM) \vee \neg(\text{tmState} = "init")$   
Inv16  $\triangleq \forall rm_i \in RM : \neg(\text{rmState}[rm_i] = "working") \vee \neg(\text{tmPrepared} = RM)$   
Inv4  $\triangleq \neg([\text{type} \mapsto "Commit"] \in \text{msgsCommit}) \vee \neg(\text{tmState} = "init")$   
Inv7  $\triangleq \neg([\text{type} \mapsto "Abort"] \in \text{msgsAbort}) \vee \neg(\text{tmState} = "init")$   
Inv1325  $\triangleq \forall rm_j \in RM : (\text{rmState}[rm_j] = "prepared") \vee \neg(\text{tmPrepared} = \text{tmPrepared} \cup \{rm_j\}) \vee \neg(\text{tmState} = "init")$   
Inv1291  $\triangleq \forall rm_j \in RM : (\text{rmState}[rm_j] = "prepared") \vee \neg([\text{type} \mapsto "Prepared", rm \mapsto rm_j] \in \text{msgsPrepared}) \vee \neg(\text{tmState} = "init")$   
Inv29  $\triangleq \forall rm_i \in RM : \neg([\text{type} \mapsto "Prepared", rm \mapsto rm_i] \in \text{msgsPrepared}) \vee \neg(\text{rmState}[rm_i] = "working")$

(1) Decompose by lemma conjunct.

# Decomposing Inductive Invariants



**(1) Decompose by lemma conjunct.**

# Decomposing Inductive Invariants

$Safety \triangleq \forall rm_i, rm_j \in RM : \neg(\text{rmState}[rm_i] = \text{"aborted"} \wedge \text{rmState}[rm_j] = \text{"committed"})$   
 $Inv73 \triangleq \forall rm_i \in RM : \forall rm_j \in RM : \neg(\text{rmState}[rm_i] = \text{"committed"}) \vee \neg(\text{rmState}[rm_j] = \text{"working"})$   
 $Inv11 \triangleq \forall rm_j \in RM : \neg([\text{type} \mapsto \text{"Abort"}] \in \text{msgsAbort}) \vee \neg(\text{rmState}[rm_j] = \text{"committed"})$   
 $Inv23 \triangleq \forall rm_i \in RM : \neg([\text{type} \mapsto \text{"Commit"}] \in \text{msgsCommit}) \vee \neg(\text{rmState}[rm_i] = \text{"aborted"})$   
 $Inv2 \triangleq \forall rm_i \in RM : \neg([\text{type} \mapsto \text{"Commit"}] \in \text{msgsCommit}) \vee \neg(\text{rmState}[rm_i] = \text{"working"})$   
 $Inv1 \triangleq \neg([\text{type} \mapsto \text{"Abort"}] \in \text{msgsAbort}) \vee \neg([\text{type} \mapsto \text{"Commit"}] \in \text{msgsCommit})$   
 $Inv53 \triangleq \forall rm_i \in RM : \neg(\text{rmState}[rm_i] = \text{"committed"}) \vee \neg(\text{tmState} = \text{"init"})$   
 $Inv1140 \triangleq \forall rm_i \in RM : (\text{rmState}[rm_i] = \text{"prepared"}) \vee \neg(\text{tmPrepared} = RM) \vee \neg(\text{tmState} = \text{"init"})$   
 $Inv16 \triangleq \forall rm_i \in RM : \neg(\text{rmState}[rm_i] = \text{"working"}) \vee \neg(\text{tmPrepared} = RM)$   
 $Inv4 \triangleq \neg([\text{type} \mapsto \text{"Commit"}] \in \text{msgsCommit}) \vee \neg(\text{tmState} = \text{"init"})$   
 $Inv7 \triangleq \neg([\text{type} \mapsto \text{"Abort"}] \in \text{msgsAbort}) \vee \neg(\text{tmState} = \text{"init"})$   
 $Inv1325 \triangleq \forall rm_j \in RM : (\text{rmState}[rm_j] = \text{"prepared"}) \vee \neg(\text{tmPrepared} = \text{tmPrepared} \cup \{rm_j\}) \vee \neg(\text{tmState} = \text{"init"})$   
 $Inv1291 \triangleq \forall rm_j \in RM : (\text{rmState}[rm_j] = \text{"prepared"}) \vee \neg([\text{type} \mapsto \text{"Prepared"}, \text{rm} \mapsto rm_j] \in \text{msgsPrepared}) \vee \neg(\text{tmState} = \text{"init"})$   
 $Inv29 \triangleq \forall rm_i \in RM : \neg([\text{type} \mapsto \text{"Prepared"}, \text{rm} \mapsto rm_i] \in \text{msgsPrepared}) \vee \neg(\text{rmState}[rm_i] = \text{"working"})$

Next

$\Rightarrow$  Safety  $\triangleq \forall rm_i, rm_j \in RM : \neg(\text{rmState}[rm_i] = \text{"aborted"} \wedge \text{rmState}[rm_j] = \text{"committed"})$

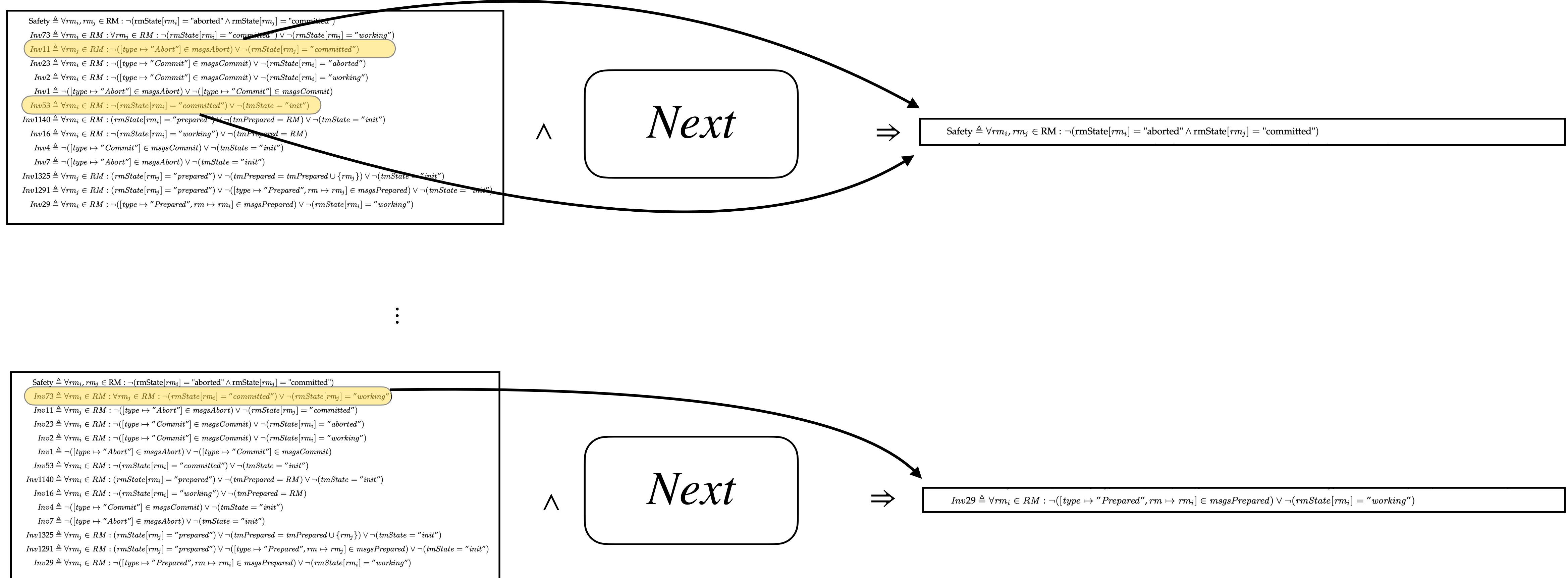
$\text{Safety} \triangleq \forall rm_i, rm_j \in RM : \neg(\text{rmState}[rm_i] = \text{"aborted"} \wedge \text{rmState}[rm_j] = \text{"committed"})$   
 $Inv73 \triangleq \forall rm_i \in RM : \forall rm_j \in RM : \neg(\text{rmState}[rm_i] = \text{"committed"}) \vee \neg(\text{rmState}[rm_j] = \text{"working"})$   
 $Inv11 \triangleq \forall rm_j \in RM : \neg([\text{type} \mapsto \text{"Abort"}] \in \text{msgsAbort}) \vee \neg(\text{rmState}[rm_j] = \text{"committed"})$   
 $Inv23 \triangleq \forall rm_i \in RM : \neg([\text{type} \mapsto \text{"Commit"}] \in \text{msgsCommit}) \vee \neg(\text{rmState}[rm_i] = \text{"aborted"})$   
 $Inv2 \triangleq \forall rm_i \in RM : \neg([\text{type} \mapsto \text{"Commit"}] \in \text{msgsCommit}) \vee \neg(\text{rmState}[rm_i] = \text{"working"})$   
 $Inv1 \triangleq \neg([\text{type} \mapsto \text{"Abort"}] \in \text{msgsAbort}) \vee \neg([\text{type} \mapsto \text{"Commit"}] \in \text{msgsCommit})$   
 $Inv53 \triangleq \forall rm_i \in RM : \neg(\text{rmState}[rm_i] = \text{"committed"}) \vee \neg(\text{tmState} = \text{"init"})$   
 $Inv1140 \triangleq \forall rm_i \in RM : (\text{rmState}[rm_i] = \text{"prepared"}) \vee \neg(\text{tmPrepared} = RM) \vee \neg(\text{tmState} = \text{"init"})$   
 $Inv16 \triangleq \forall rm_i \in RM : \neg(\text{rmState}[rm_i] = \text{"working"}) \vee \neg(\text{tmPrepared} = RM)$   
 $Inv4 \triangleq \neg([\text{type} \mapsto \text{"Commit"}] \in \text{msgsCommit}) \vee \neg(\text{tmState} = \text{"init"})$   
 $Inv7 \triangleq \neg([\text{type} \mapsto \text{"Abort"}] \in \text{msgsAbort}) \vee \neg(\text{tmState} = \text{"init"})$   
 $Inv1325 \triangleq \forall rm_j \in RM : (\text{rmState}[rm_j] = \text{"prepared"}) \vee \neg(\text{tmPrepared} = \text{tmPrepared} \cup \{rm_j\}) \vee \neg(\text{tmState} = \text{"init"})$   
 $Inv1291 \triangleq \forall rm_j \in RM : (\text{rmState}[rm_j] = \text{"prepared"}) \vee \neg([\text{type} \mapsto \text{"Prepared"}, rm \mapsto rm_j] \in \text{msgsPrepared}) \vee \neg(\text{tmState} = \text{"init"})$   
 $Inv29 \triangleq \forall rm_i \in RM : \neg([\text{type} \mapsto \text{"Prepared"}, rm \mapsto rm_i] \in \text{msgsPrepared}) \vee \neg(\text{rmState}[rm_i] = \text{"working"})$

Next

$$\Rightarrow \boxed{Inv29 \triangleq \forall rm_i \in RM : \neg([type \mapsto "Prepared", rm \mapsto rm_i] \in msgsPrepared) \vee \neg(rmState[rm_i] = "working")}$$

# (1) Decompose by lemma conjunct.

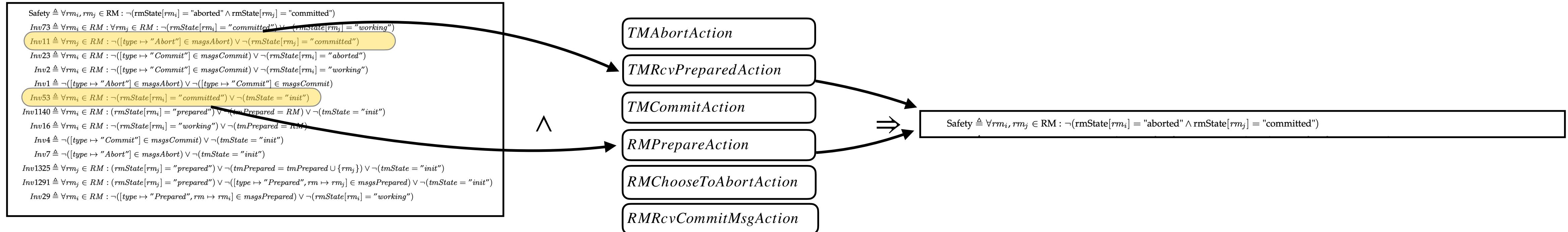
# Decomposing Inductive Invariants



**(2) Find inductive support lemmas.**

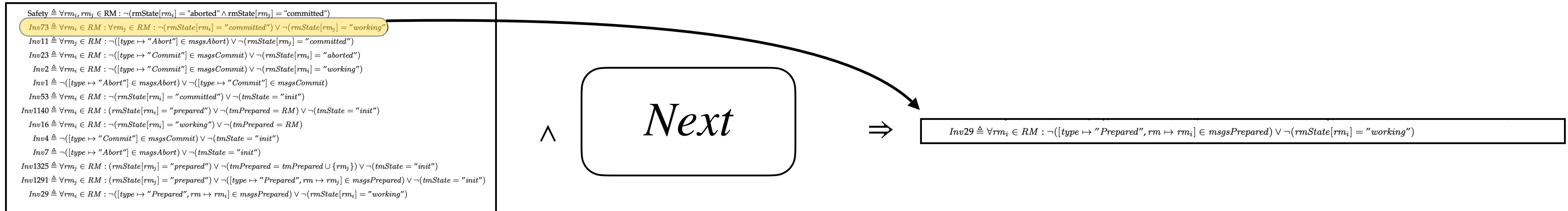
**(1) Decompose by lemma conjunct.**

# Decomposing Inductive Invariants



**(3) Decompose by action disjunct.**

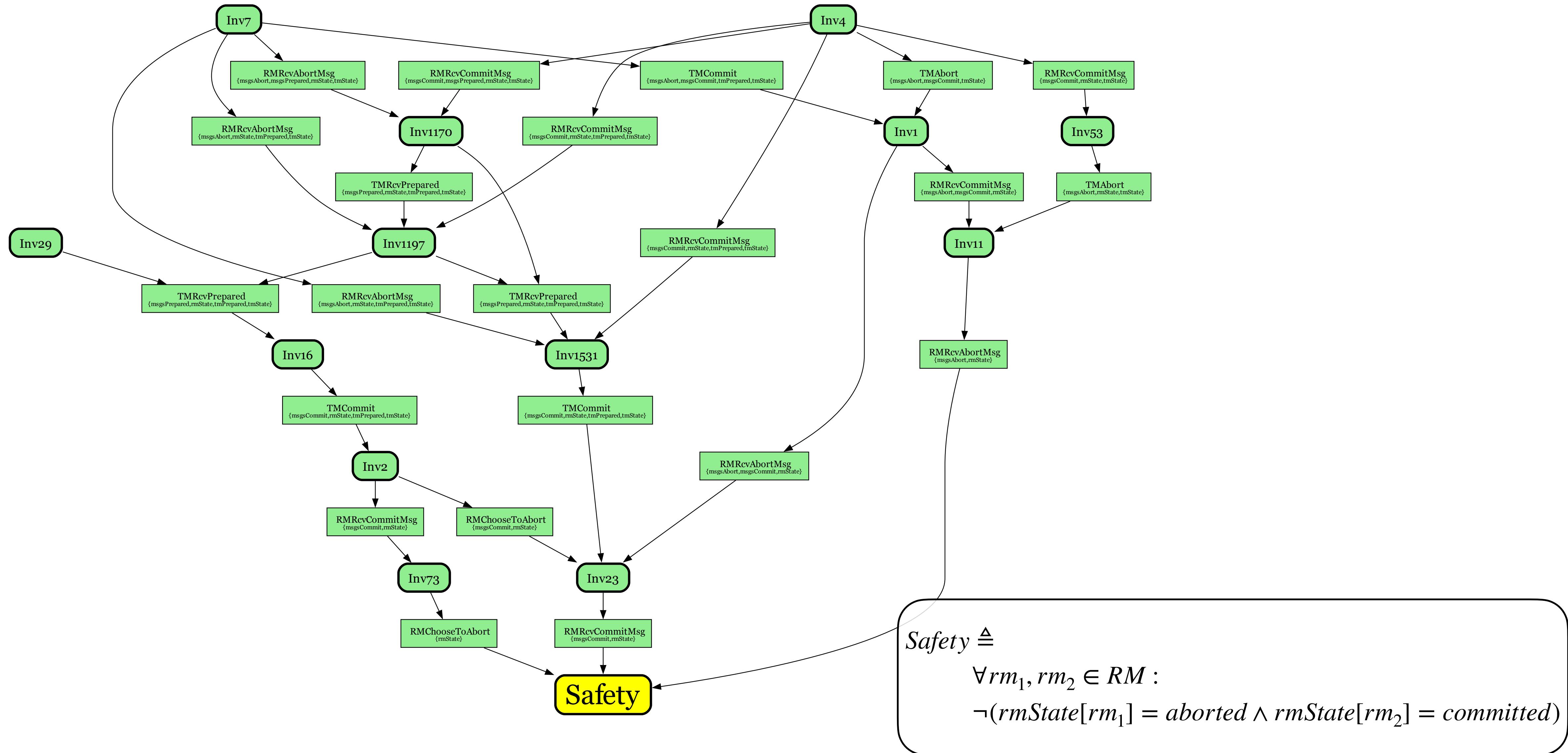
⋮



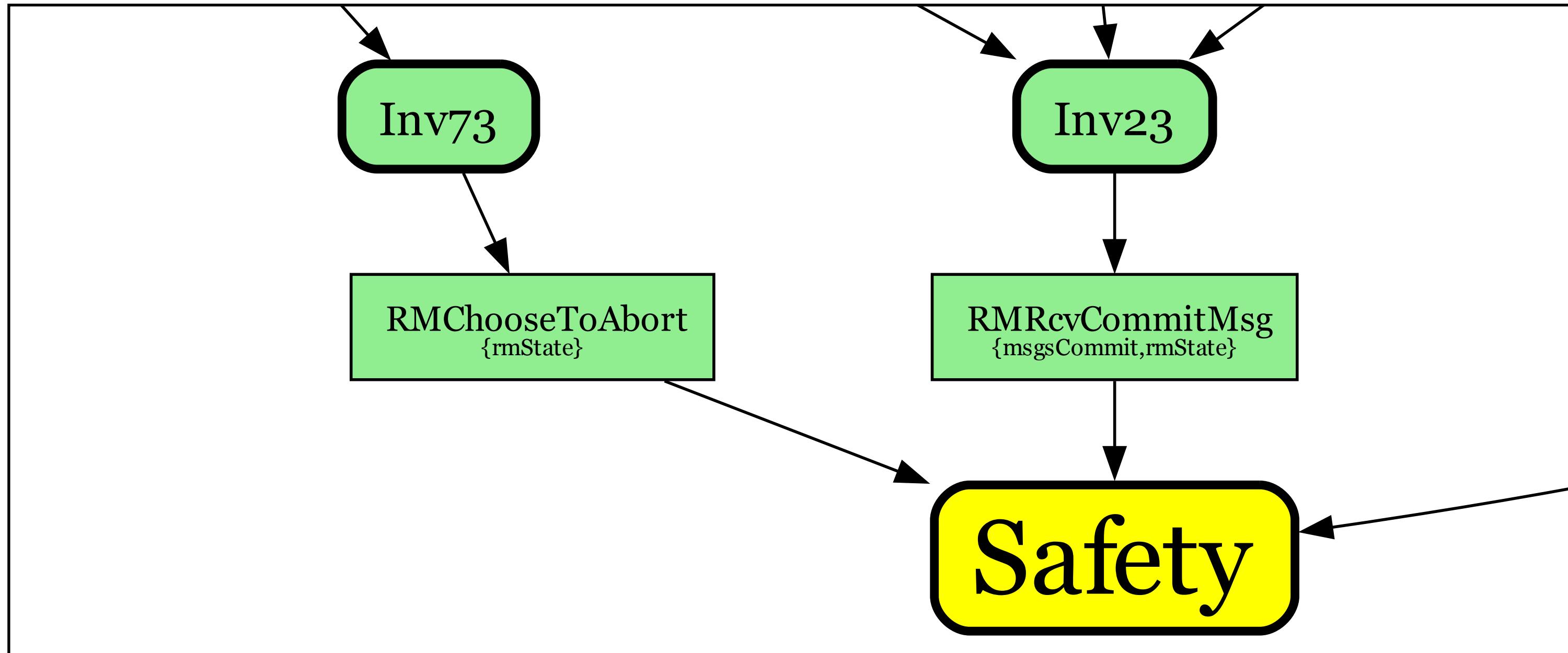
**(2) Find inductive support lemmas.**

**(1) Decompose by lemma conjunct.**

# Inductive Proof Graph: Two-Phase Commit



# Variable Slices



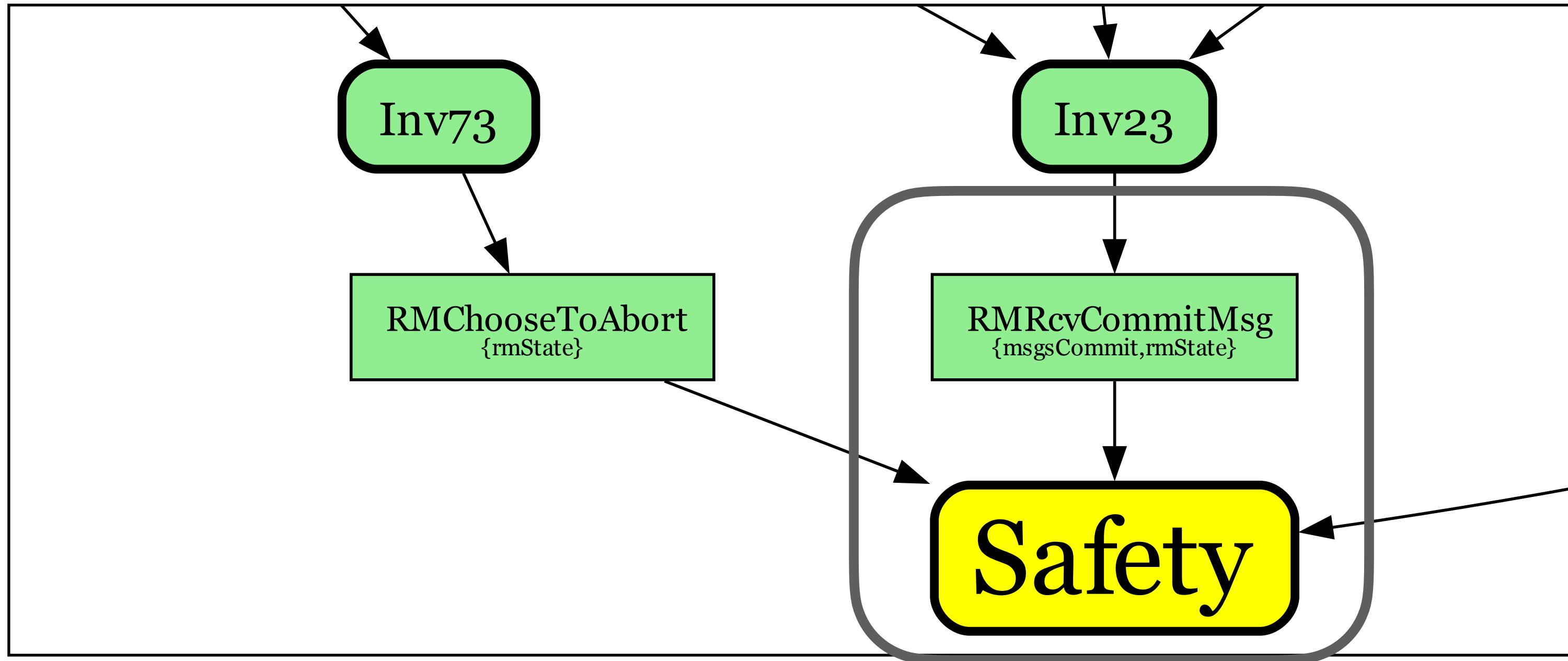
*VARIABLES*  
*rmState,*  
*tmState,*  
*tmPrepared,*  
*msgsPrepared,*  
*msgsCommit,*  
*msgsAbort*

Variable **slices** can be statically computed at each node based on (lemma,action) pair  $(L, A)$ .

Subset of state variables needed for support lemmas to discharge inductive node.

