Revisiting the AMBA AHB bus case study

Ioannis Filippidis Richard M. Murray

May 2, 2015

Abstract

This report describes a number of changes to the ARM AMBA bus case study from [1] that lead to significant reduction in synthesis time. In addition, it identifies the reason of blowup for the synthesized strategies in earlier studies as lack of binary decision diagram (BDD) reordering during strategy construction. Enabling dynamic BDD reordering with the group sifting algorithm, we synthesized strategies for as many as 18 masters, with both the original and revised specifications. This conclusion is based on detailed experimental measurements that show the changes of BDD sizes over time for the fixpoint and other variables during the nested fixed point computation, including the cumulative time spent on BDD reordering and the total number of BDD nodes. The measurements were obtained for eight different cases, allowing to compare the original with the revised specifications, with strategy reordering enabled or not, and conjoining the weak fairness guarantees or merging them in a single Büchi automaton. The revised specification proposed here is expressed using the open Promela language.

Contents

1 Introduction

The ARM processor Advanced Microcontroller Bus Architecture (AMBA) [2] specifies a number of different bus protocols. Among them, the Advanced High-performance (AHB) architecture has been studied extensively in the reactive synthesis literature [1, 3, 4, 5, 6, 7, 8].

The AHB bus comprises of masters that need to communicate with slaves, and an arbiter that controls the bus and decides which master is given access to the bus. The arbiter receives requests from the masters that desire to access the bus, and must reposnd in a weakly fair way. In other words, every master that keeps uninterruptedly requesting the bus must eventually be granted access to it. Note that the AMBA technical manual does not specify any requirement on fairness, but instead leaves to the designer. For automated synthesis, weak fairness is one possible formalization that ensures servicing of all the masters.

In addition, a master can request that the access be locked. However, the arbiter makes no promises as to whether a request for the lock will be granted.

A specification for the arbiter appeared in [1] and is presented in detail in [5]. In [7], the authors formalize also the specifications for masters and slaves connected to the bus. Here, we revise the specification of [5], and express it in the open PROMELA language.

The original specification includes some properties that are not in the GR(1) fragment. In [5], these properties are translated to deterministic Büchi automata, by introducing auxiliary variables for representing the nodes of the automaton. These variables are added to the problem's alphabet. The resulting formulae are much less readable, and not easy to modify and experiment with.

Here, we specify these properties directly as processes (transition systems), with progress states where needed.

During expression of the specification in the PROMELA language, a choice had to be made about the representation of the fairness requirements. In the original specification, these take the form of a conjunction of recurrence formulae

$$\bigwedge_{i=0}^{N-1} \Box\Diamond(request[i] \rightarrow master = i).$$

It is possible to rewrite this conjunction as a Büchi automaton (BA) that checks each fairness condition, one after the other. The property described by the BA is *equivalent* to that described by the conjunction. This change reduces the number of recurrence goals from N to 1. In the GR(1) synthesis algorithm [9], this reduces the number of inner fixed point computations from N to 1.

An initial motivation for this modification was to obtain a specification parameterized by the domain of a counter variable (local to the BA process), which avoids the need to regenerate the specification from an auxiliary script (as has been typically the case in similar studies).

Another motivation was to explore the design space, and the sensitivity of the specification, by varying the amount of detail that was specified. In particular, we were interested in observing whether the runtimes improve significantly in case that the order of goals was given and fixed. Note that the Büchi automaton above does *not* fix the order that goals have to be satisfied. This variant was an initial attempt that placed more constraints on the design, to see whether those "hints" to the synthesizer had a significant effect, or not.

This experimentation revealed a notable difference between the two alternatives. When using a Büchi automaton, there is a single liveness goal. So a single sub-strategy is synthesized for this goal, and it need not be combined to other sub-strategies, as there are none. In contrast, when

there are multiple liveness goals, the individual sub-strategies need to be combined into a single one. The process of combining the sub-strategies is disruptive to the BDD variable ordering. As a result, it is impossible to scale synthesis using a conjunction, without enabling dynamic reordering *during* the synthesis of the final, monolithic strategy. Reordering during strategy construction is not necessary, and in fact has a negative effect, if the liveness goals are represented as BA. The trade-off is introducing auxiliary variables (a counter) in the state space that are used to represent the nodes of the BA. Together with the sequencing of goals that the BA represents, this leads to longer runtimes, but scales without difficulty to a larger number of masters, without any need for reordering during strategy construction.

The revised specification is given in listing Listing 1, where some additional changes involve weakening the assumptions, and modifying assumption A1. An instance of the original specification is given in listing Listing 2. In [5], assumption A1 requires that