$Safety \triangleq$   
 $\forall rm_1, rm_2 \in RM :$   
 $\neg(rmState[rm_1] = aborted \wedge rmState[rm_2] = committed)$

# Variable Slices

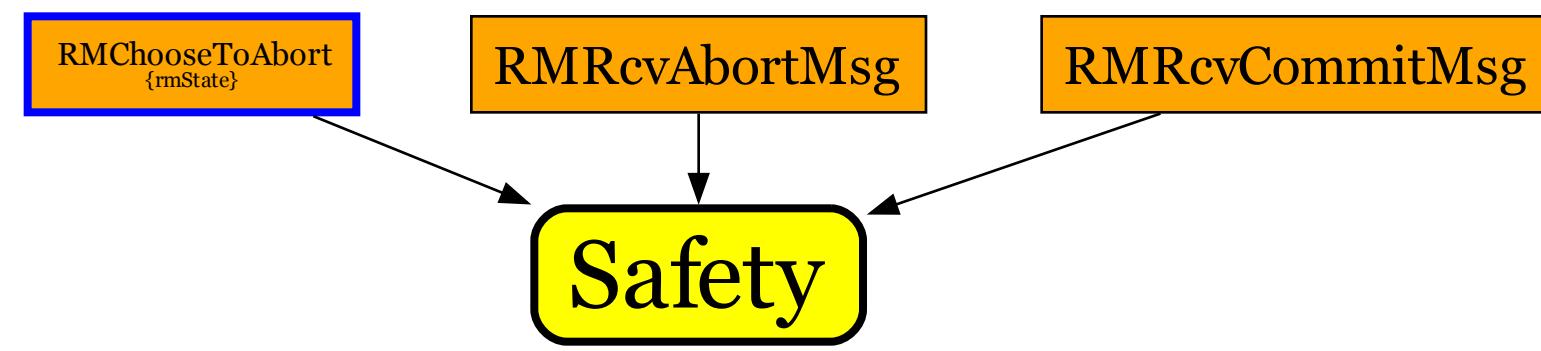


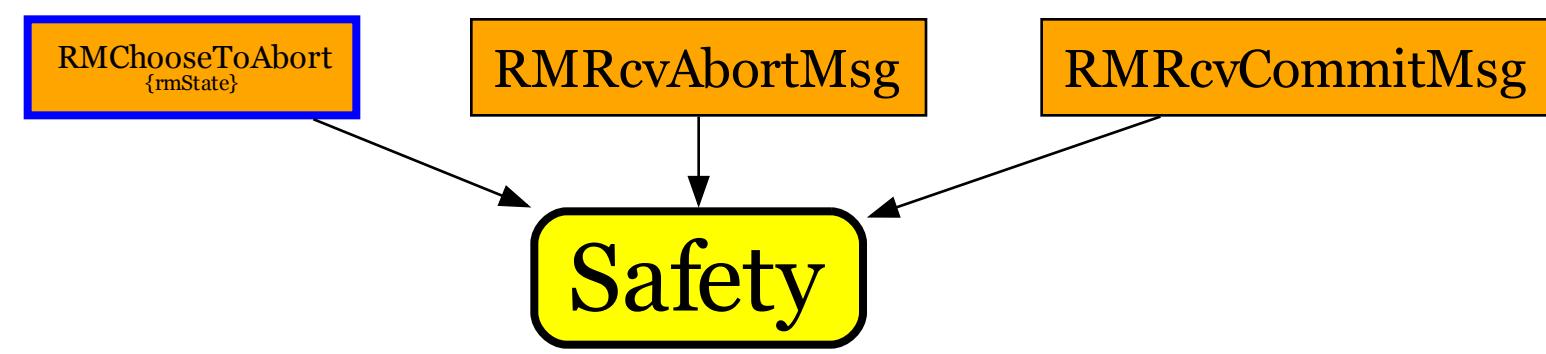
VARIABLES  
*rmState*,  
*tmState*,  
*tmPrepared*,  
*msgsPrepared*,  
*msgsCommit*,  
*msgsAbort*

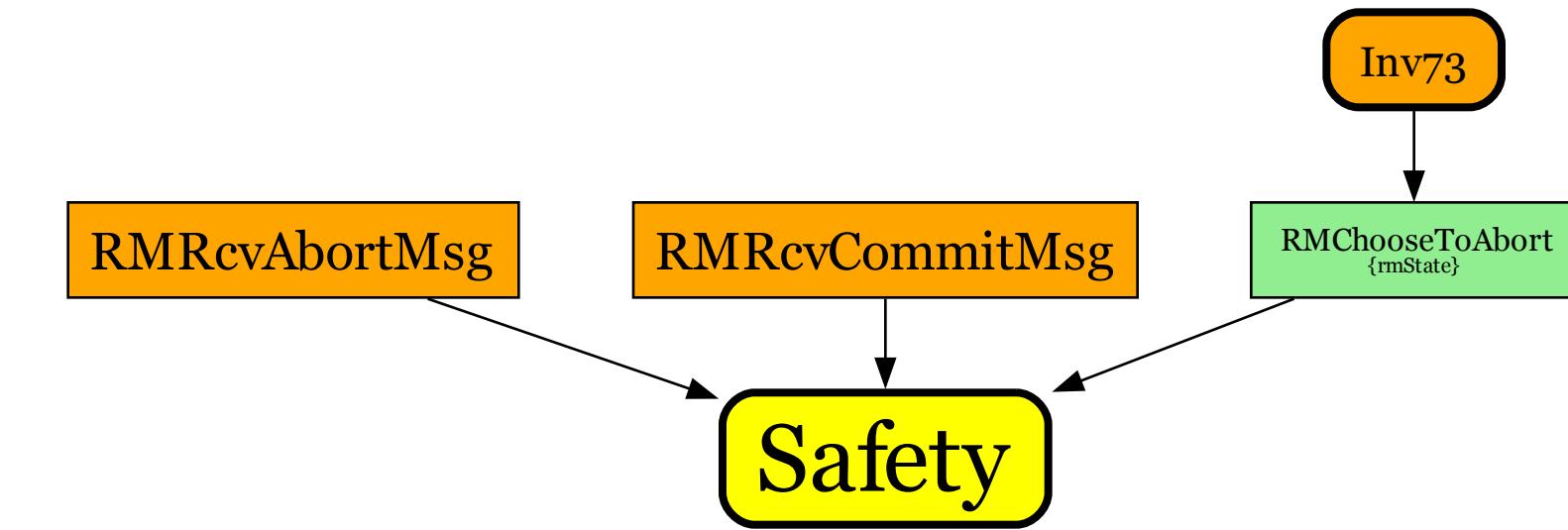
Variable **slices** can be statically computed at each node based on (lemma,action) pair  $(L, A)$ .

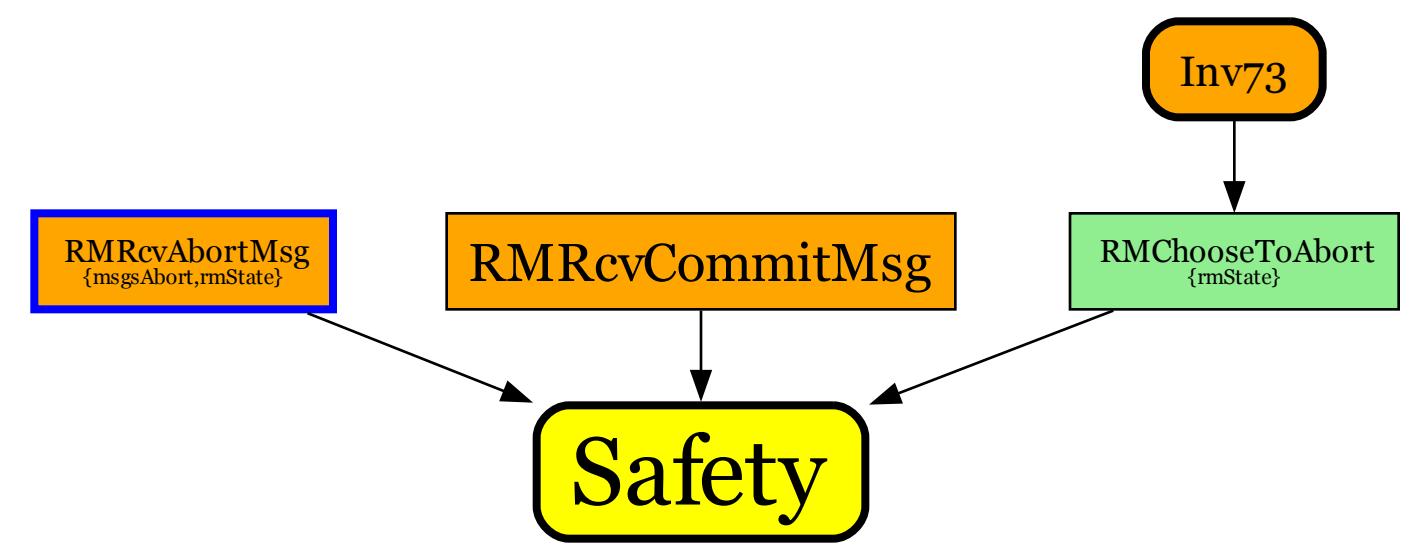
Subset of state variables needed for support lemmas to discharge inductive node.

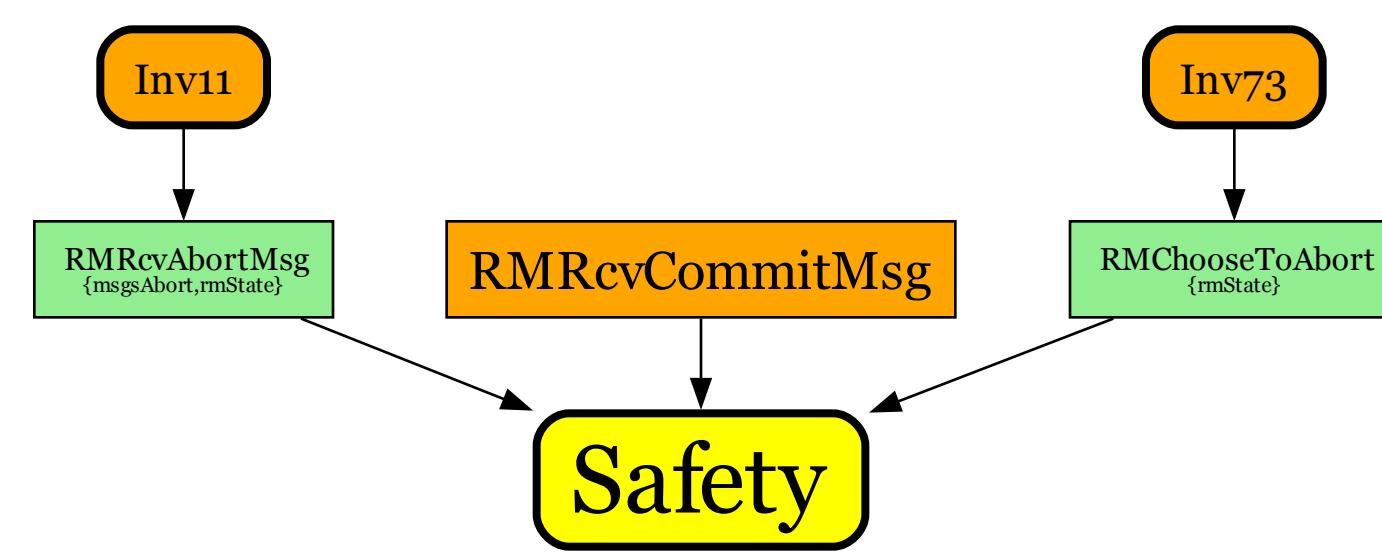
$Safety \triangleq$   
 $\forall rm_1, rm_2 \in RM :$   
 $\neg(rmState[rm_1] = aborted \wedge rmState[rm_2] = committed)$

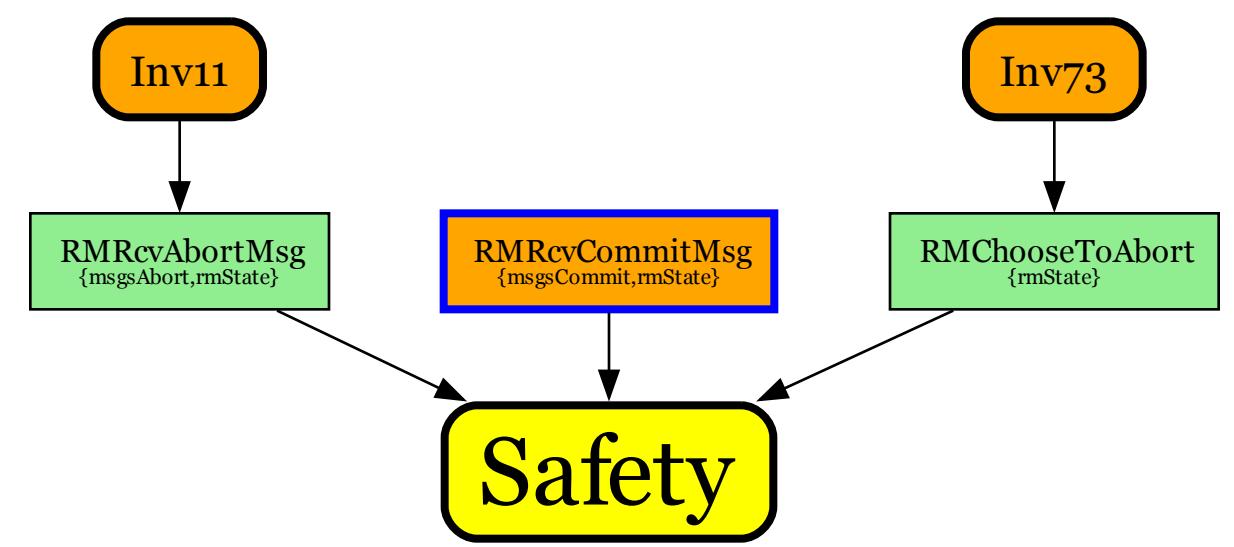


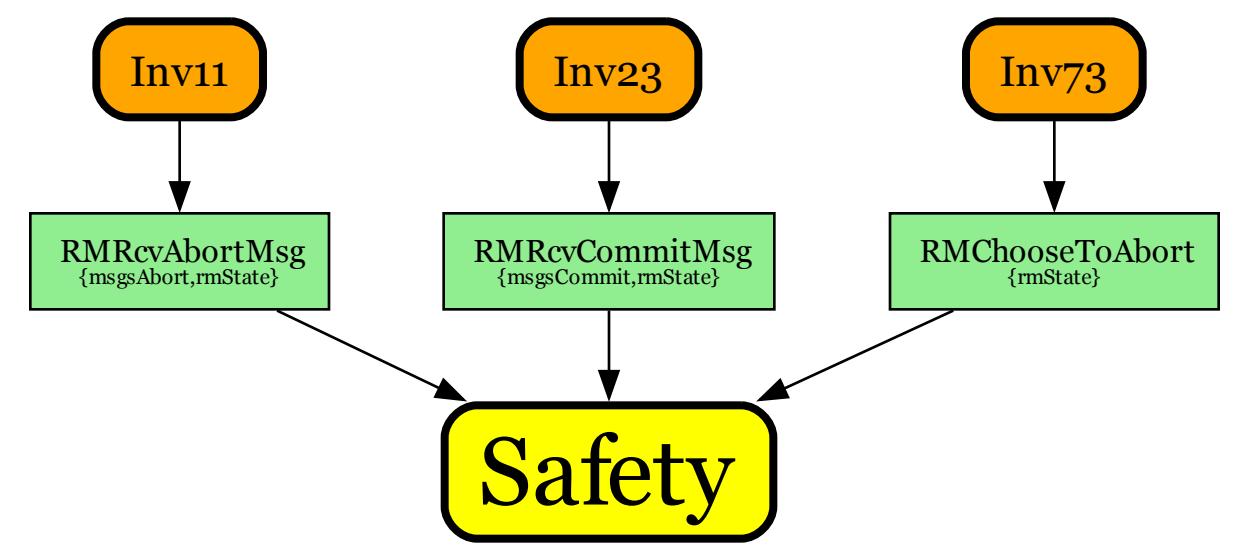


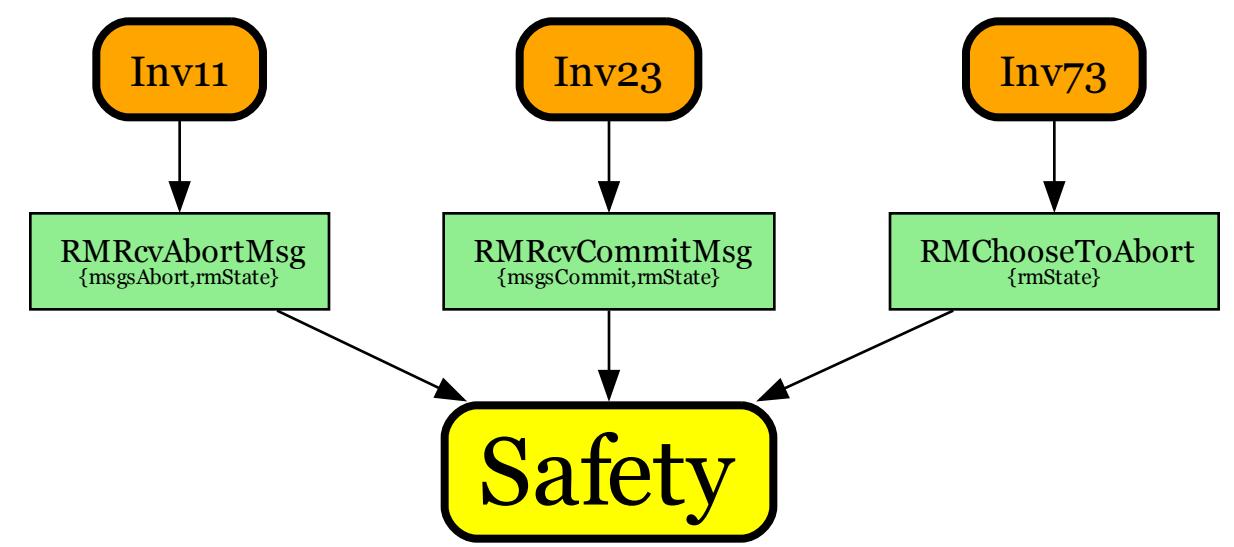


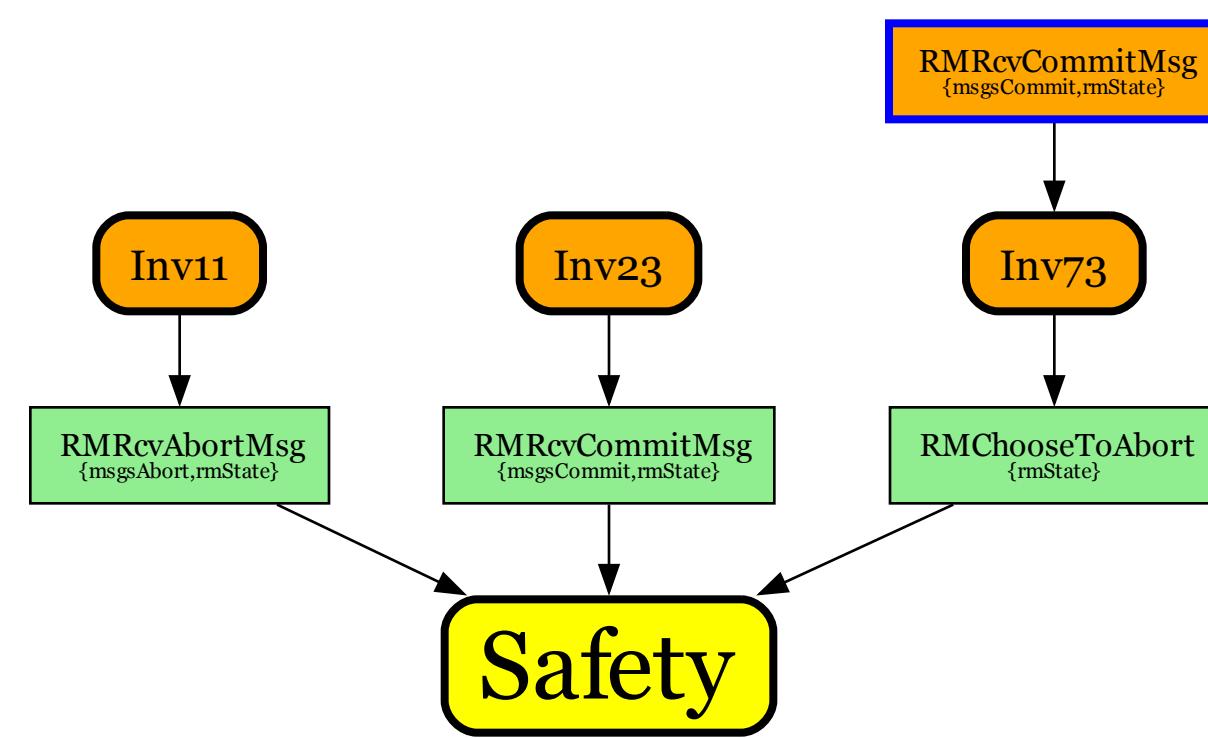


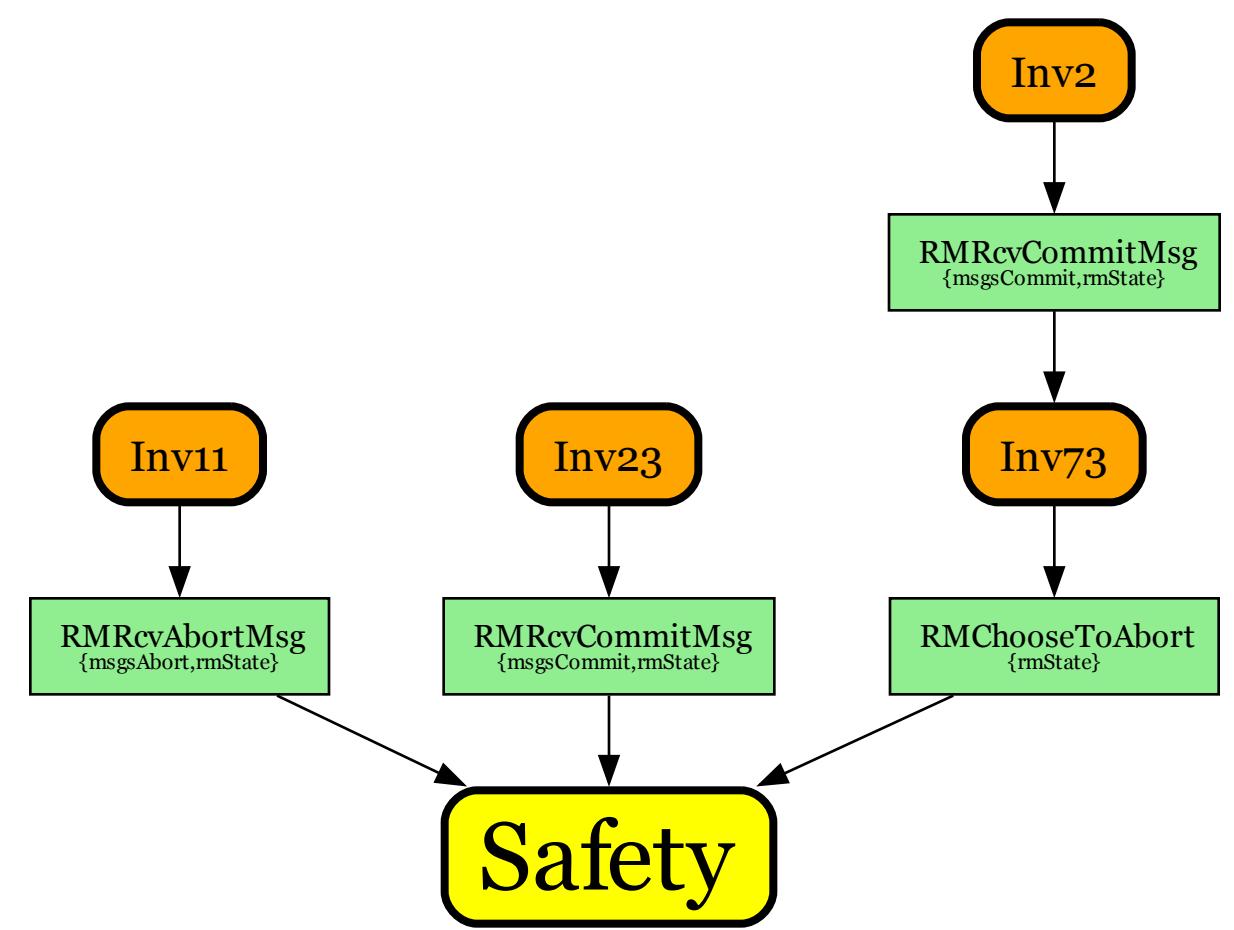


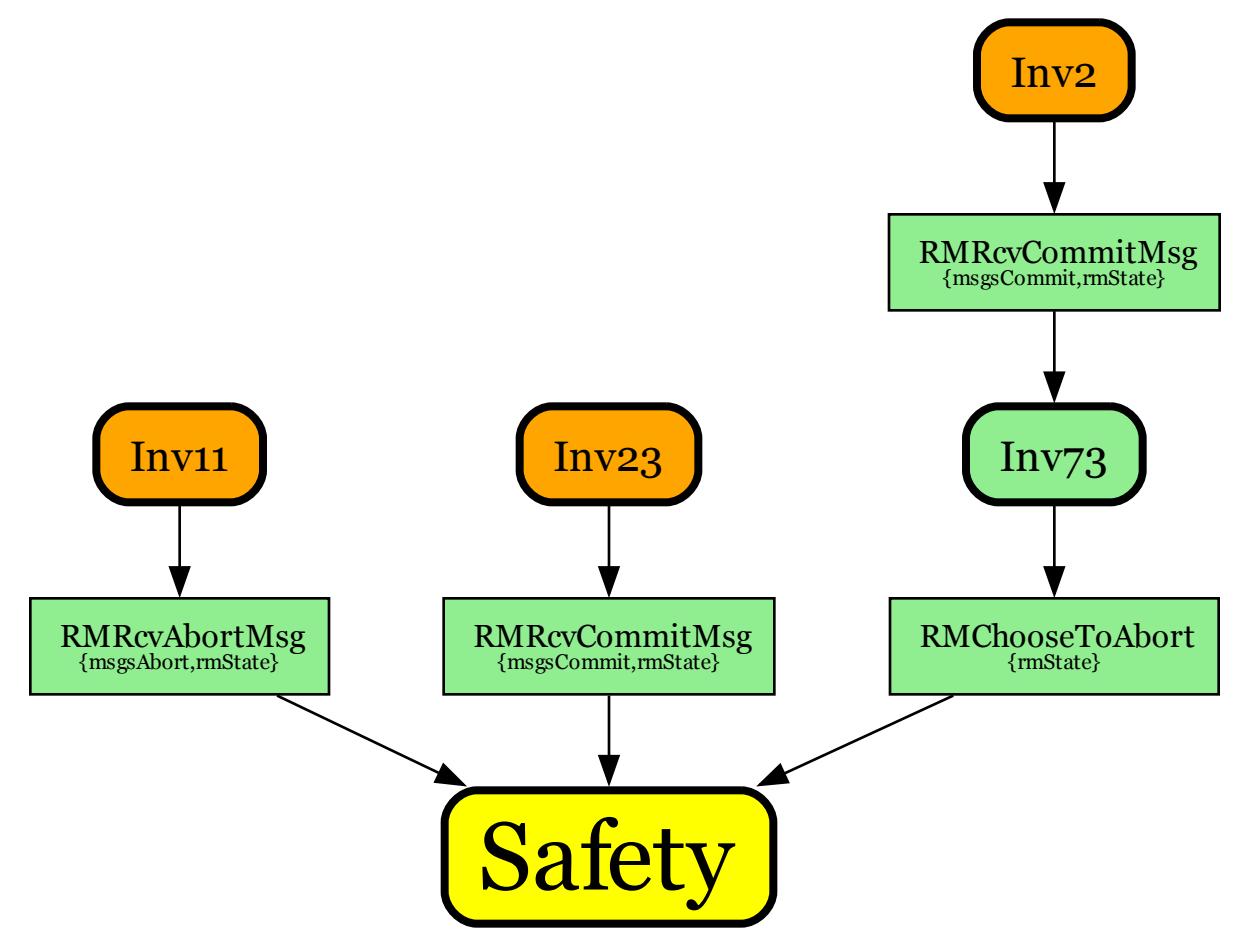


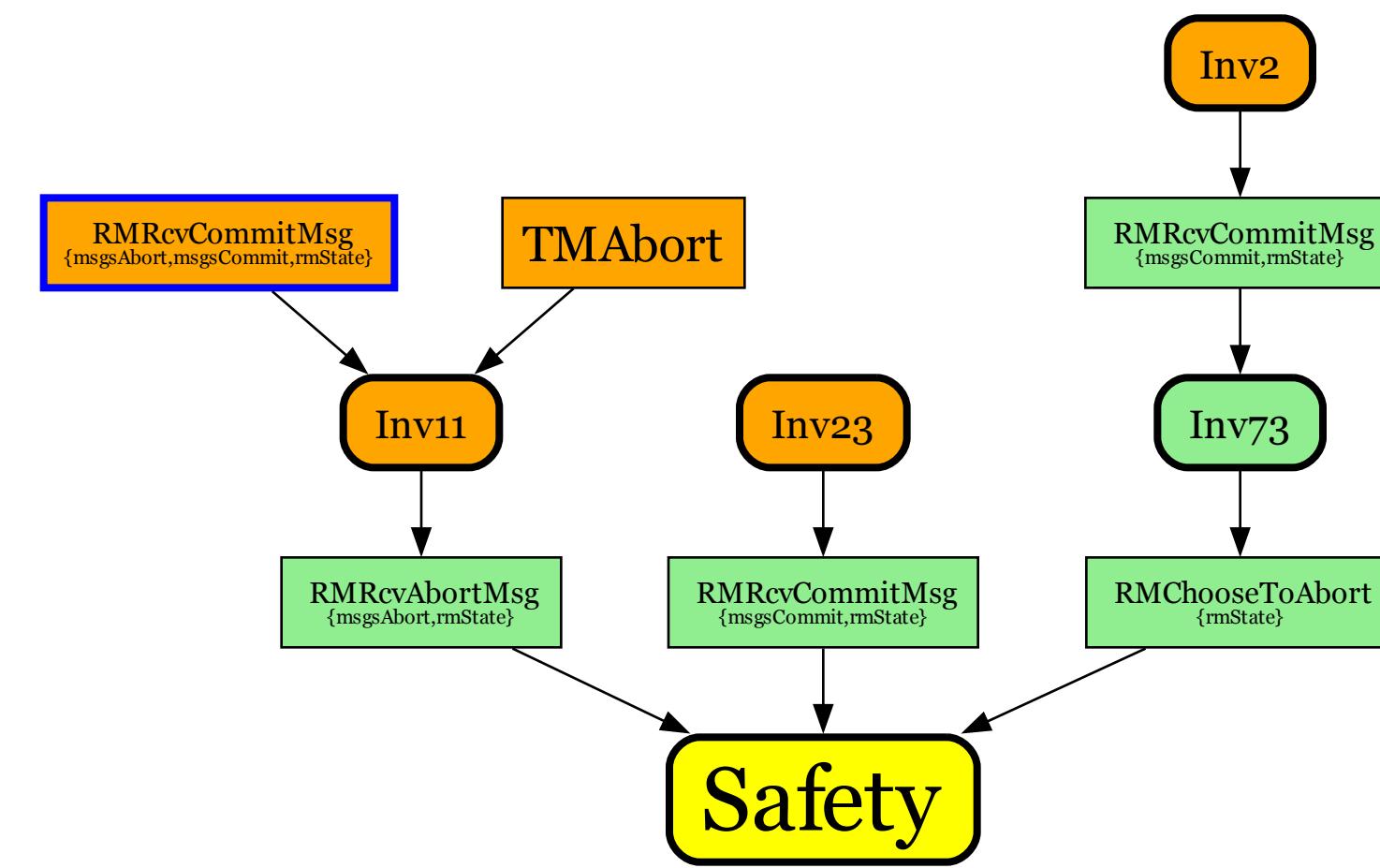


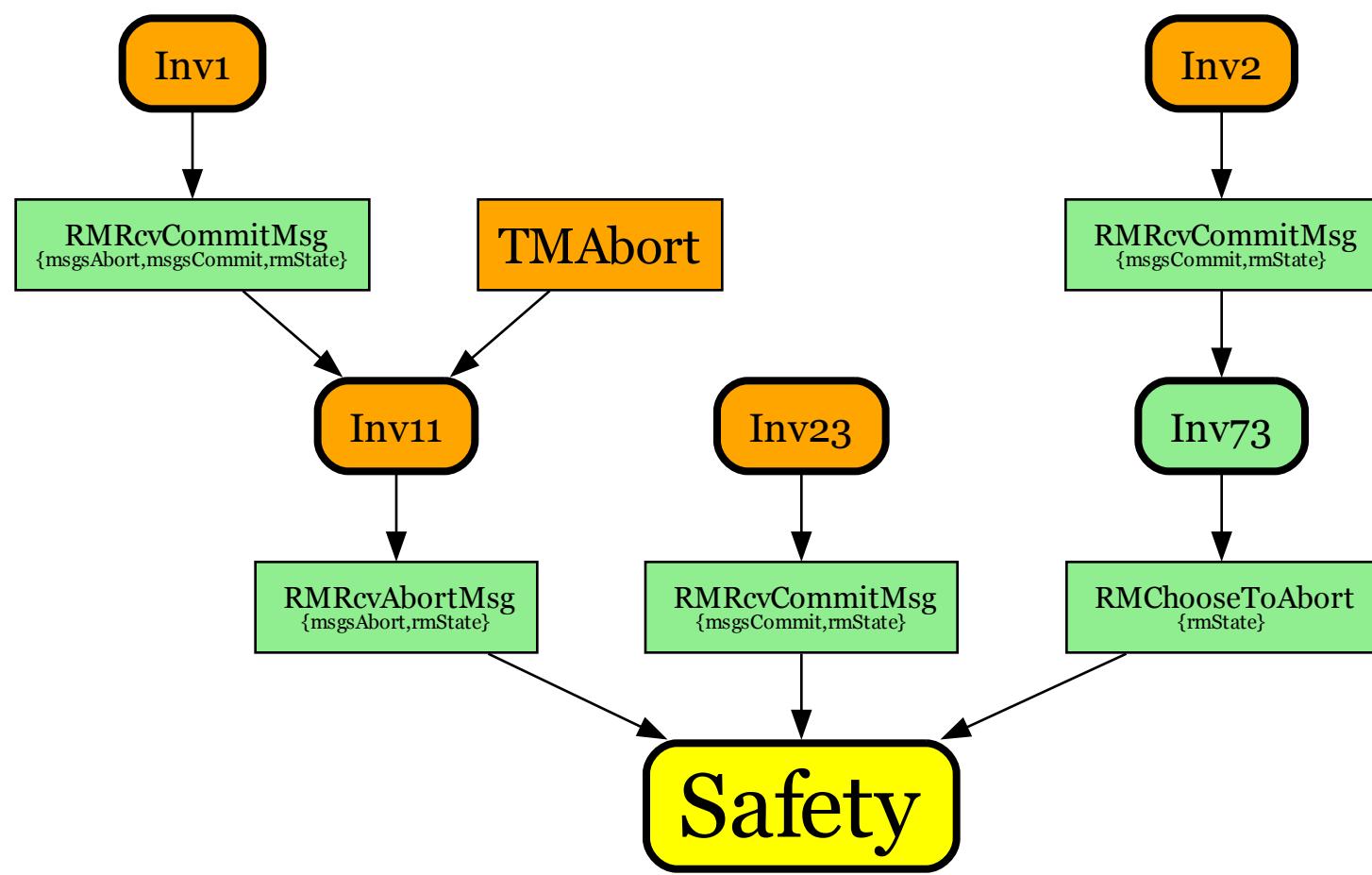


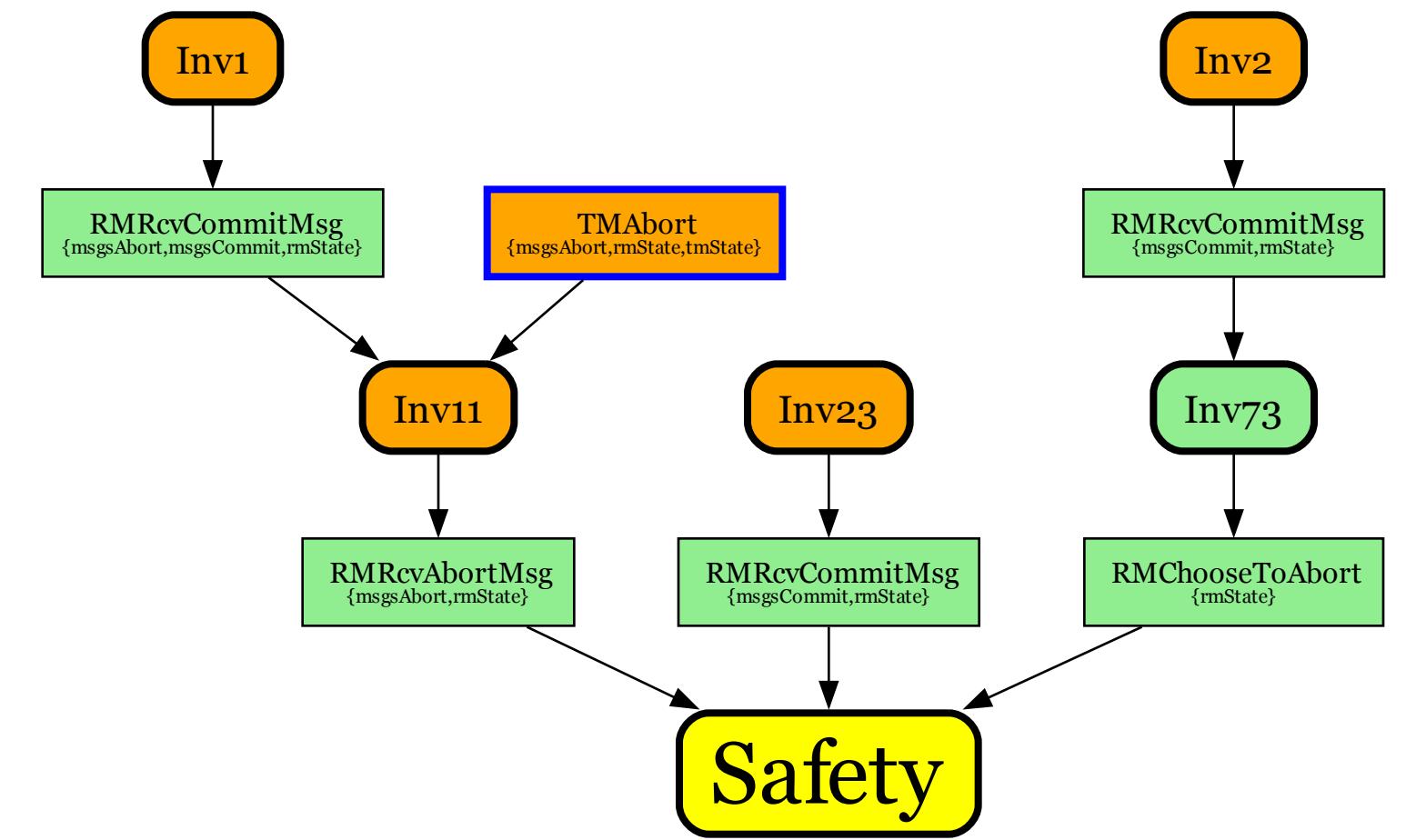


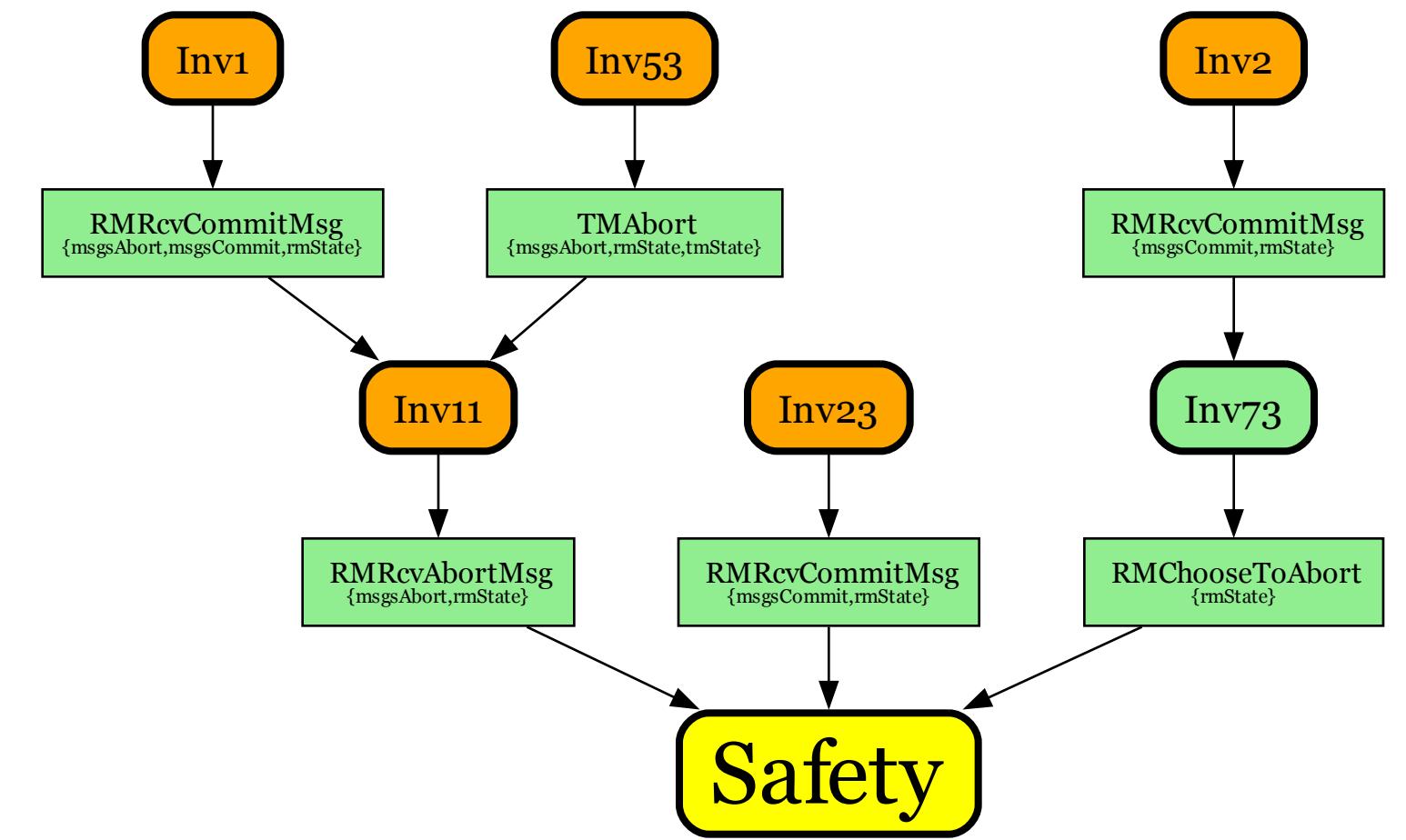


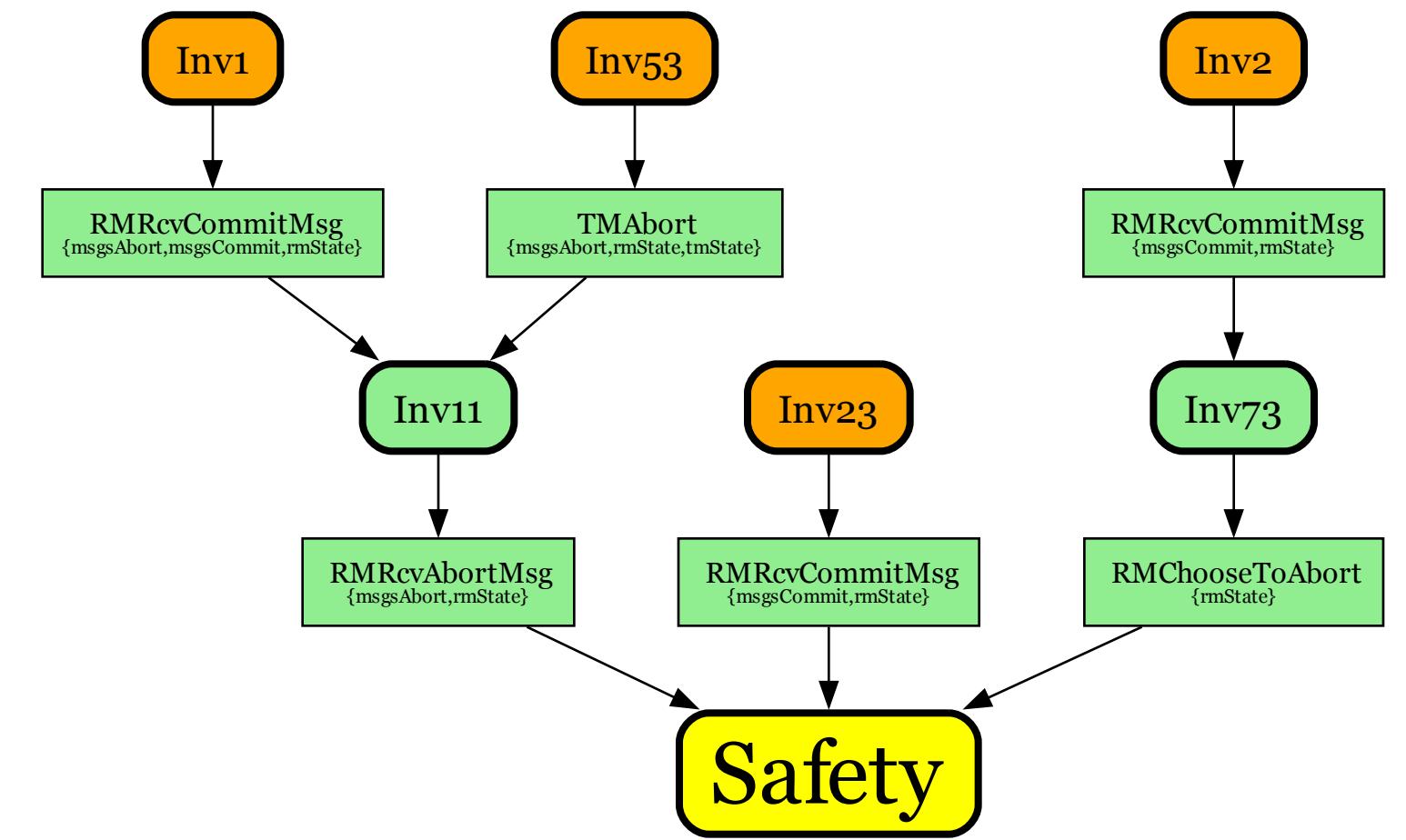


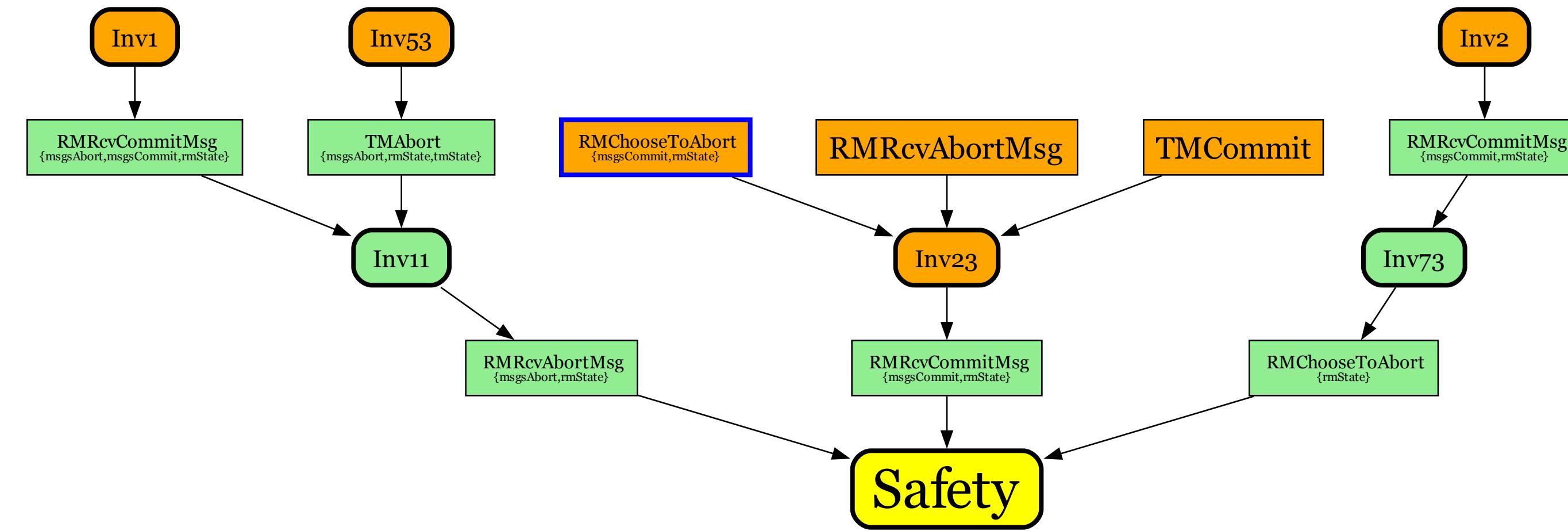


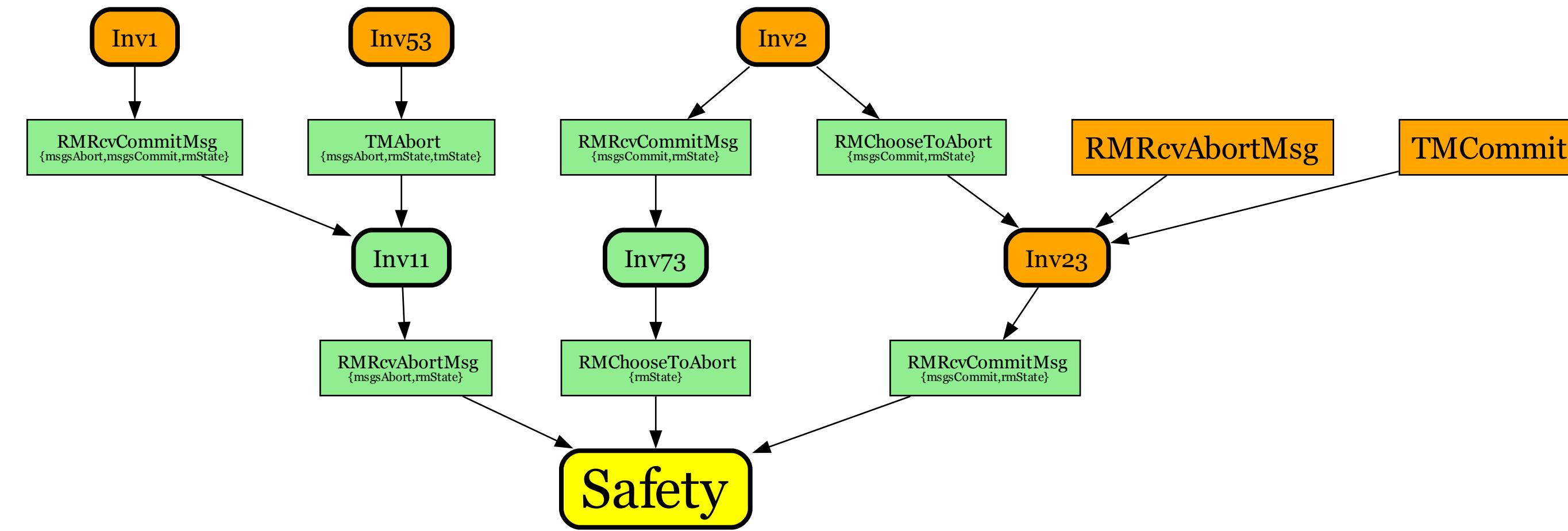


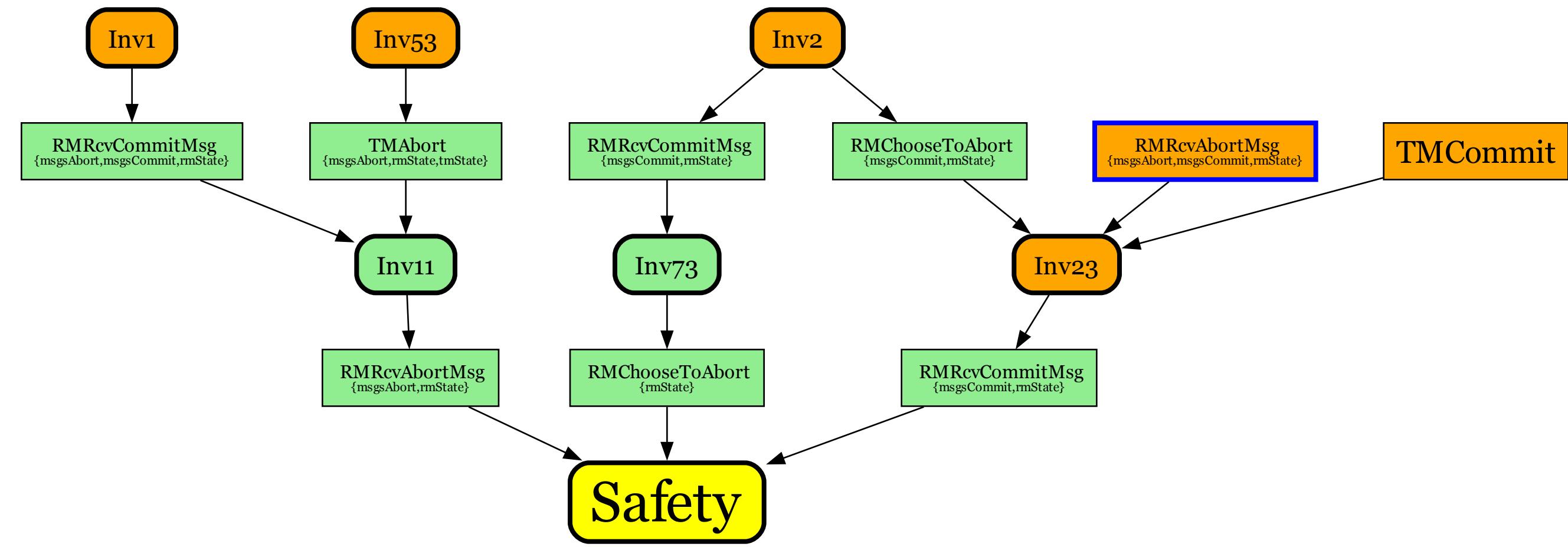


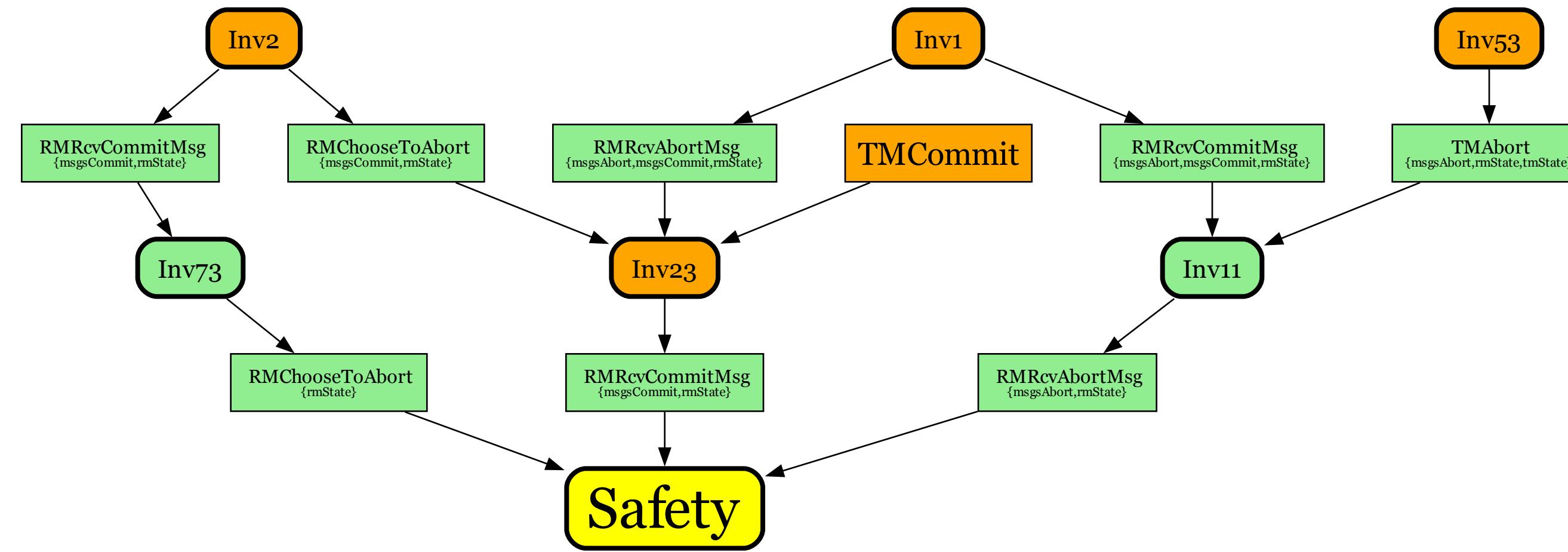


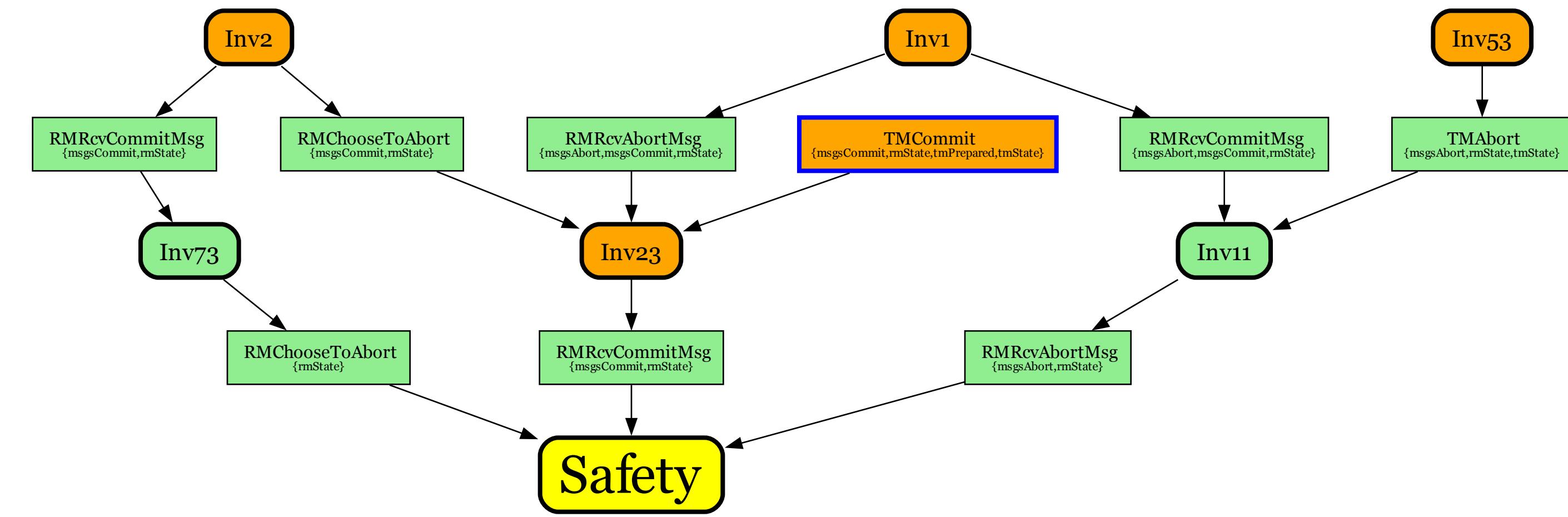


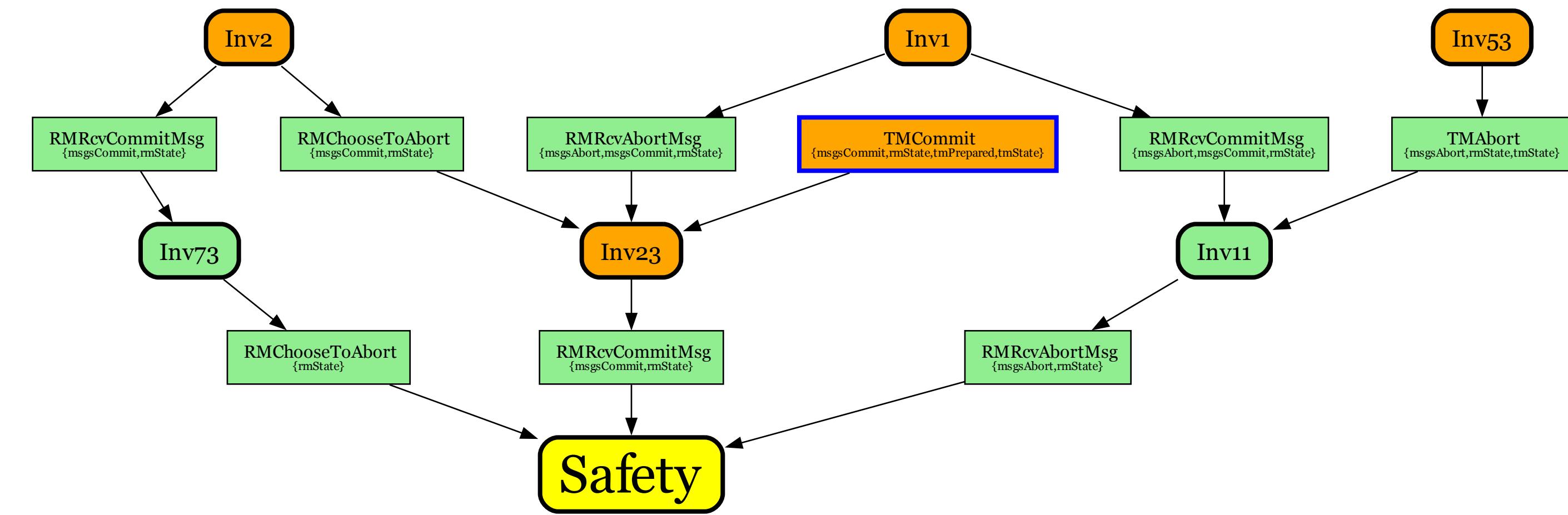


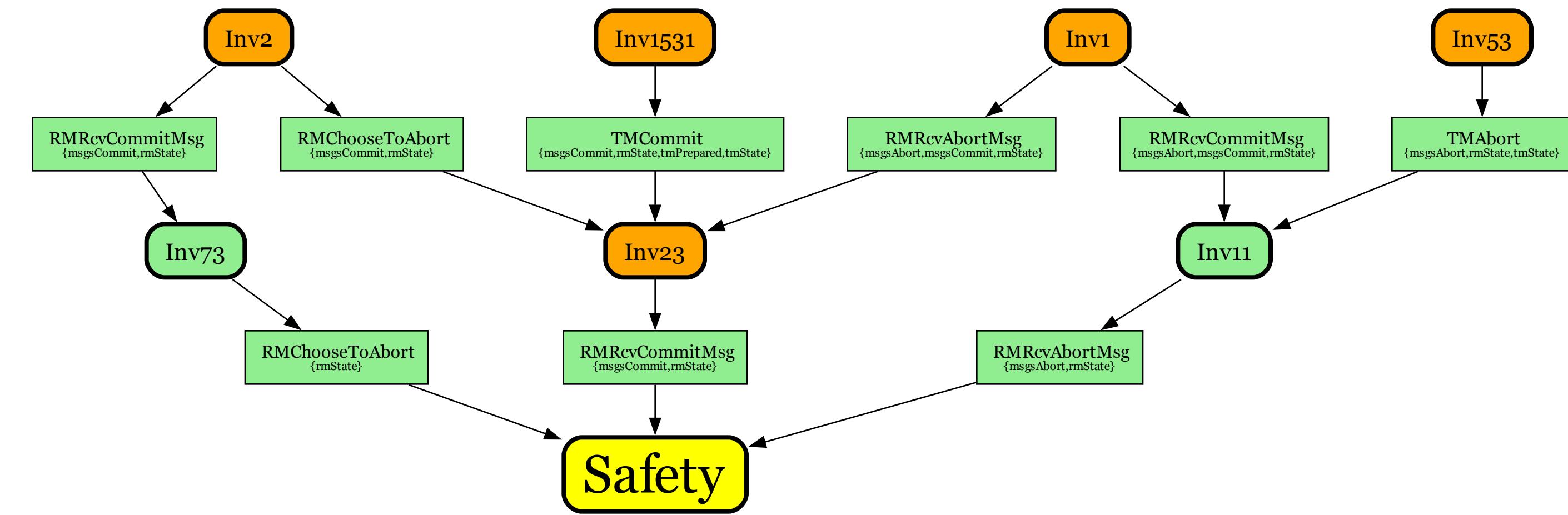


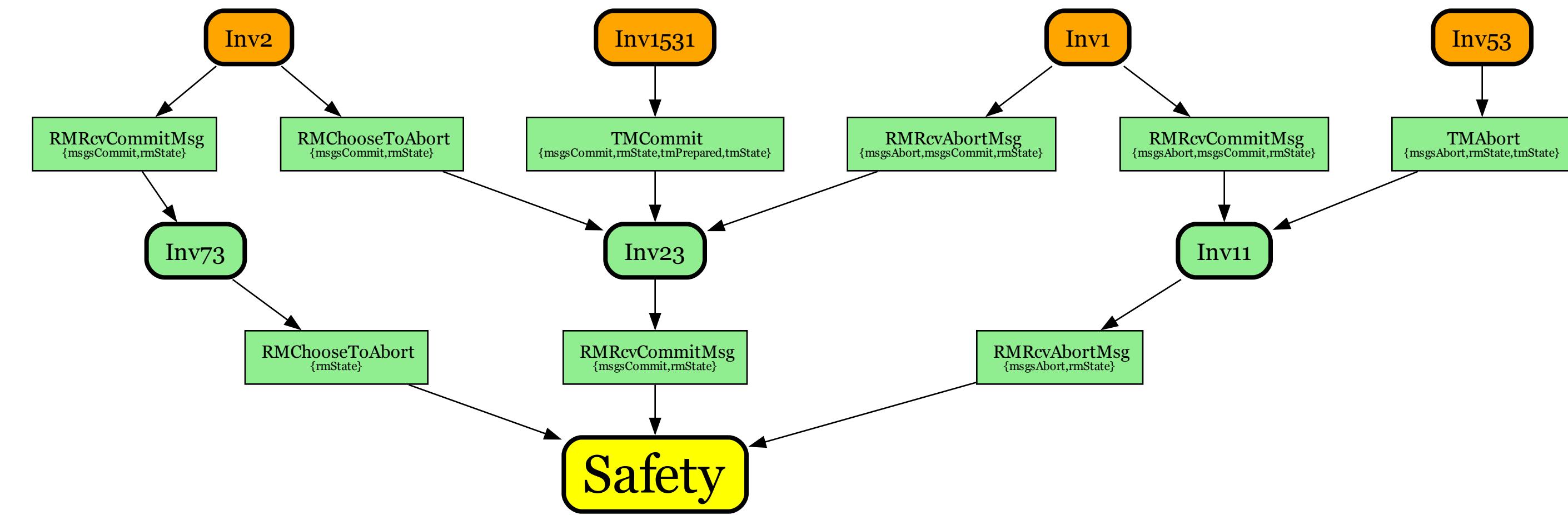


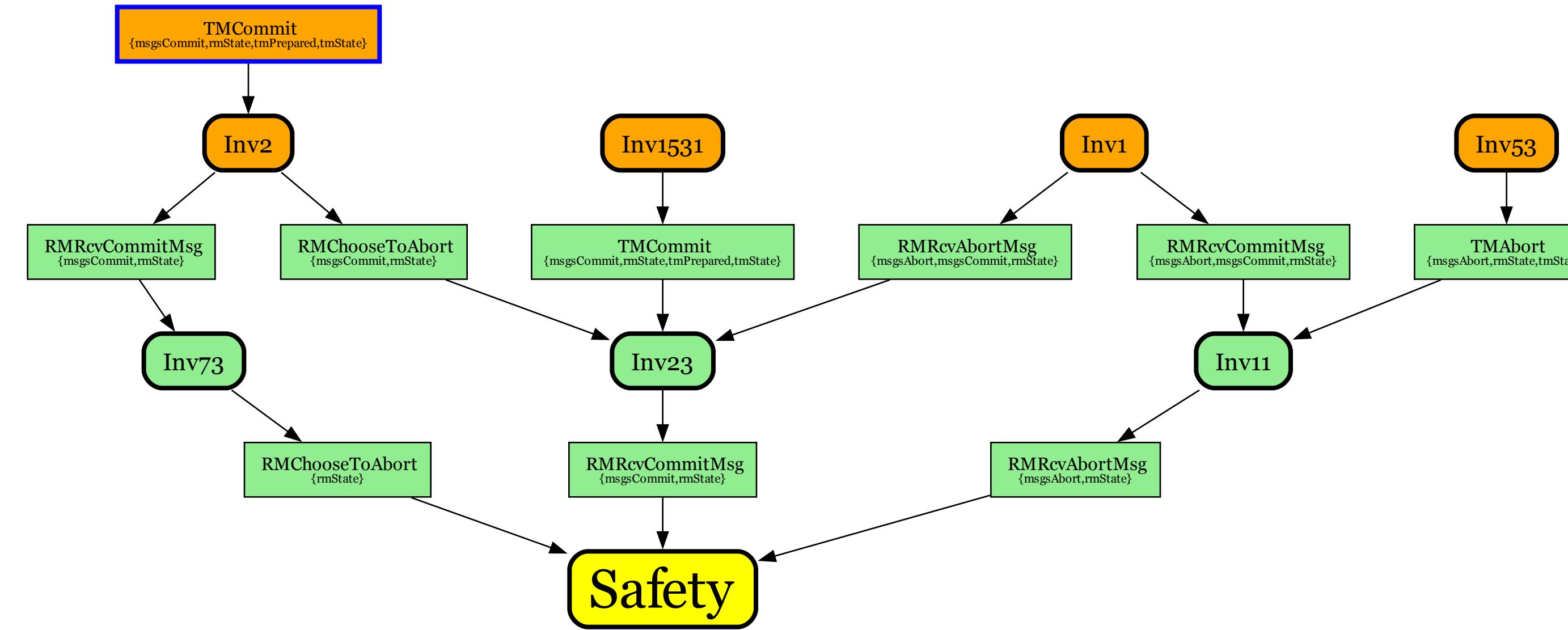


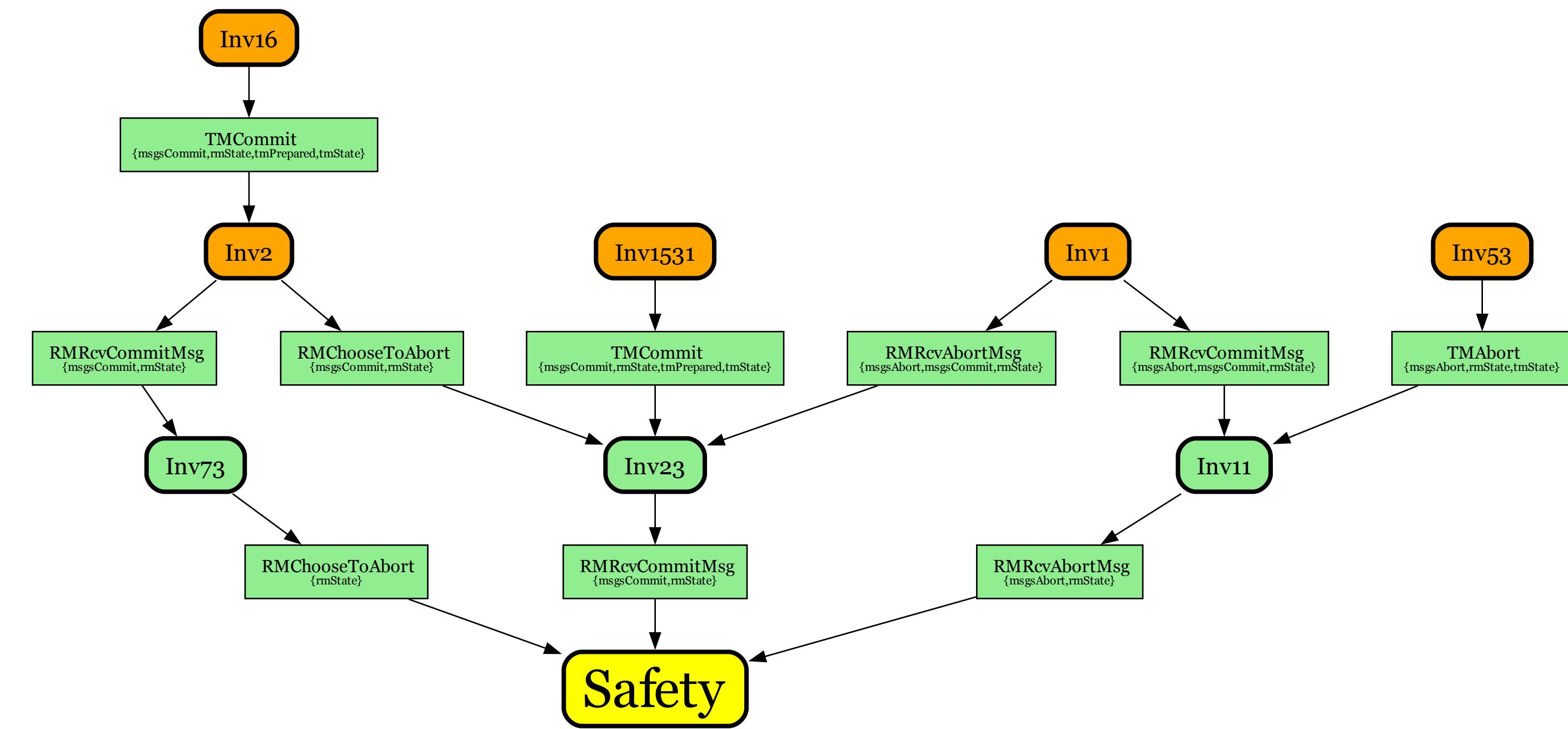


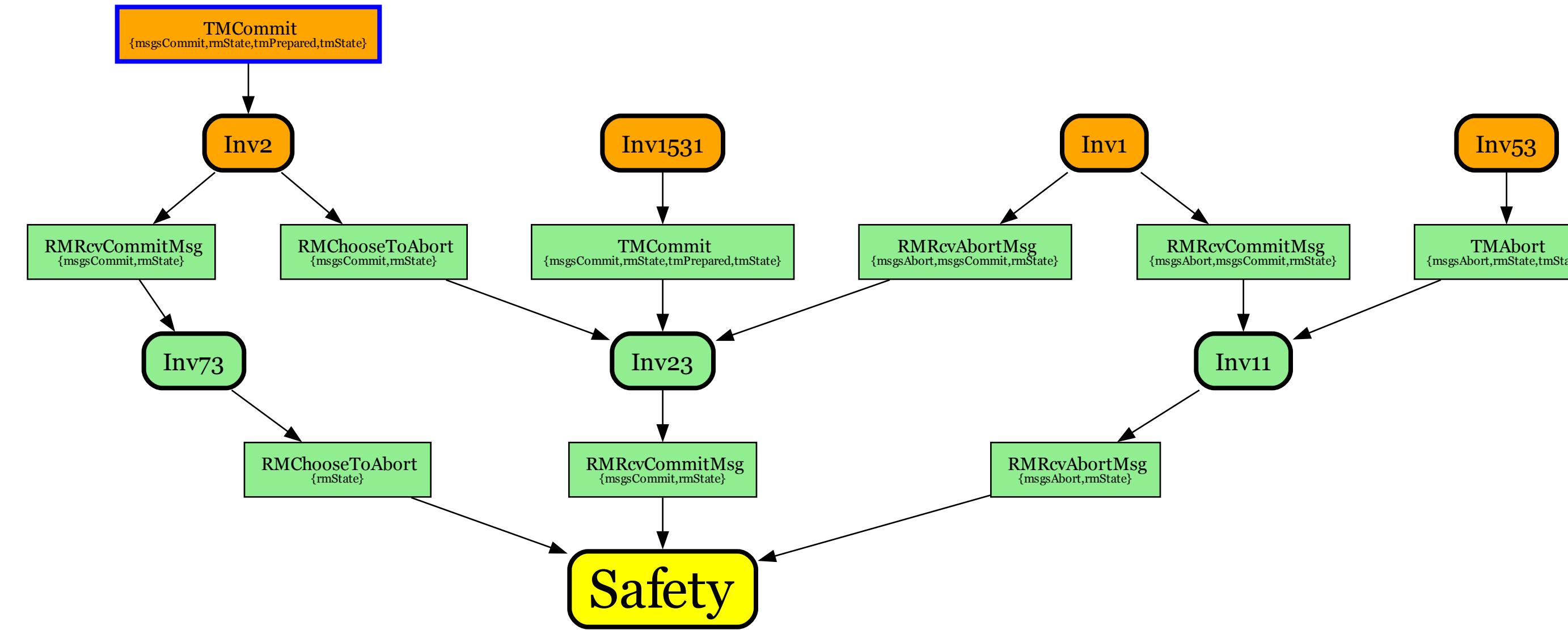


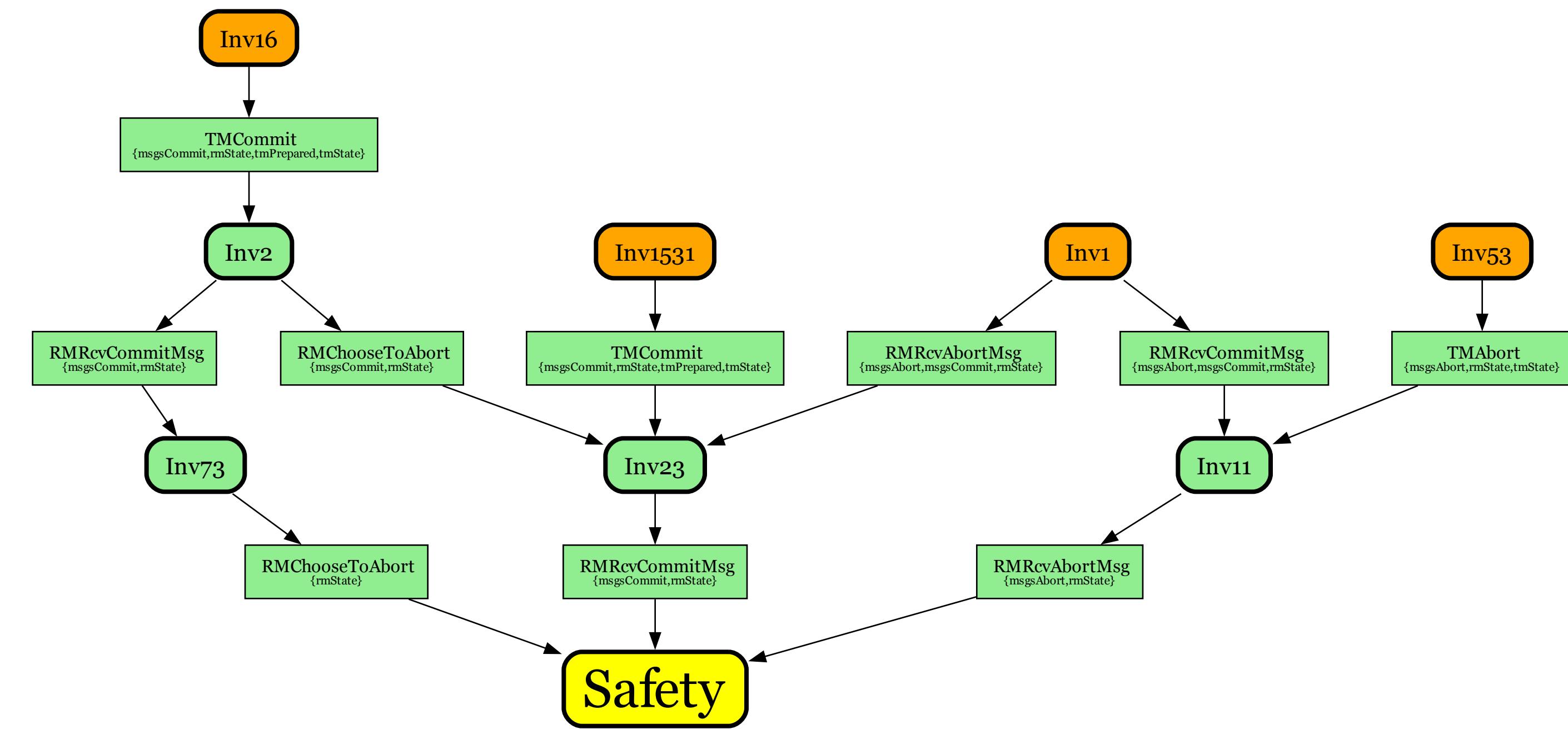


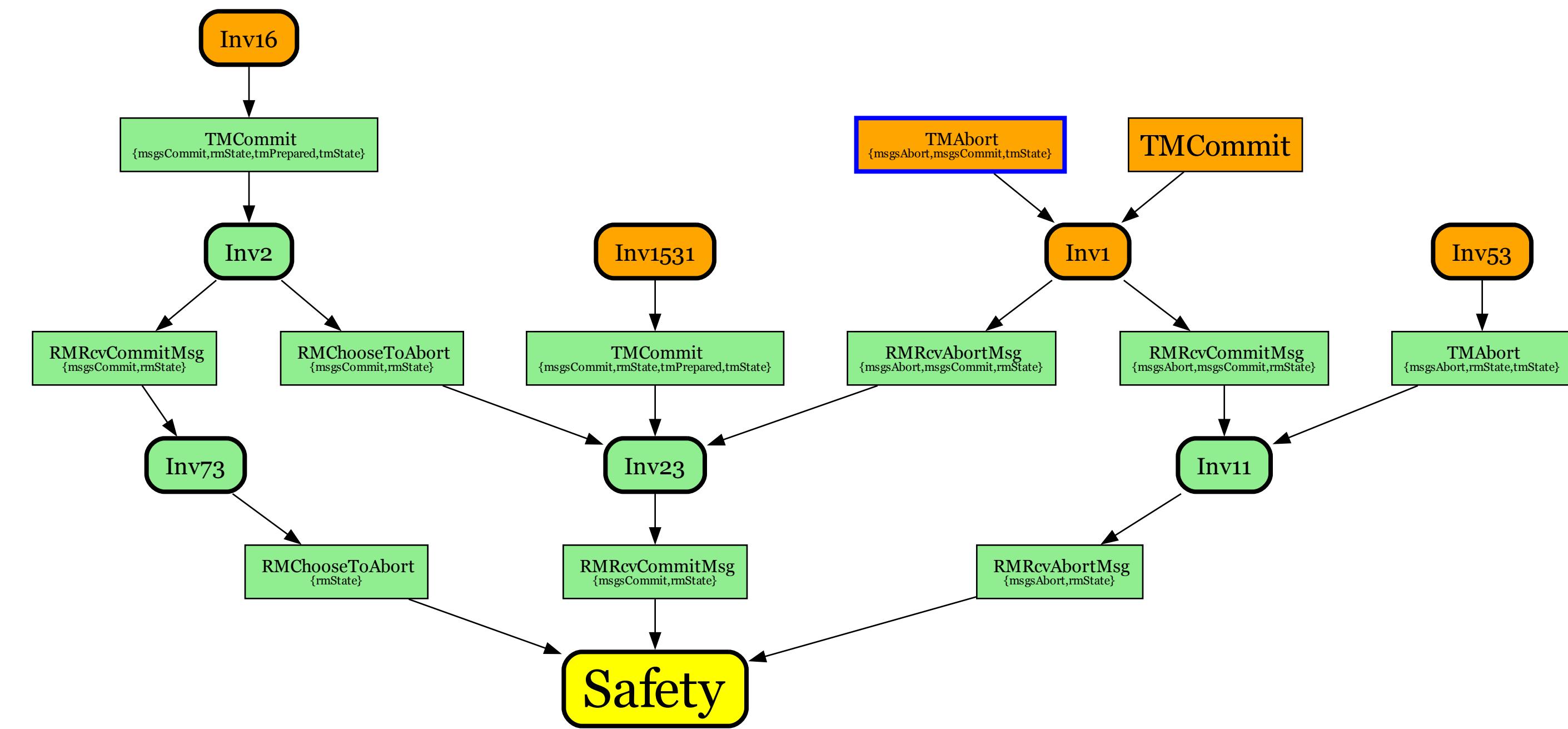


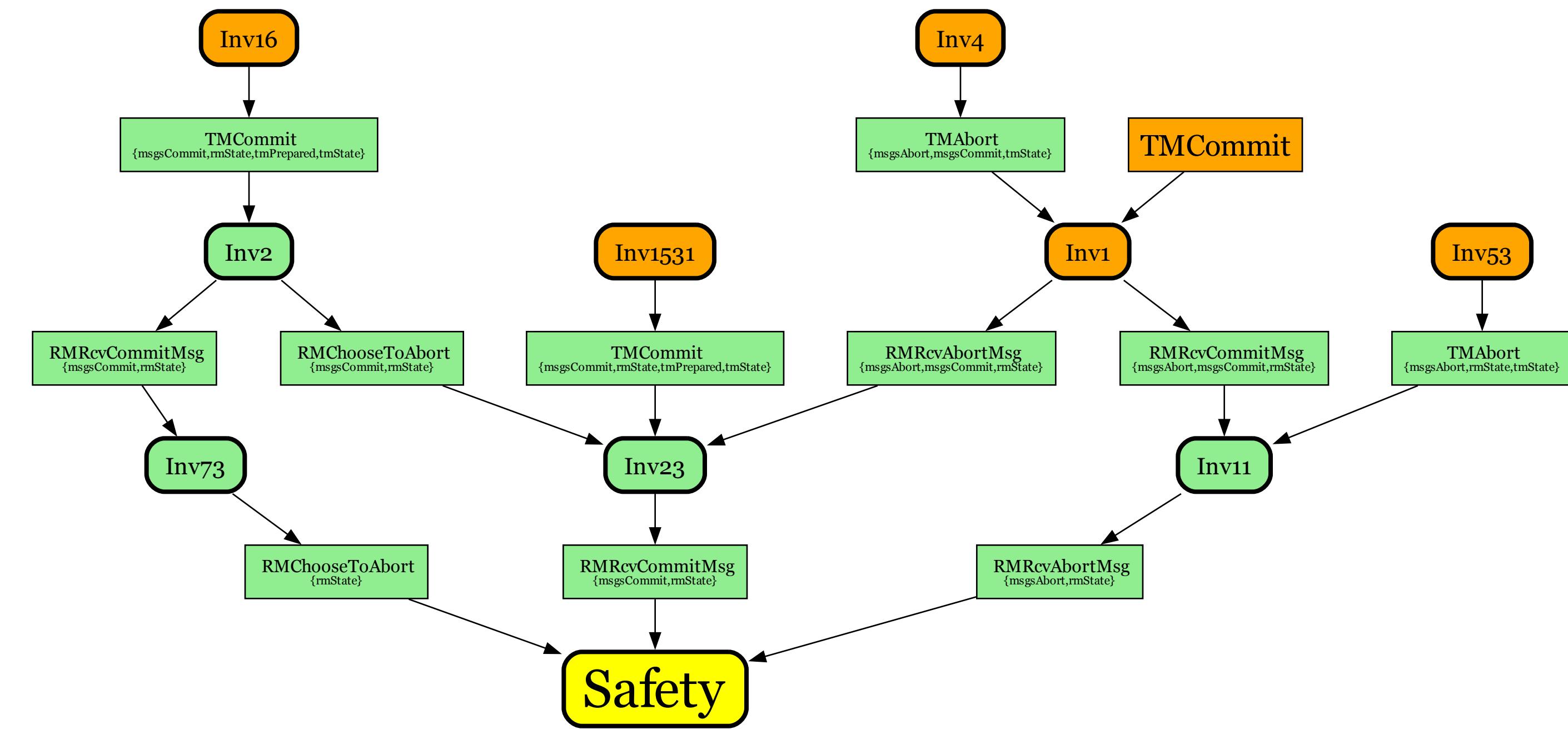


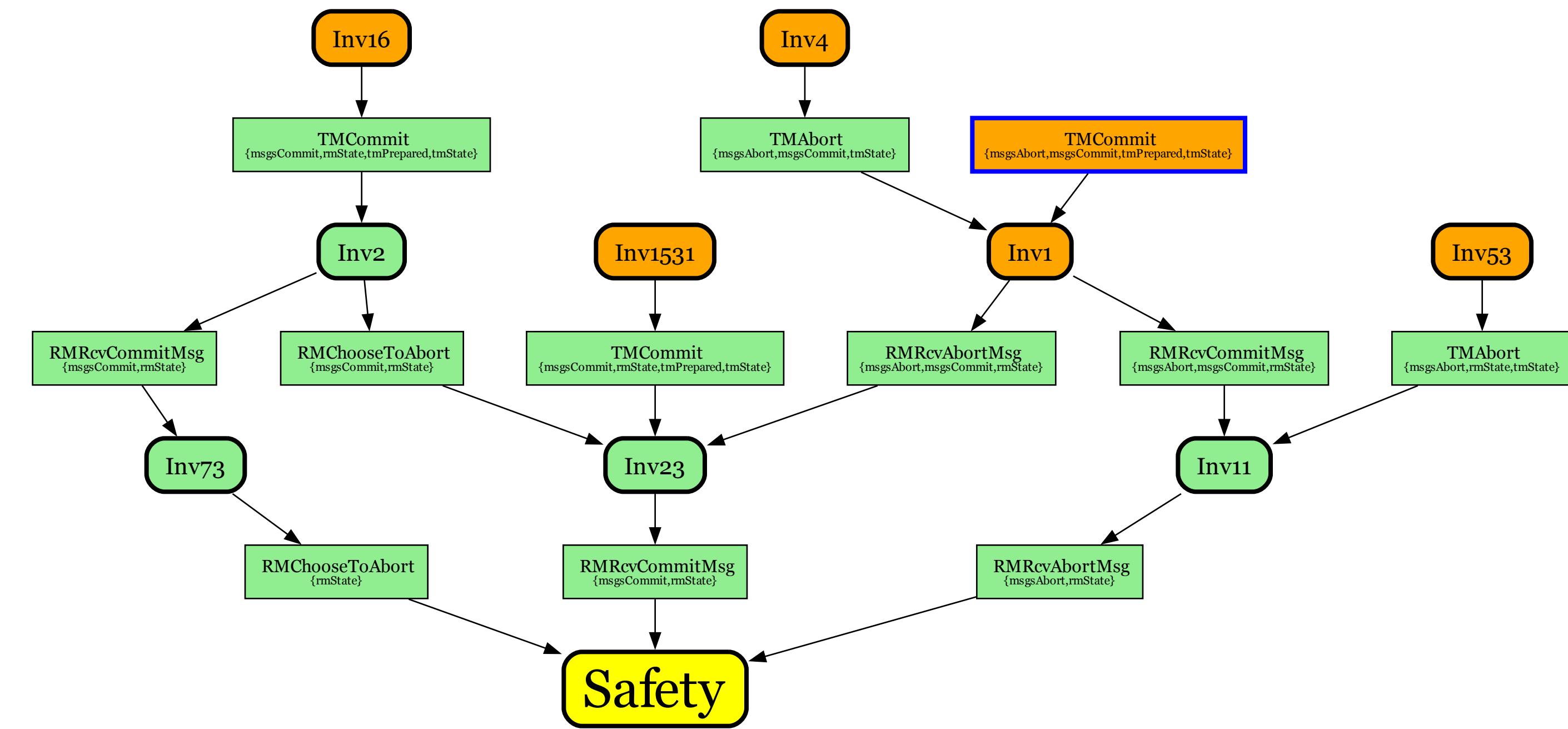


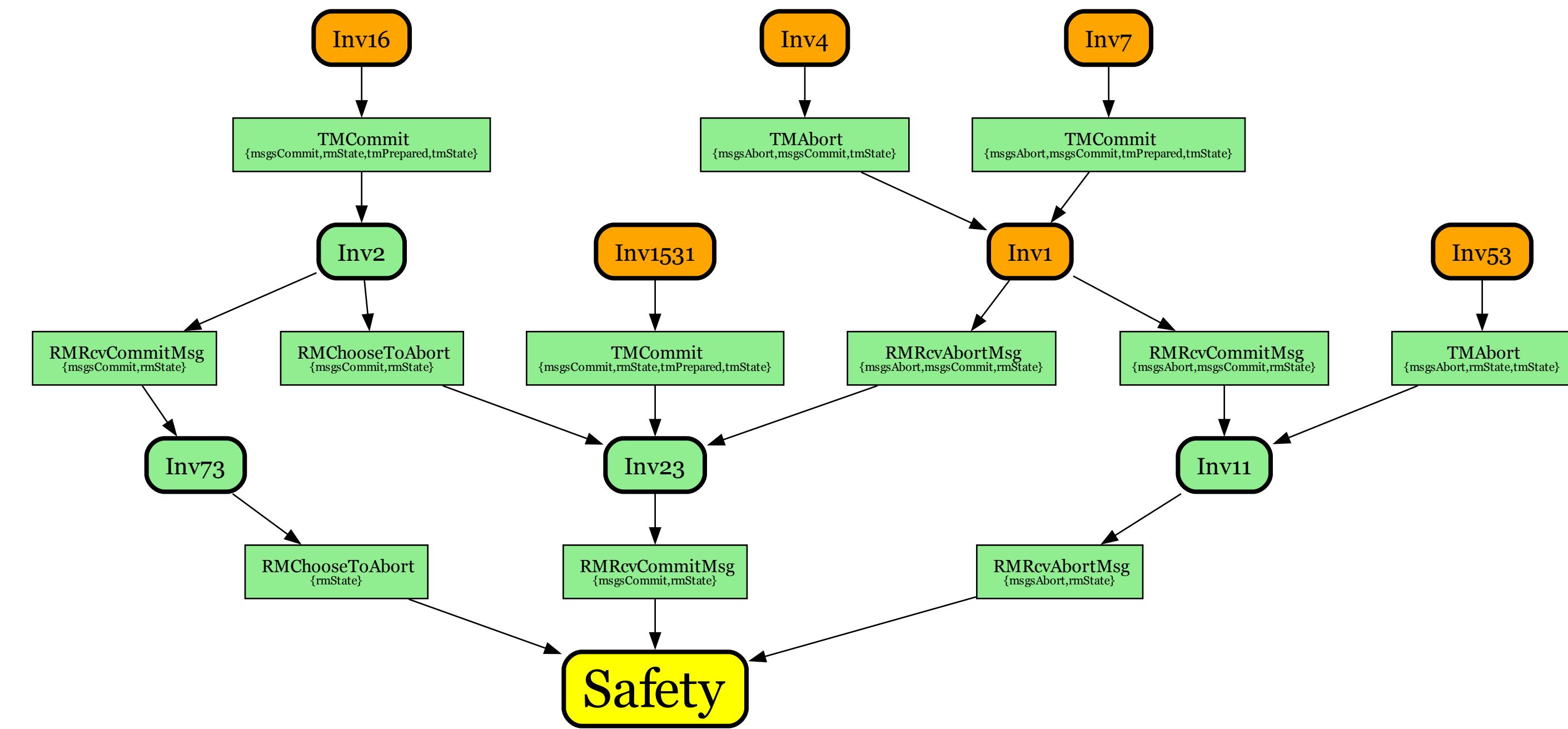


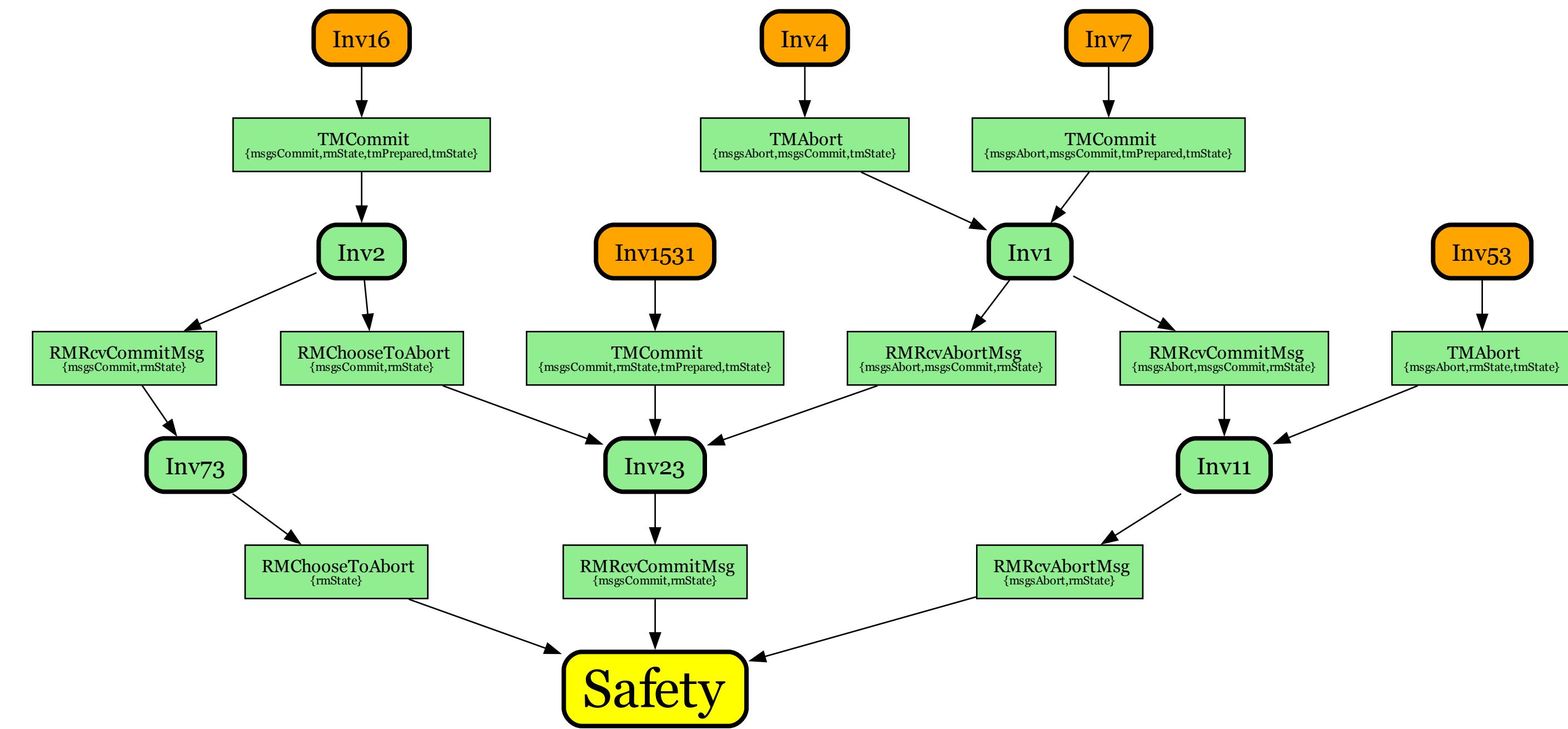


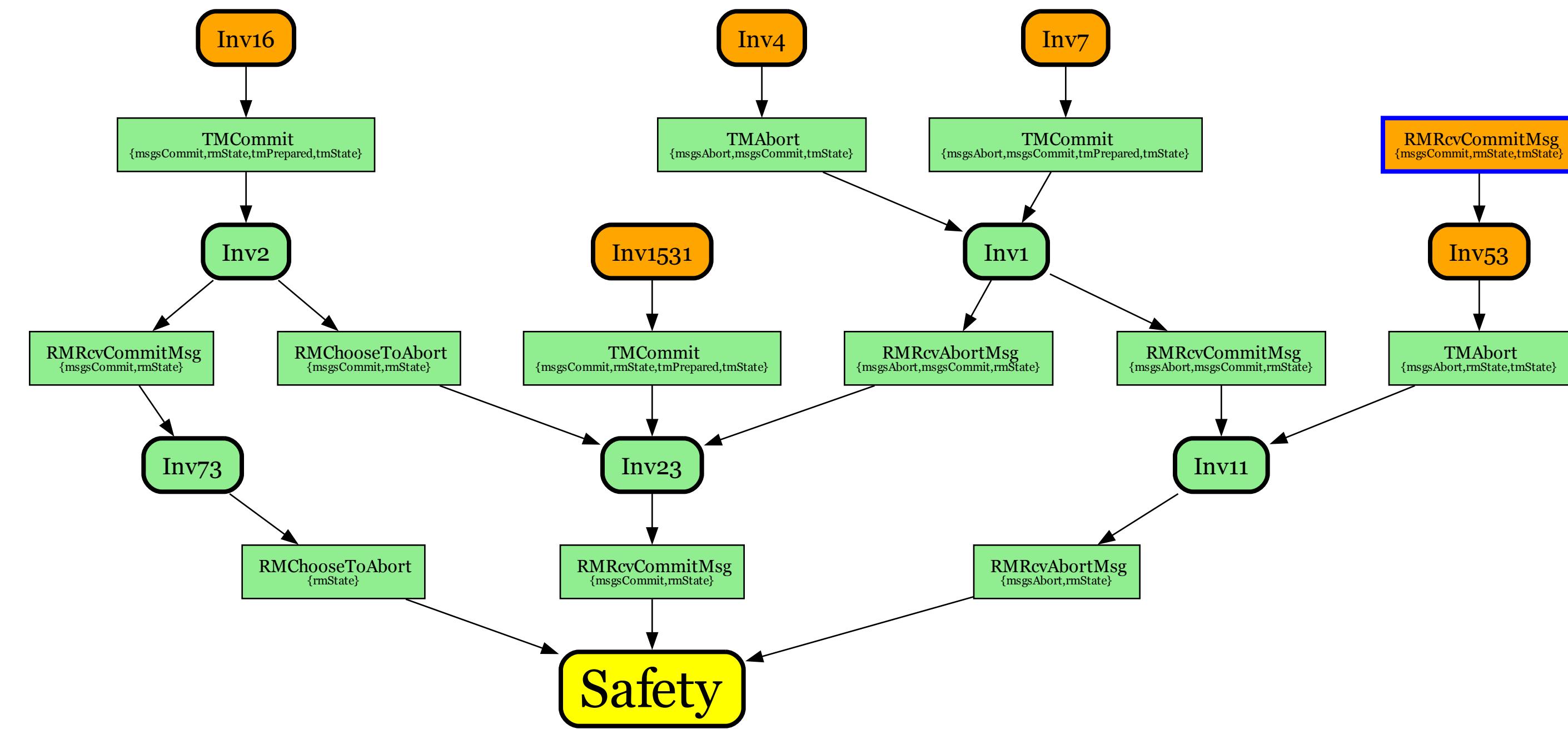


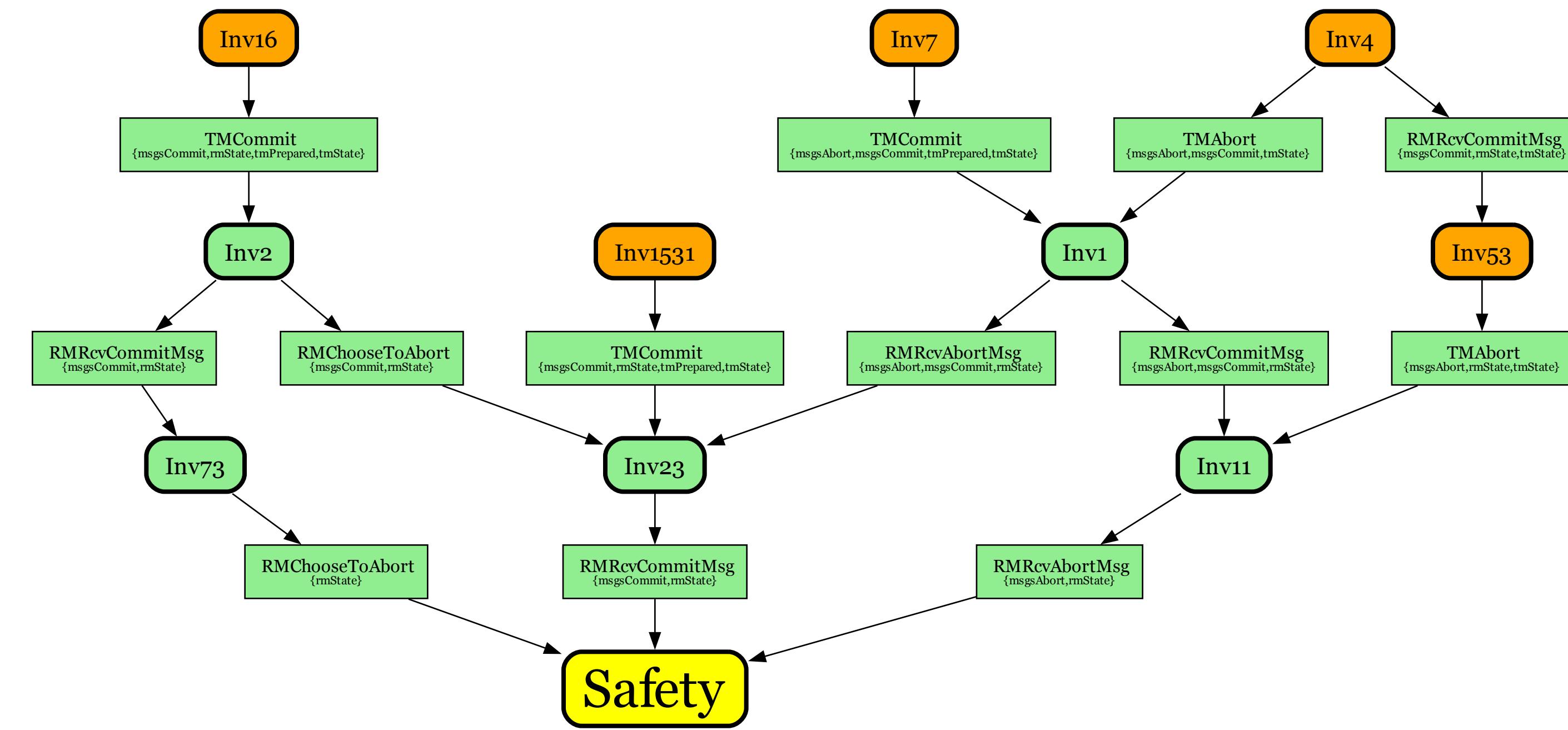


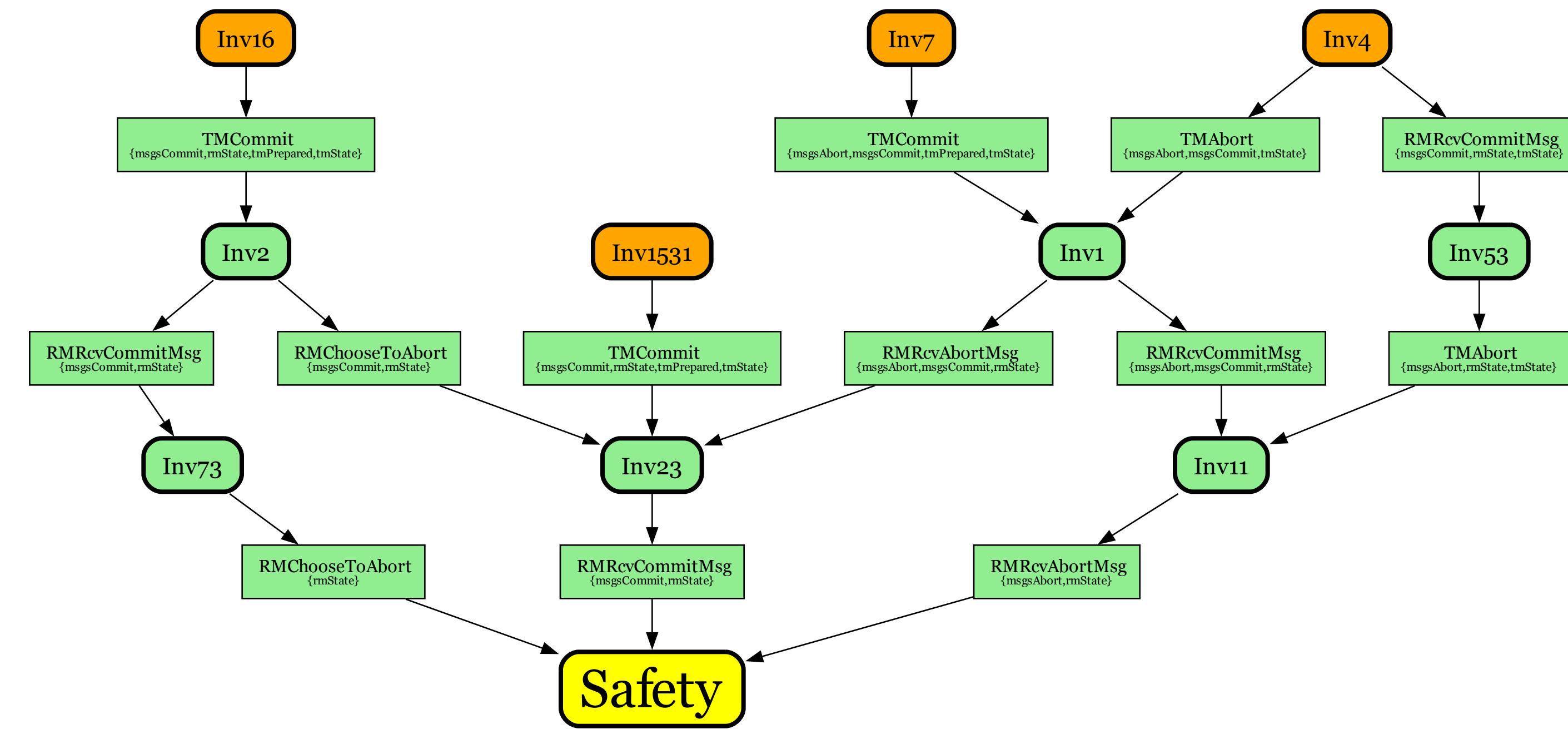


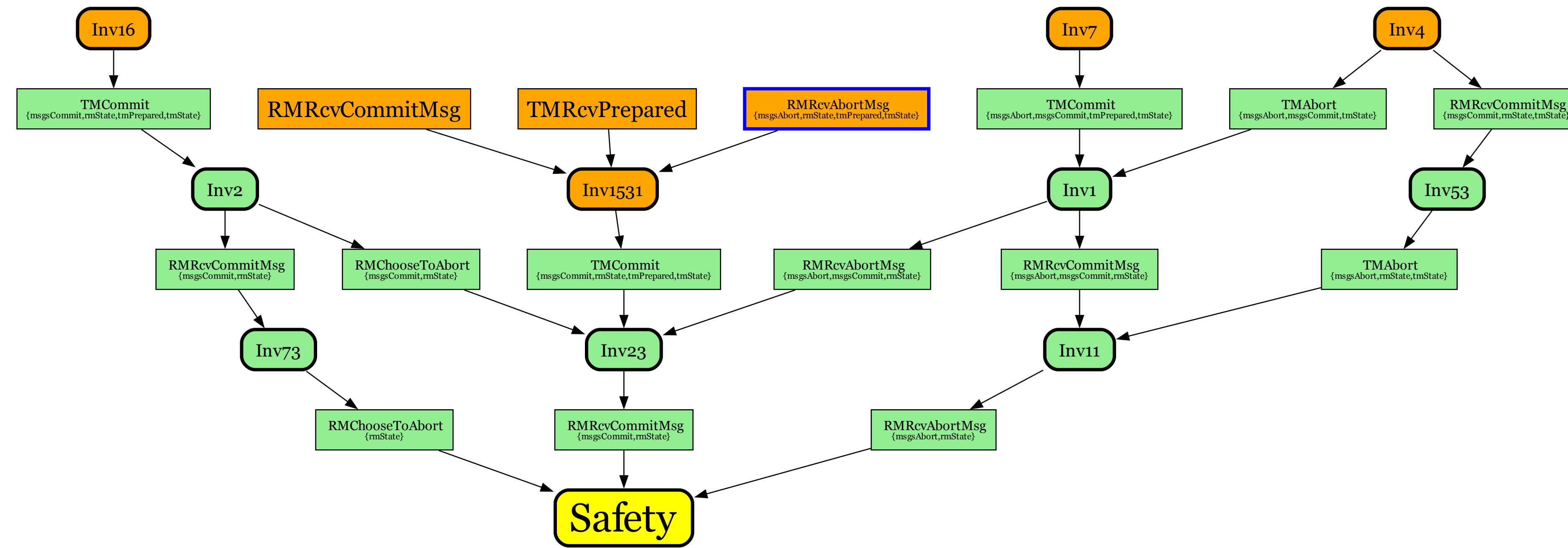


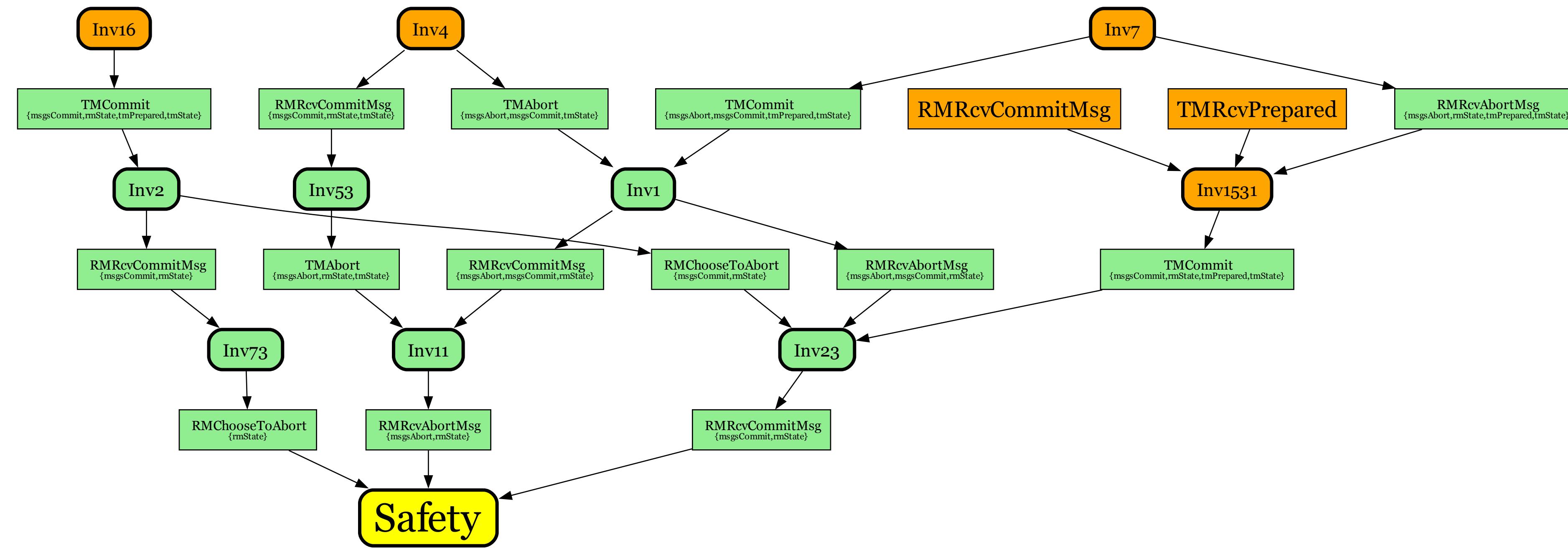


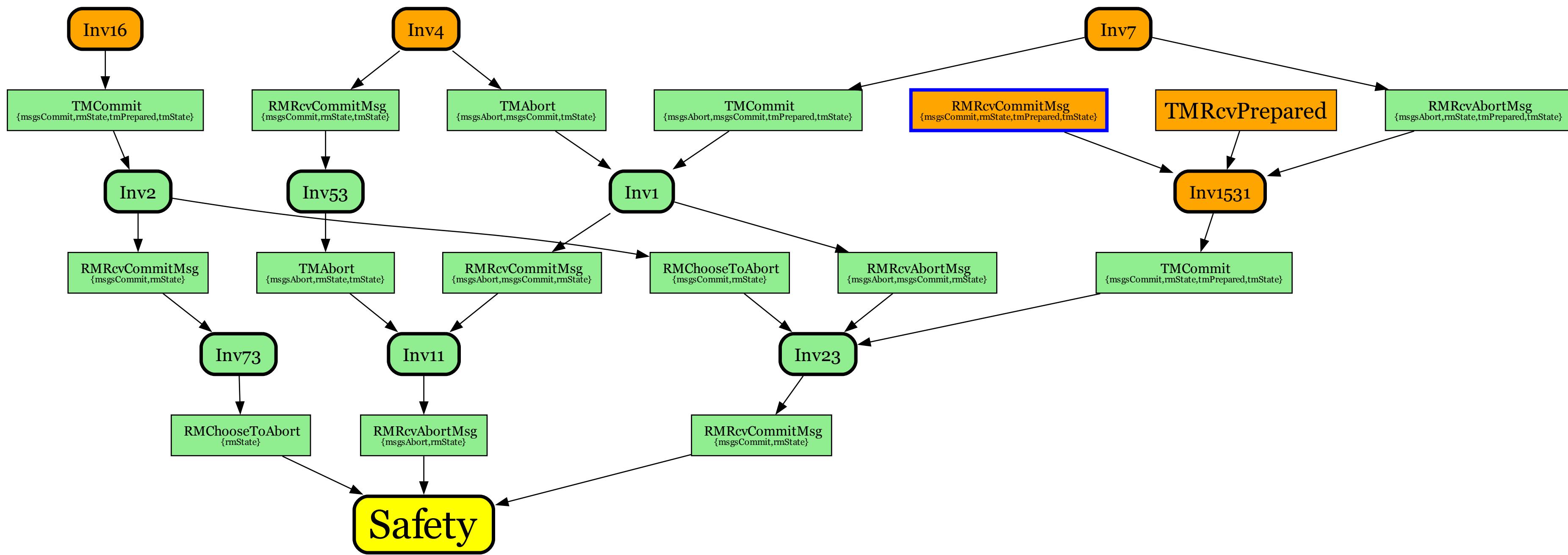


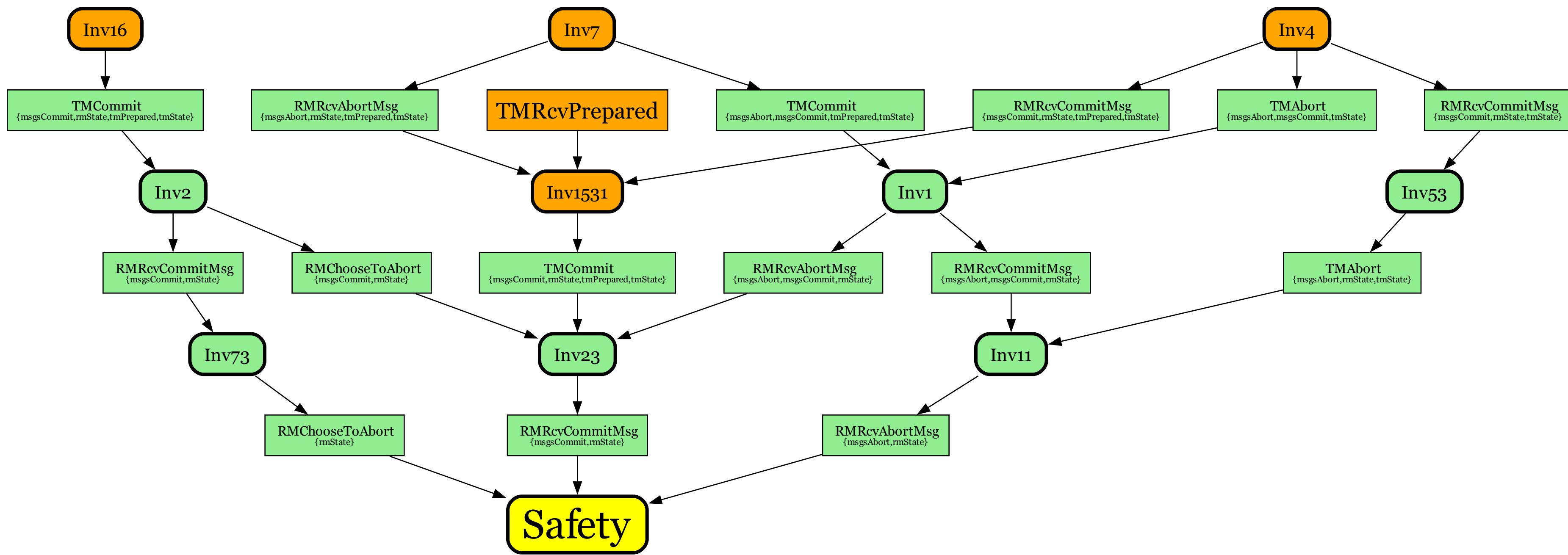


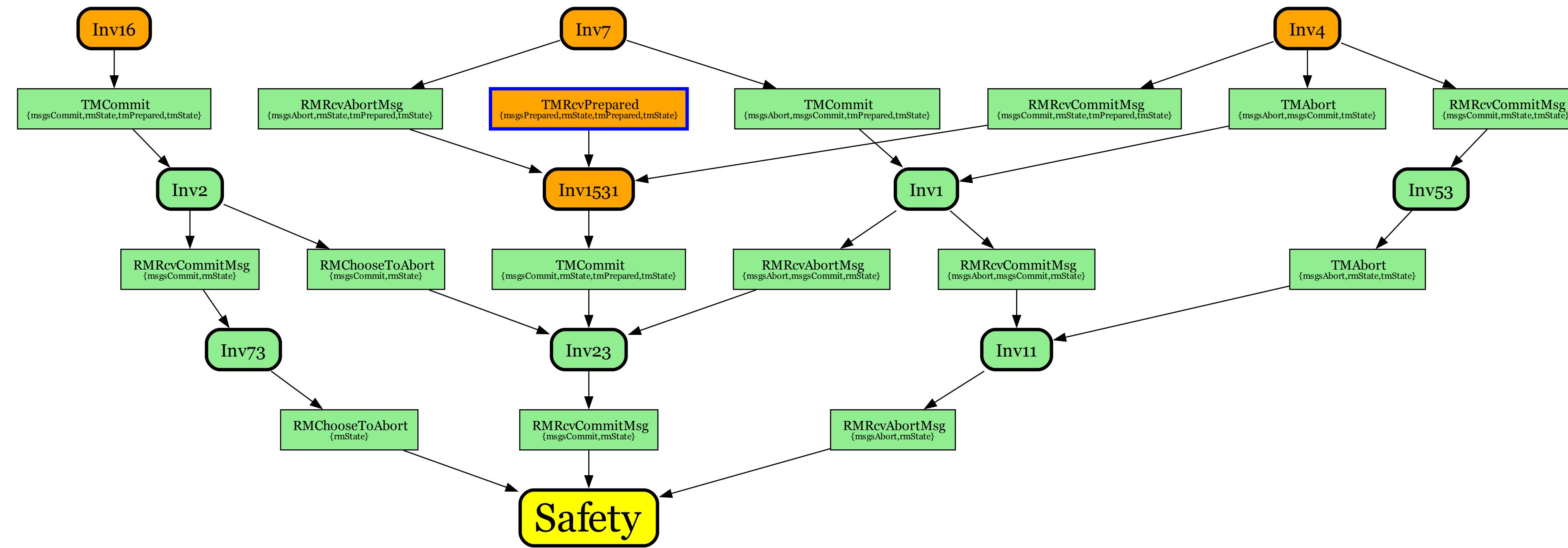


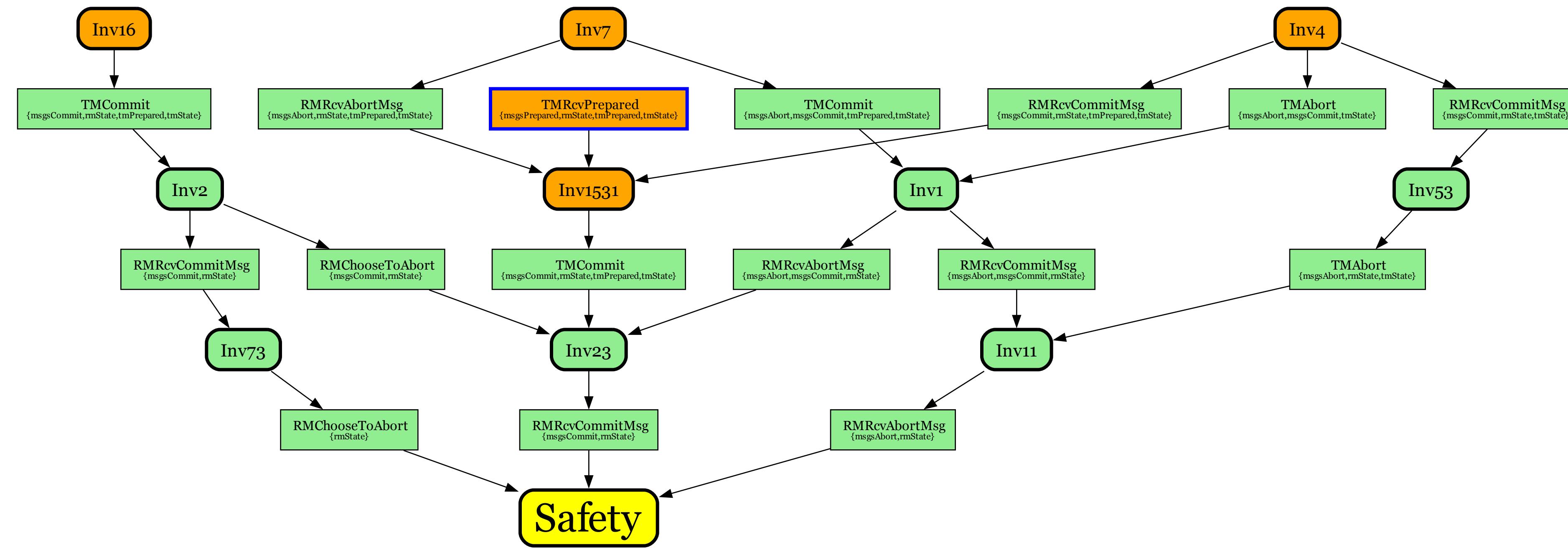


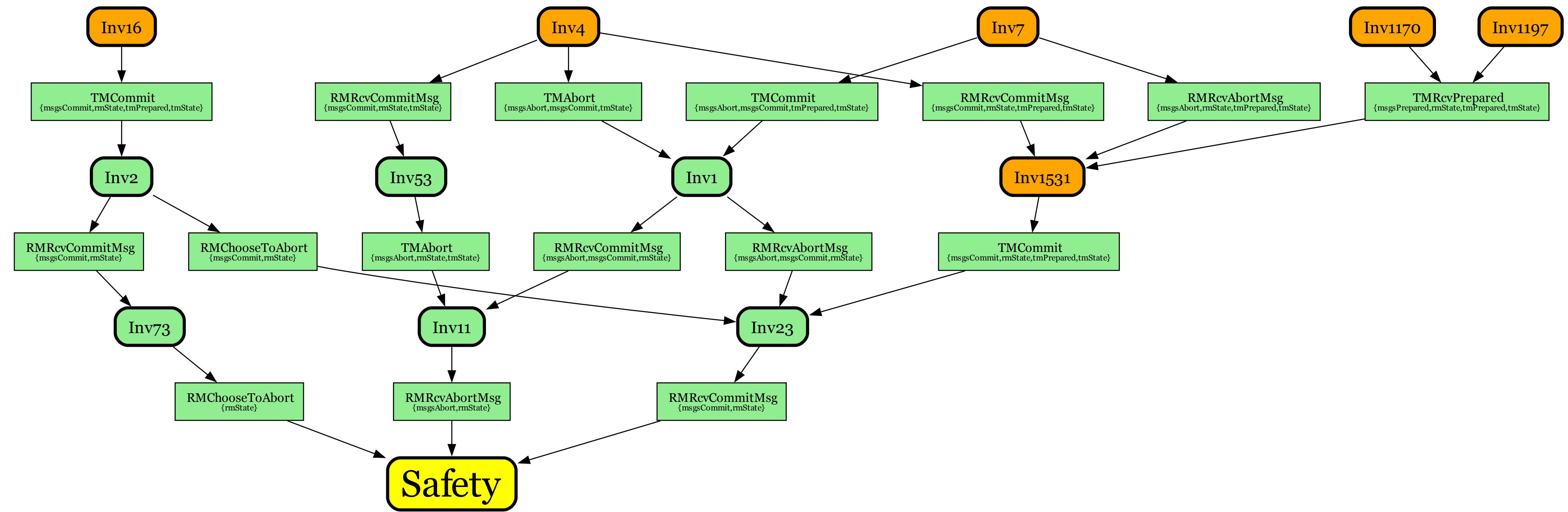


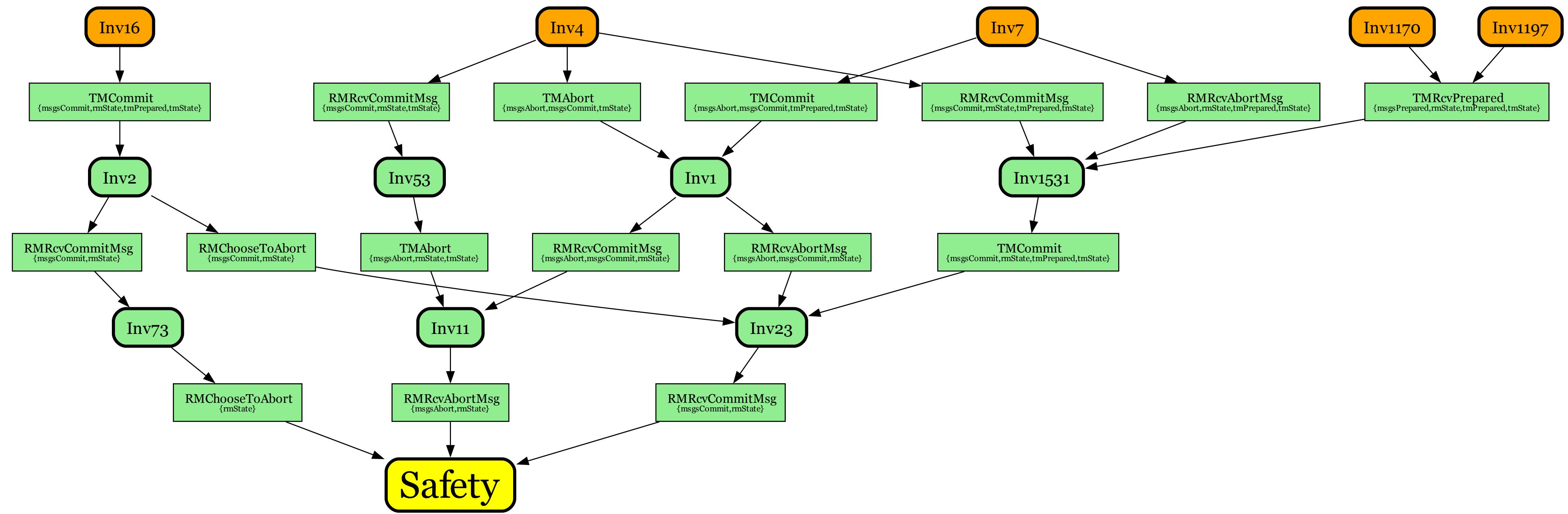


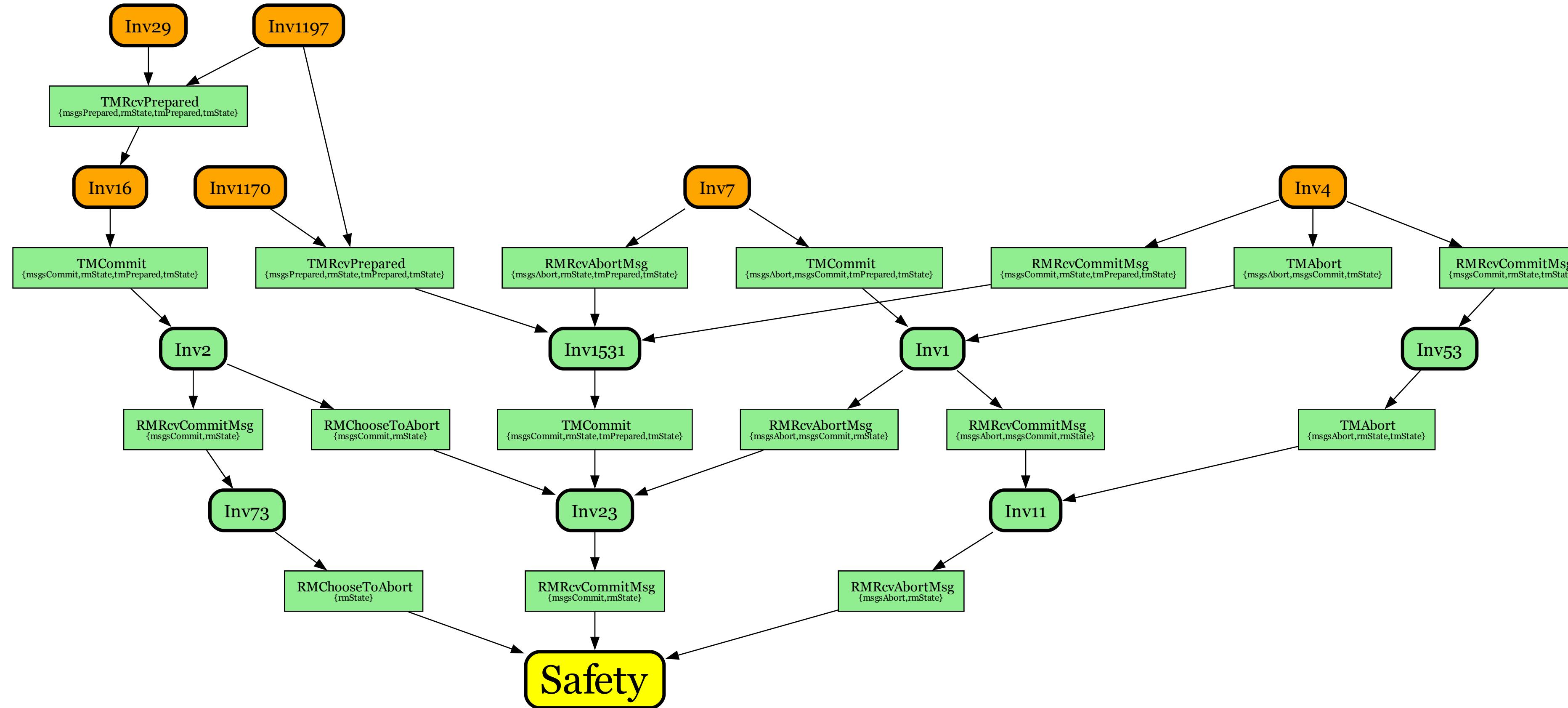


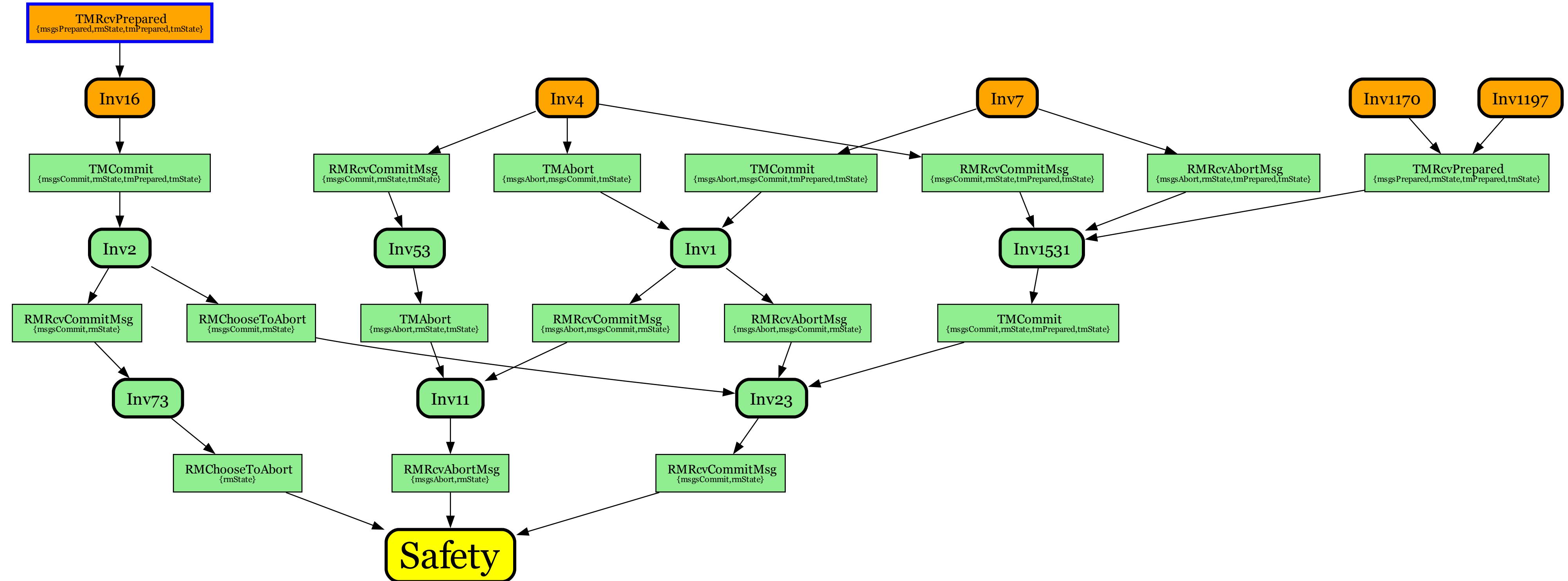


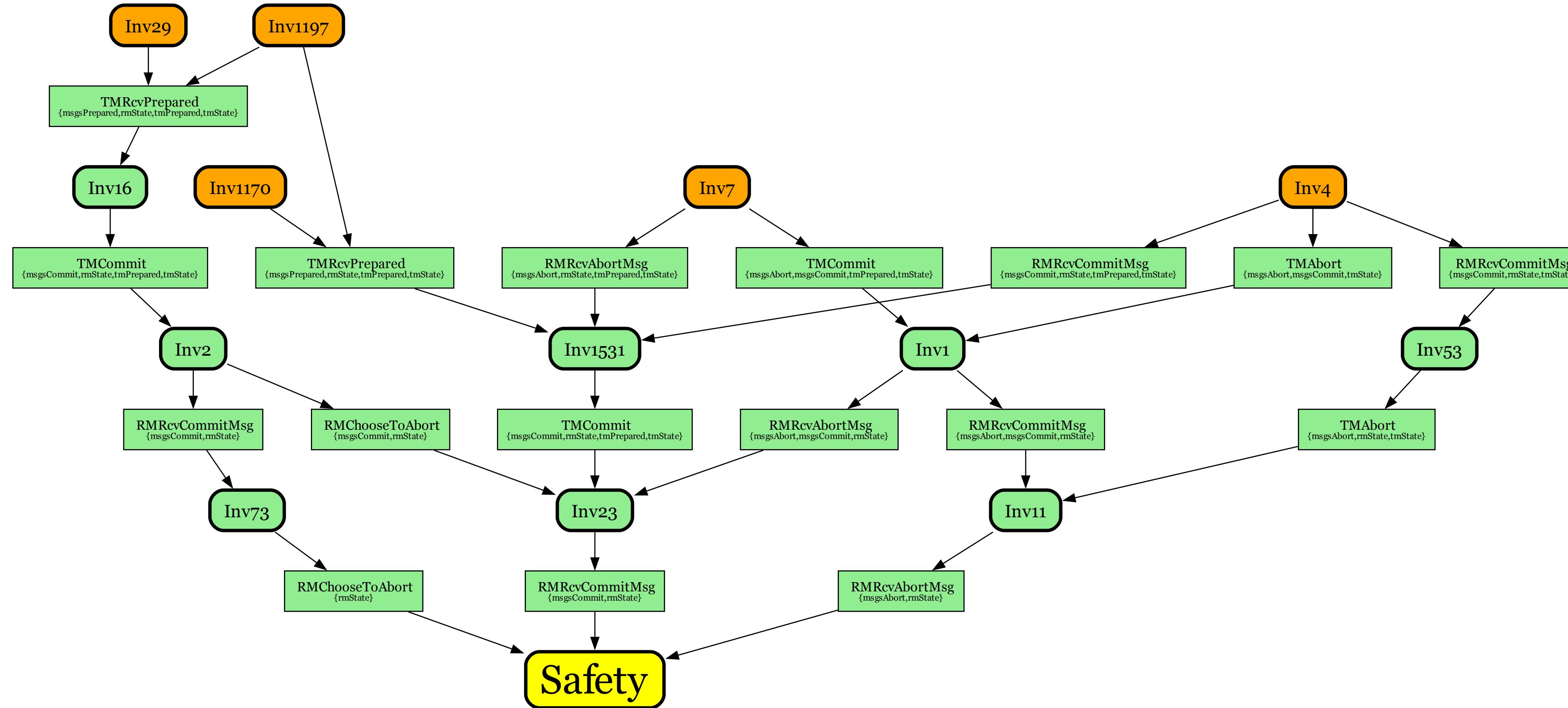


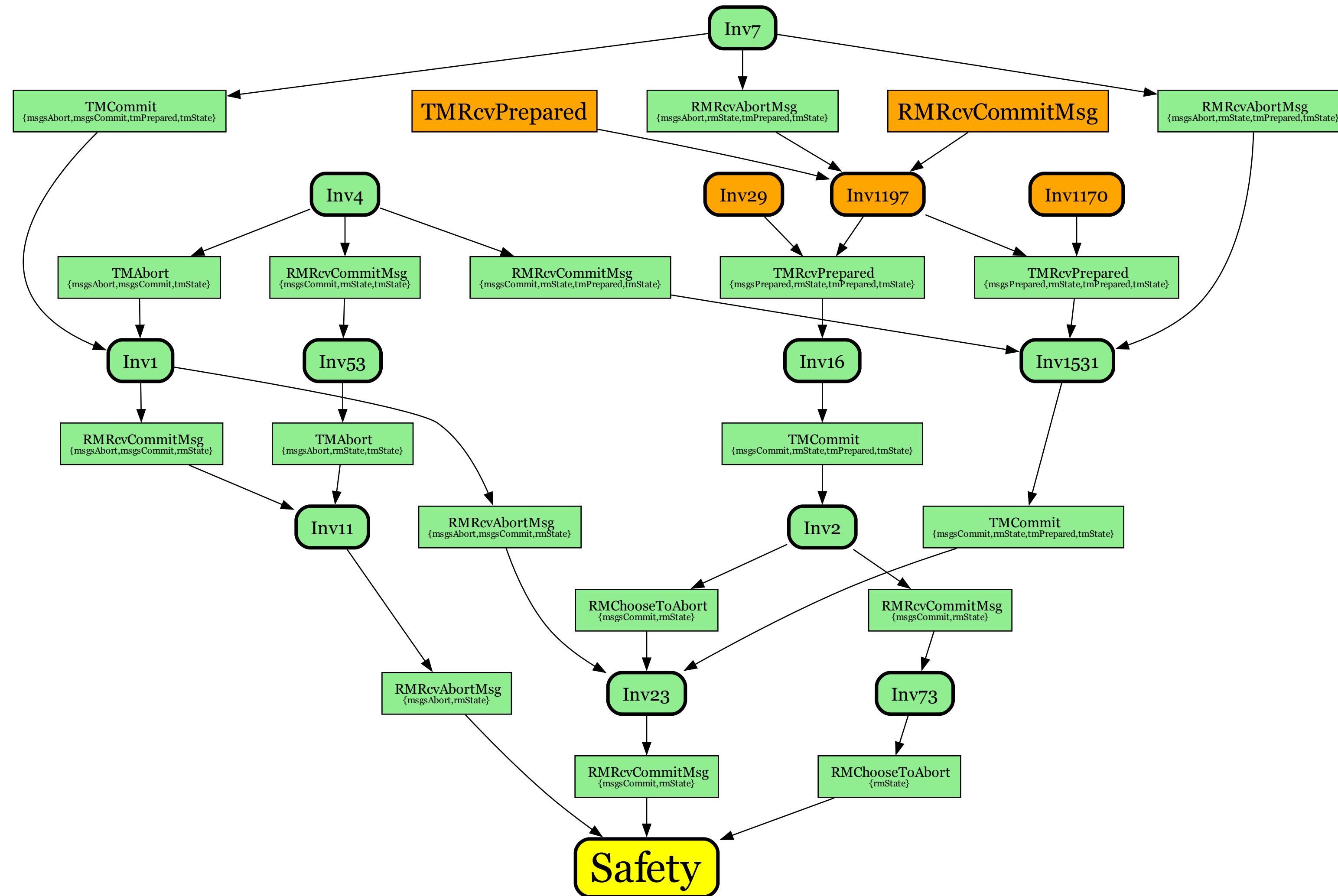


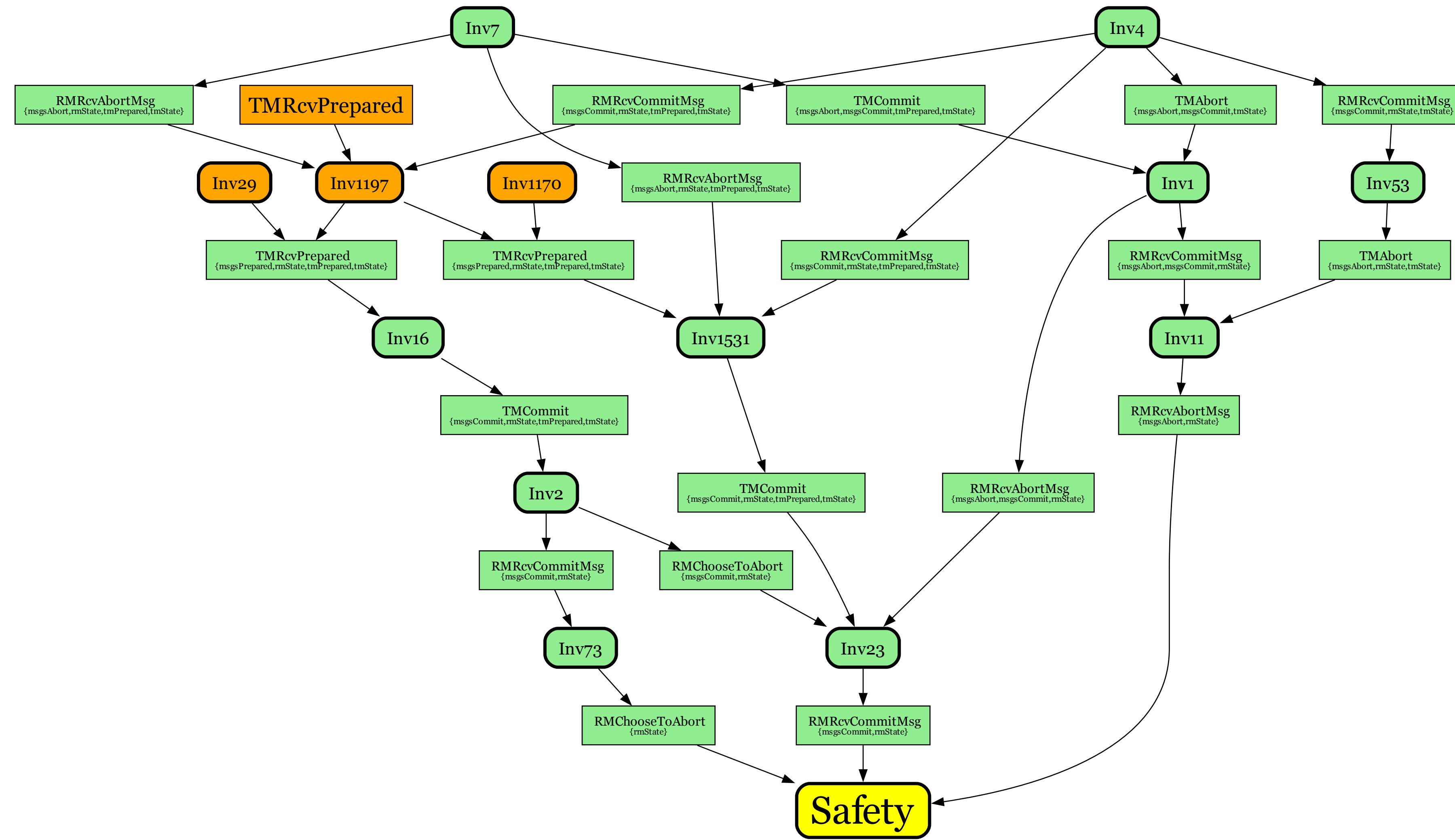


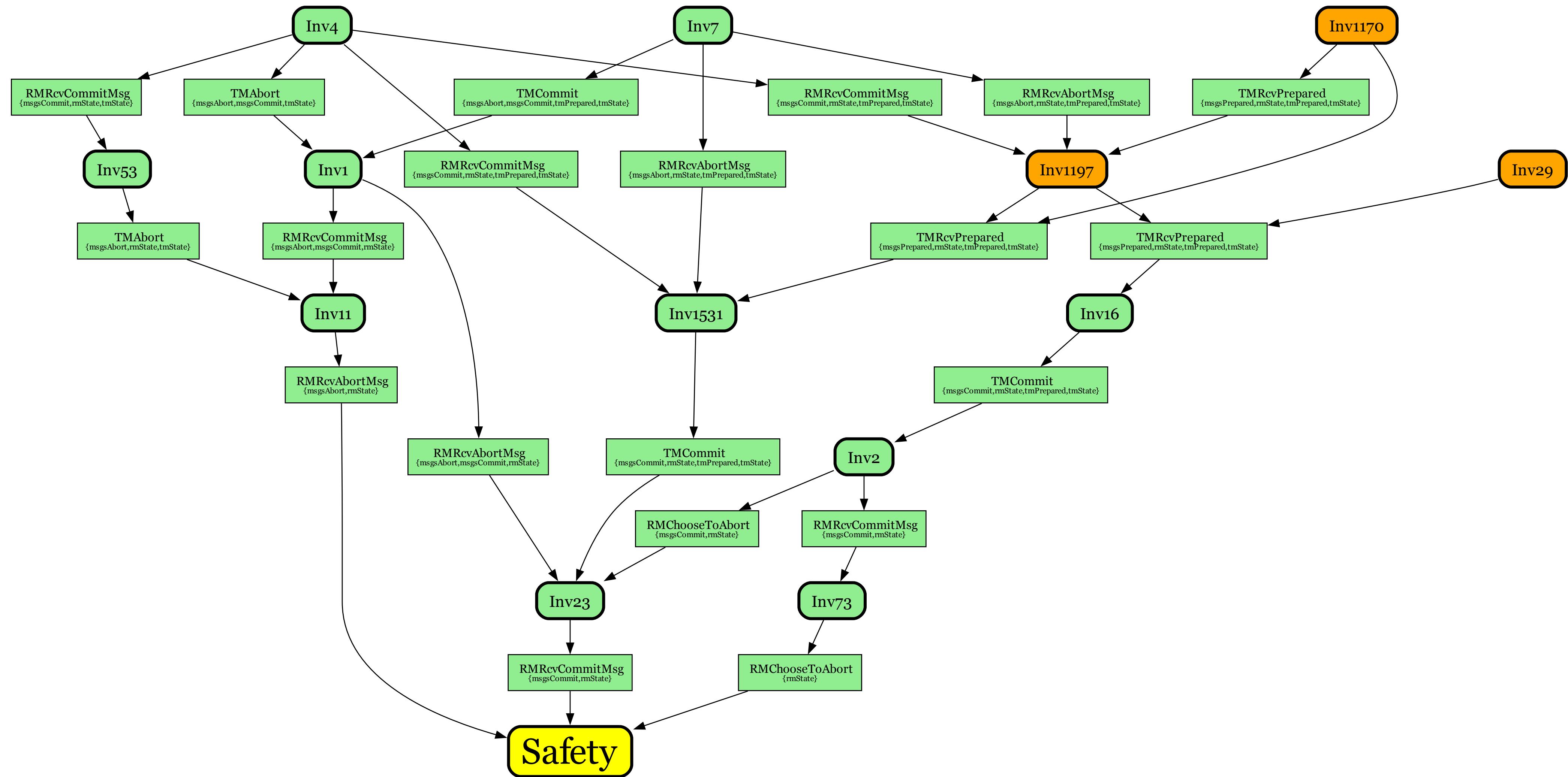


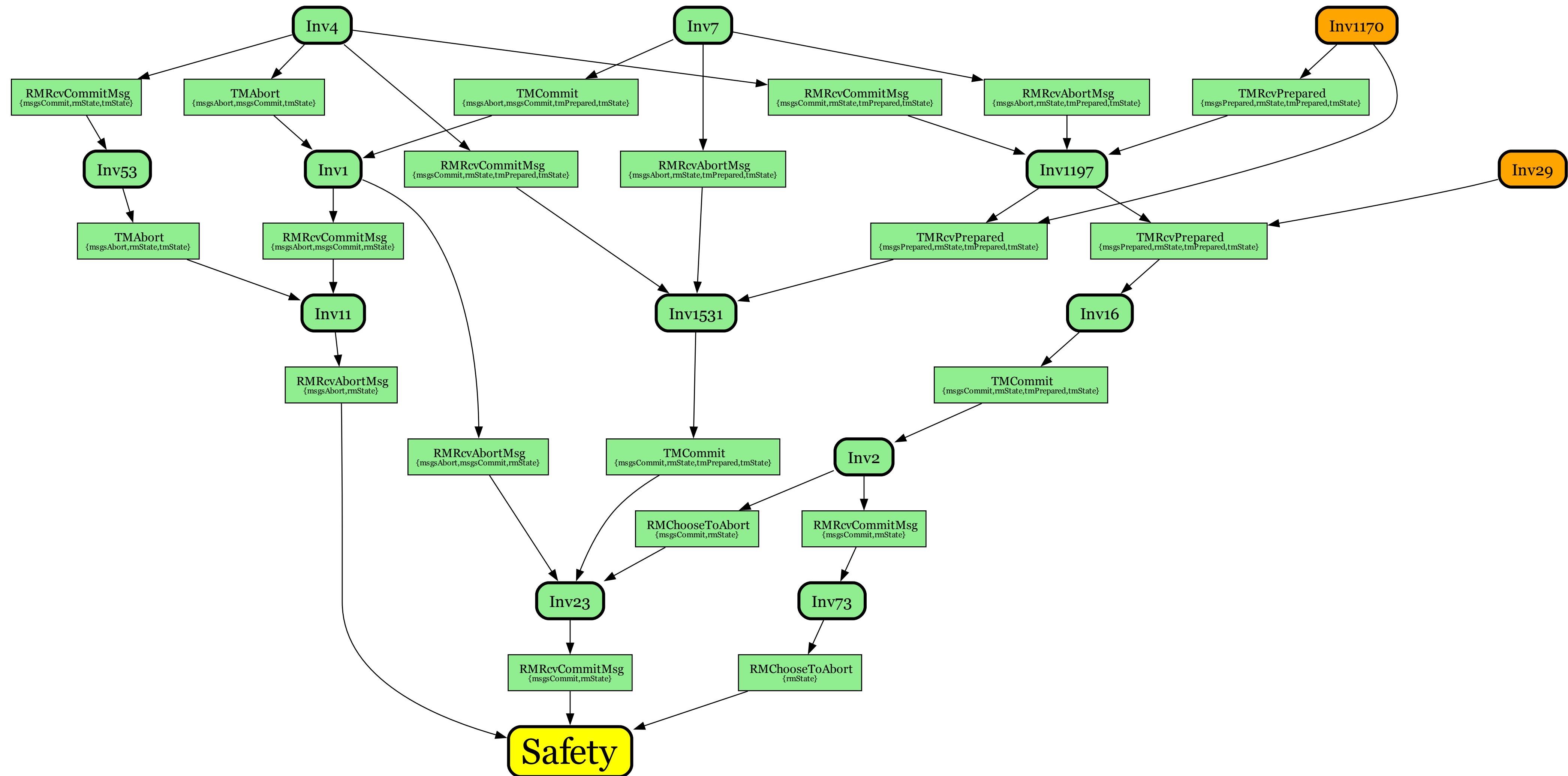


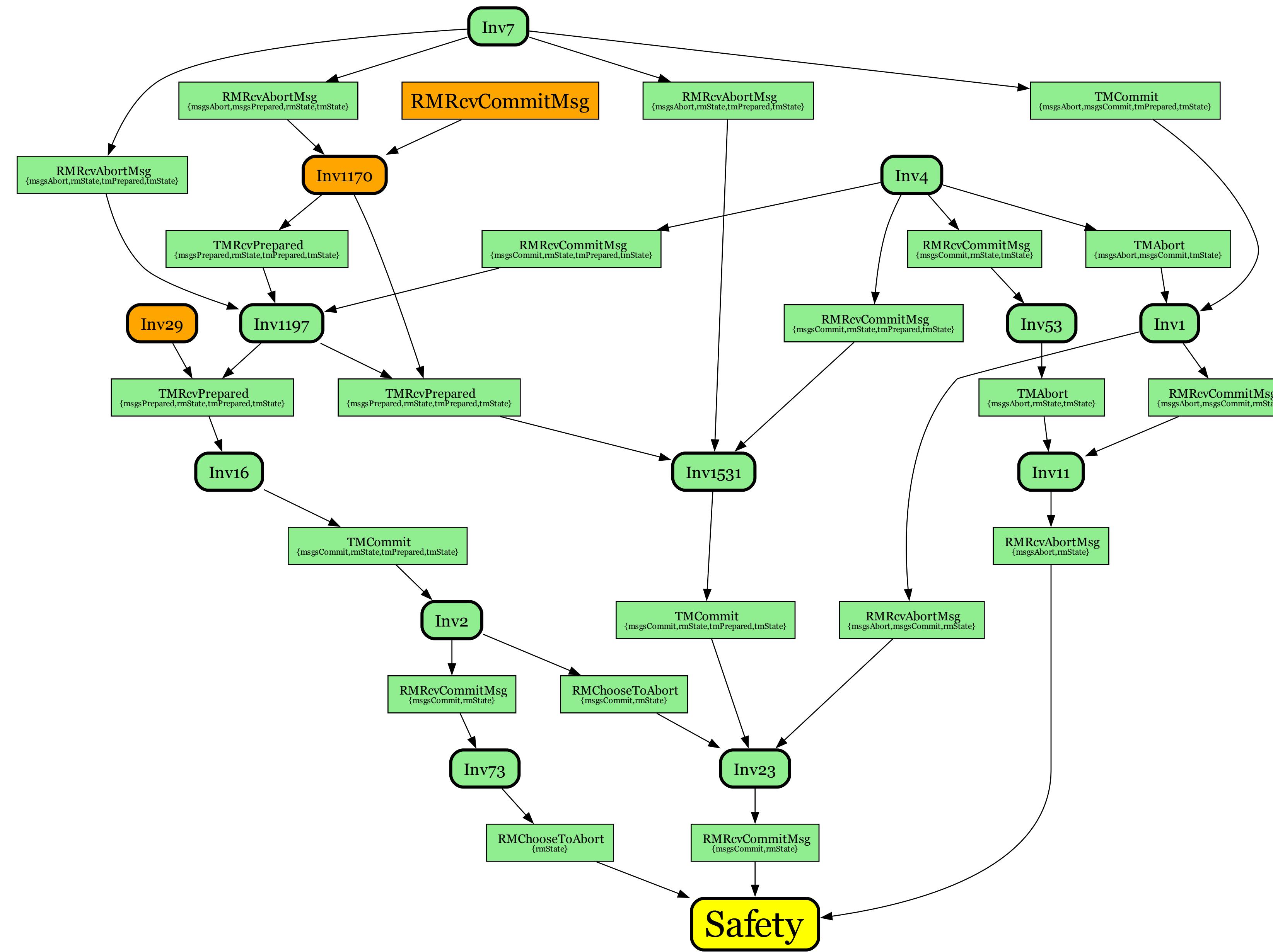


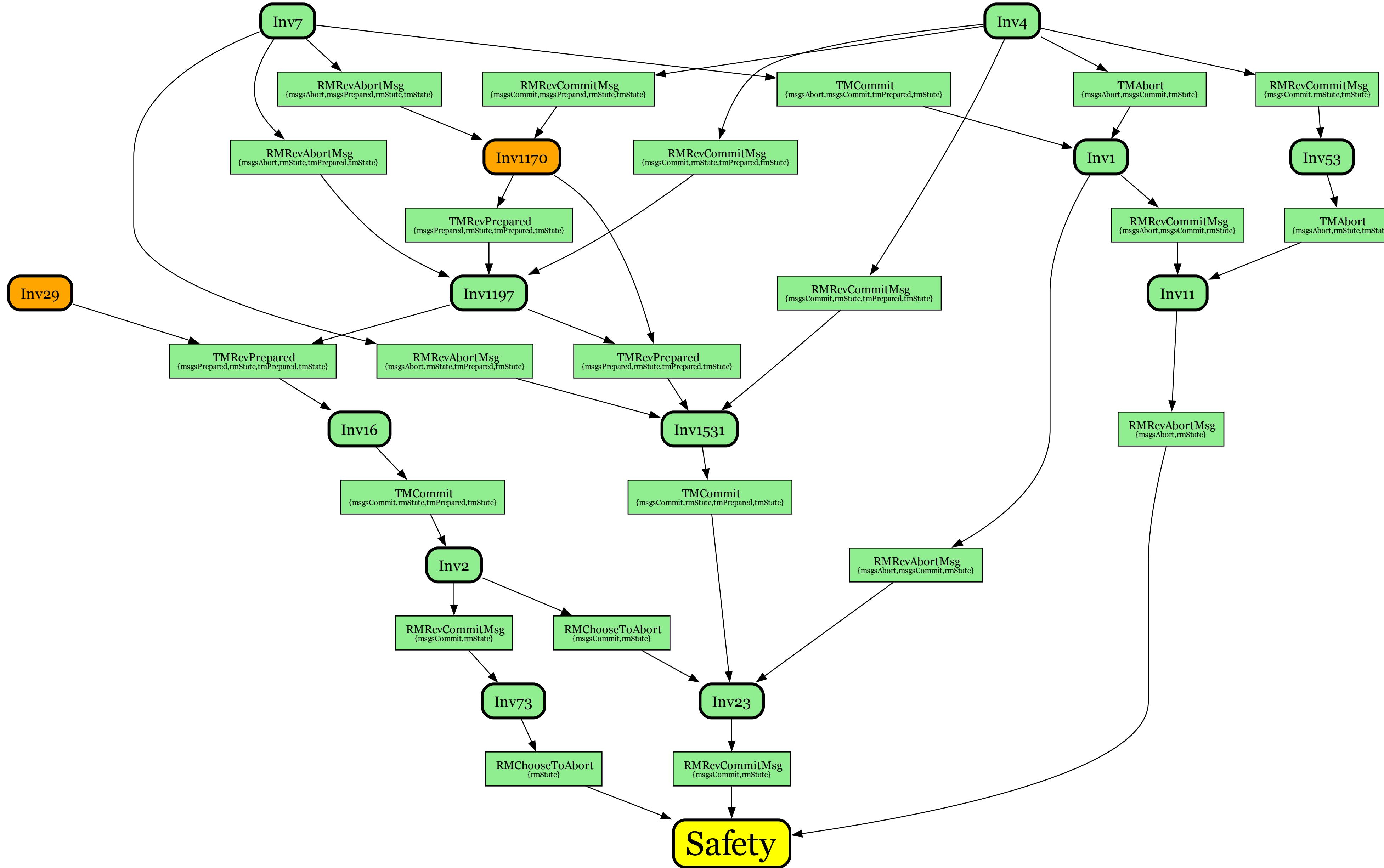


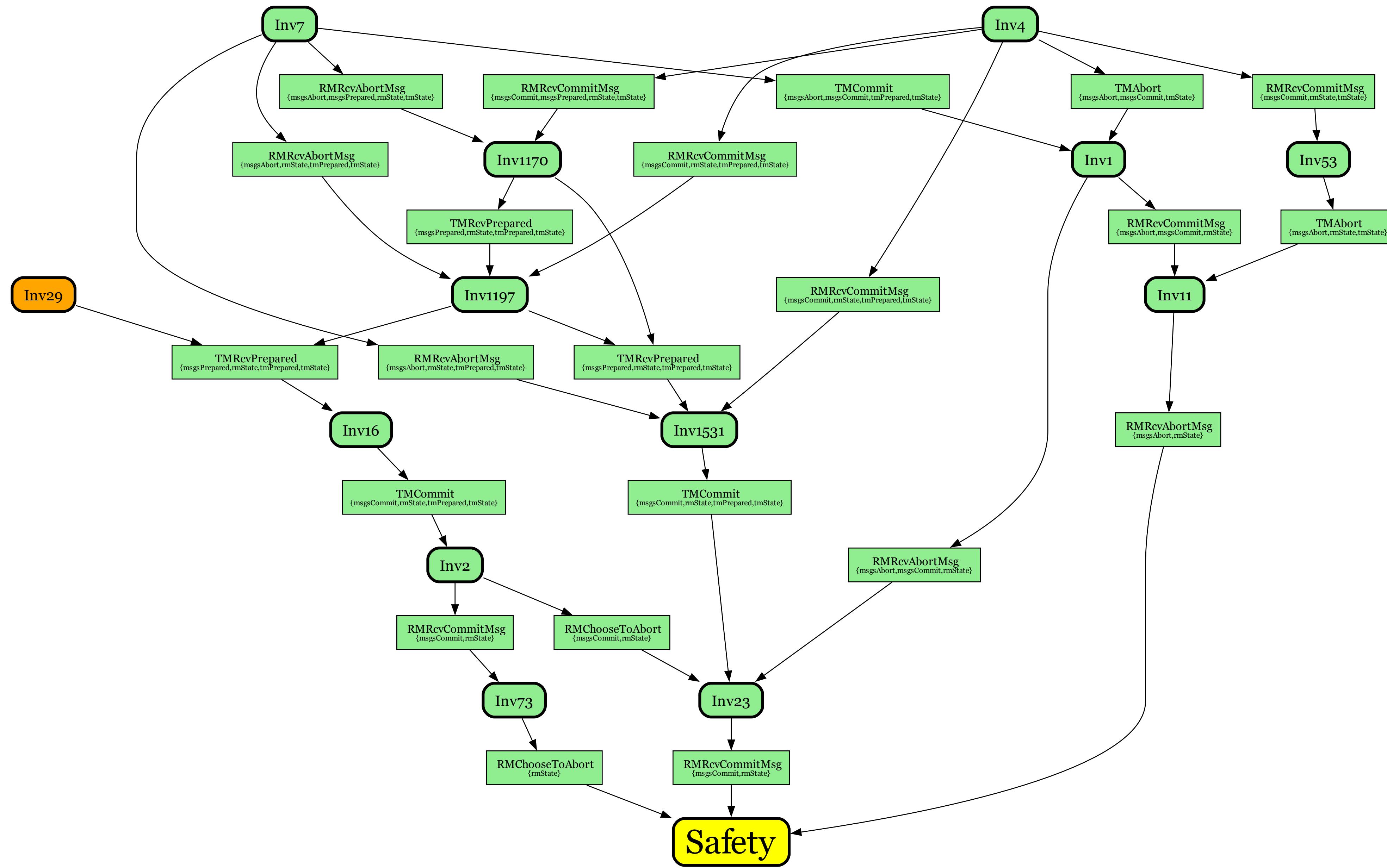


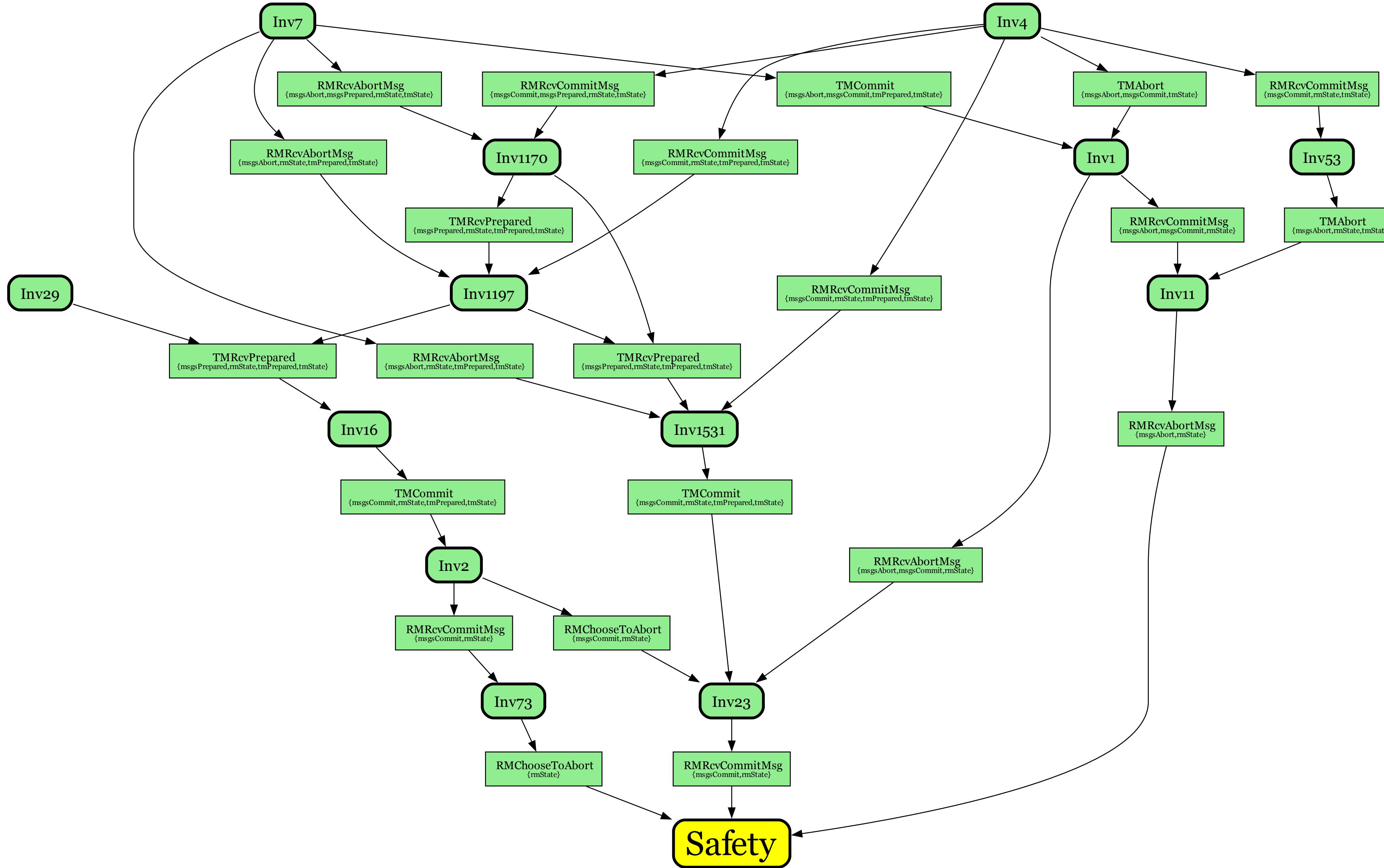












# Inductive Proof Graph Synthesis

Implemented technique for synthesizing proof graphs into a tool, ***scimitar***, which accepts TLA+ protocol specifications.

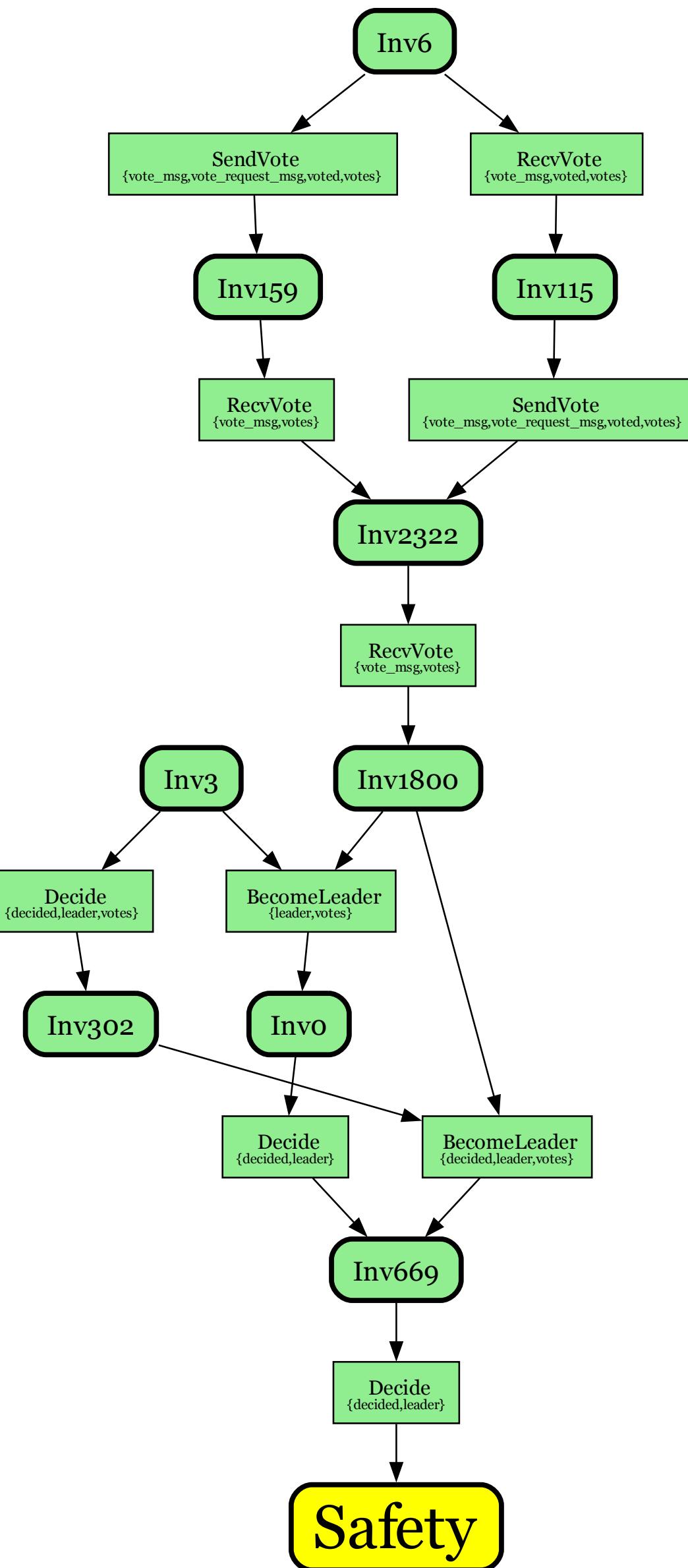


Uses the TLC model checker and TLA+ proof system (TLAPS) for internal verification routines.

Syntax-guided technique for synthesizing lemmas. Guided by counterexamples to induction (CTIs) at each local inductive proof obligation.

[github.com/will62794/scimitar](https://github.com/will62794/scimitar)

# SimpleConsensus



Simplified leader-based consensus algorithm.

10 lemma nodes.

6 state variables, 5 actions.

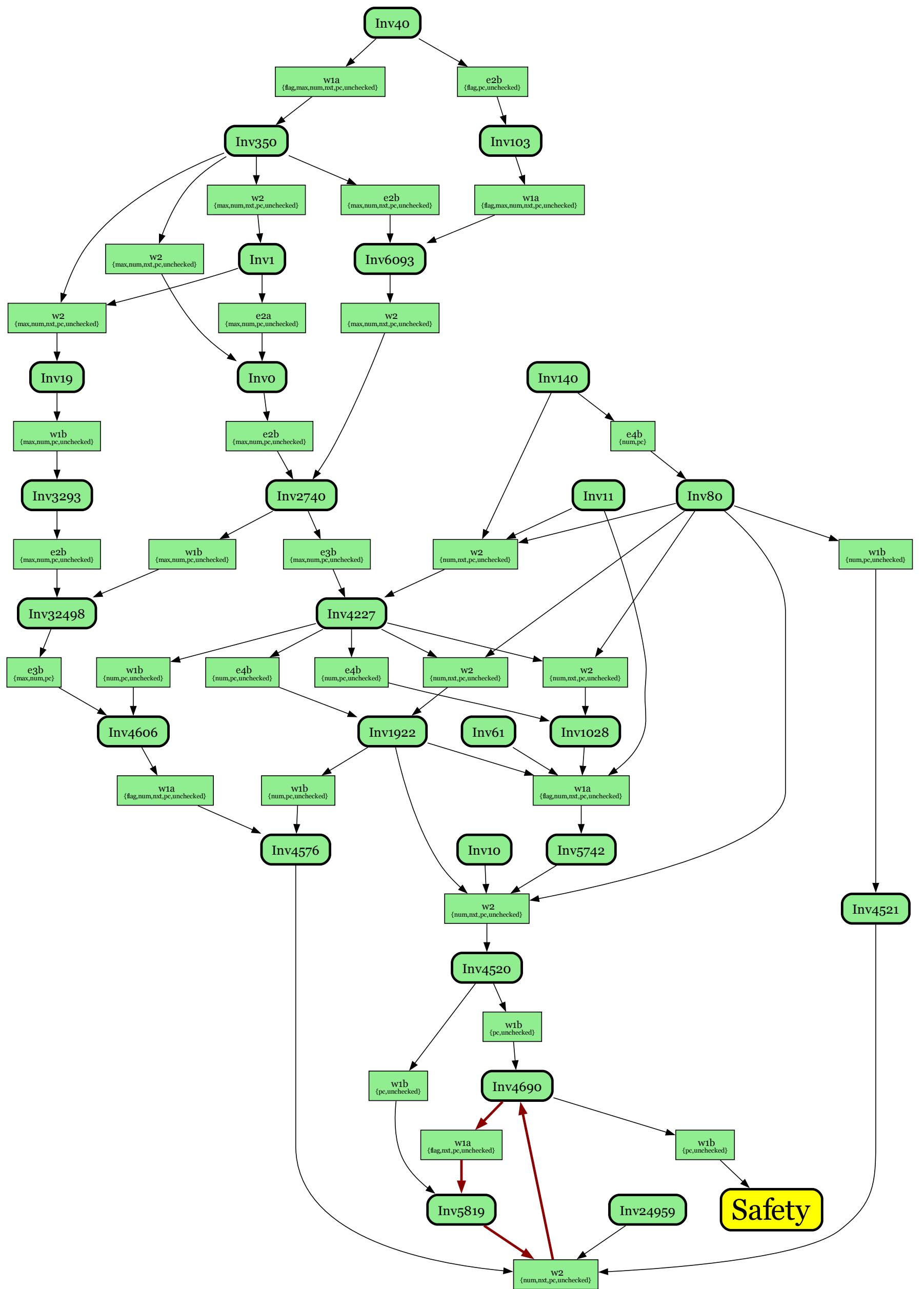
Synthesized in ~5 minutes.

*VARIABLE*  
 $vote\_request\_msg$ ,  
 $voted$ ,  
 $vote\_msg$ ,  
 $votes$ ,  
 $leader$ ,  
 $decided$

*Safety*  $\triangleq$

$\forall n_1, n_2 \in Node, v_1, v_2 \in Value :$   
 $(v_1 \in decided[n_1] \wedge v_2 \in decided[n_2]) \Rightarrow (v_1 = v_2)$

# Bakery



Lamport's concurrent  
Bakery algorithm for  
mutual exclusion.

27 lemma nodes

6 state variables, 14  
actions.

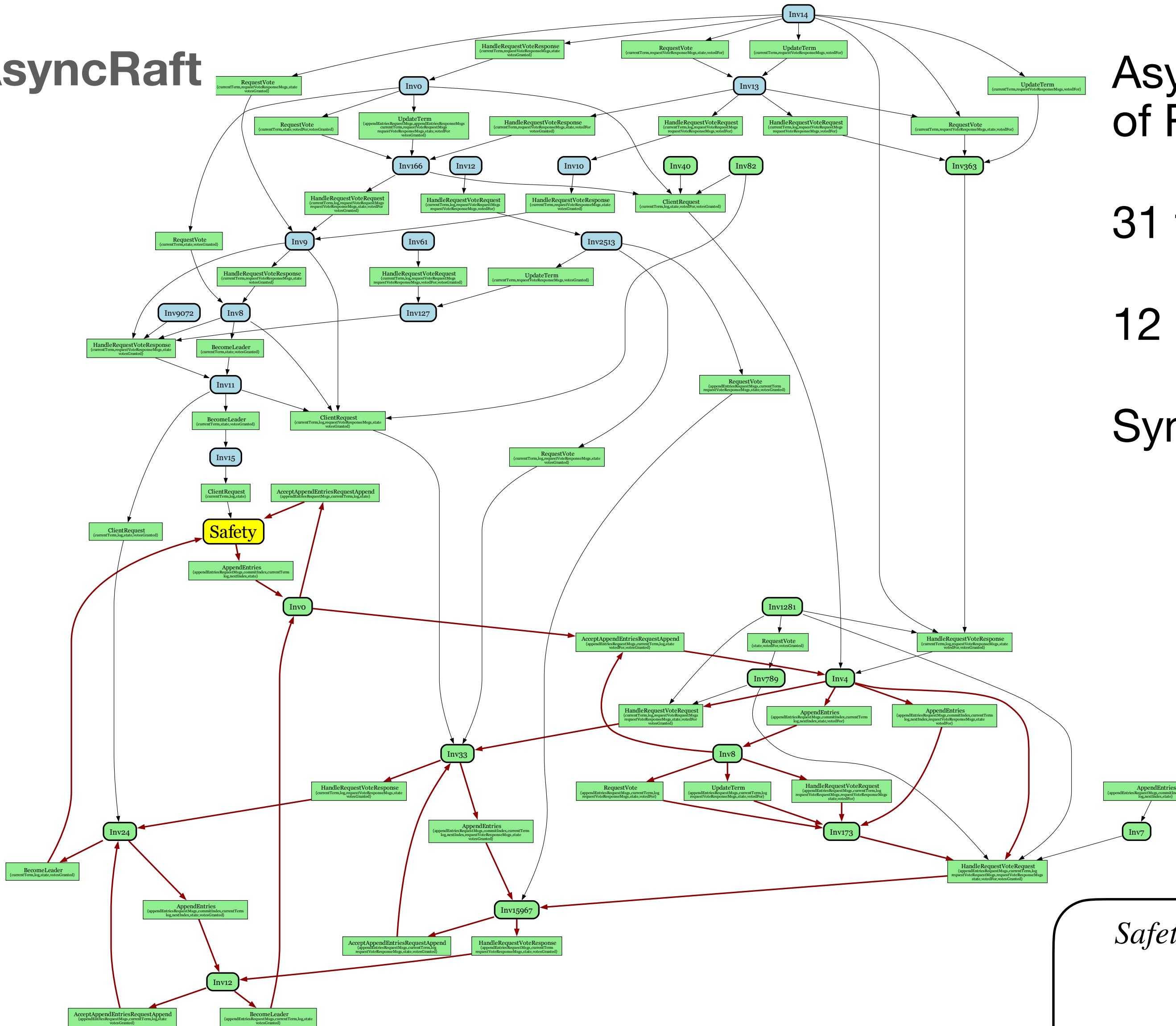
Synthesized in ~ 4.5  
hours

VARIABLES  
*num*,  
*flag*,  
*pc*,  
*unchecked*,  
*max*,  
*nxt*

Mutual exclusion property.  
*Safety*  $\triangleq$

$$\forall i, j \in \text{Procs} : (i \neq j) \Rightarrow \neg(\text{pc}[i] = \text{cs} \wedge \text{pc}[j] = \text{cs})$$

# AsyncRaft



Asynchronous specification  
of Raft consensus.

31 total lemma nodes.

12 state variables, 9 actions.

Synthesized in ~14 hours.

**VARIABLES**

- requestVoteRequestMsgs*
- requestVoteResponseMsgs*
- appendEntriesRequestMsgs*
- appendEntriesResponseMsgs*
- currentTerm*
- state*
- votedFor*
- log*
- commitIndex*
- votesGranted*
- nextIndex*
- matchIndex*

*Safety*  $\triangleq$

$\forall i, j \in Server :$

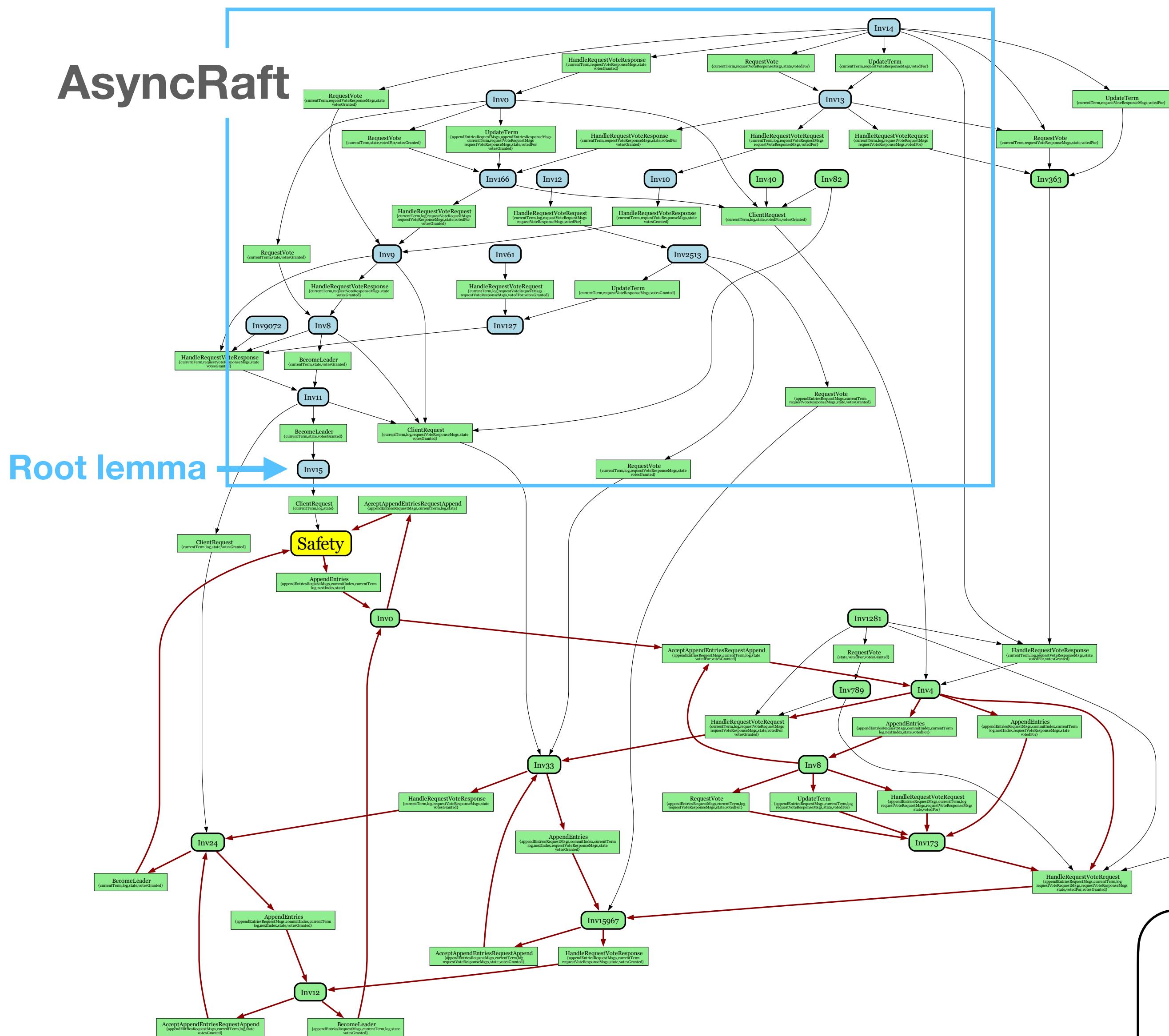
$(state[i] = Leader) \Rightarrow$

$\nexists k \in DOMAIN log[j] :$

$\wedge (log[j][k] = currentTerm[i]$

$\wedge \nexists ind \in DOMAIN log[i] : (ind = k \wedge log[i][k] = log[j][k]))$

# Election safety support subgraph.



Asynchronous specification  
of Raft consensus.

31 total lemma nodes.

12 state variables, 9 actions.

Synthesized in ~14 hours.

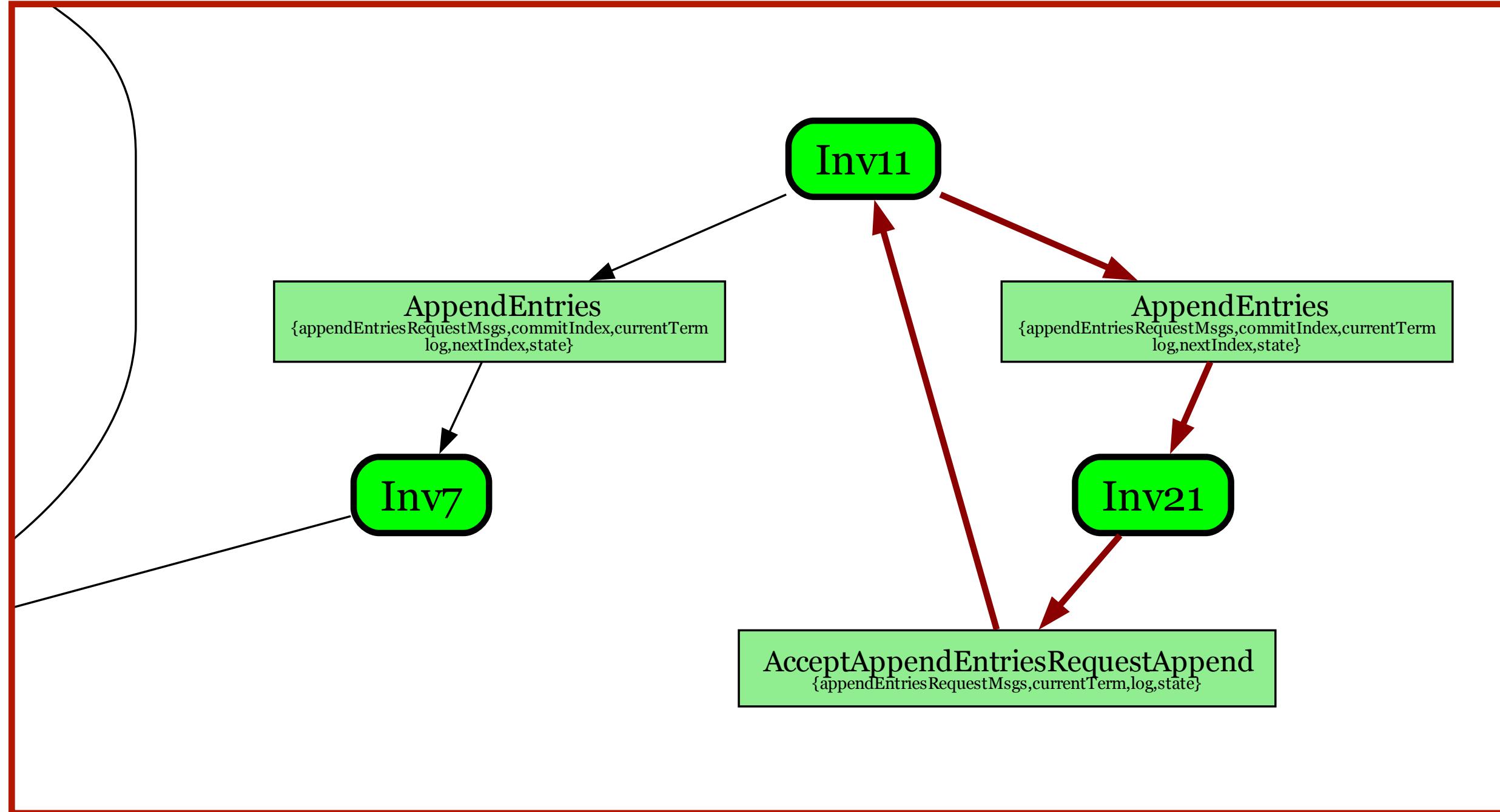
Message induction cycle.

$$\begin{aligned}
 Safety \triangleq & \\
 \forall i, j \in Server : & \\
 (state[i] = Leader) \Rightarrow & \\
 \exists k \in DOMAIN log[j] : & \\
 \wedge (log[j][k] = currentTerm[i]) & \\
 \wedge \nexists ind \in DOMAIN log[i] : (ind = k \wedge log[i][k] = log[j][k])
 \end{aligned}$$

**VARIABLES**

- requestVoteRequestMsgs*
- requestVoteResponseMsgs*
- appendEntriesRequestMsgs*
- appendEntriesResponseMsgs*
- currentTerm*
- state*
- votedFor*
- log*
- commitIndex*
- votesGranted*
- nextIndex*
- matchIndex*

## AsyncRaft - Message Induction Cycle Motifs



Induction cycles in message passing distributed protocols.

Invariants that must hold in *local* state and also *network* state.

Manifests as cycles in these proof graphs.

$$Inv11 \triangleq \forall i \in Server : \forall k \in DOMAIN \ log[i] : log[i][k] \leq currentTerm[i]$$

$$\begin{aligned} Inv21 \triangleq \\ \forall m \in appendEntriesRequestMsgs : \\ \neg(m.mentries \neq \langle \rangle \wedge m.mentries[1] > m.mterm) \end{aligned}$$

# Conclusions & Future Work

*Inductive proof decomposition*: a compositional, interpretable approach to inductive invariant development.

In future, more exploration of empirical structure of these graphs.

Other compositional features for synthesis acceleration

- Quantifier structures
- Local grammar tuning, etc.

**Tool:** <https://github.com/will62794/scimitar>

**Paper draft:** <https://will62794.github.io>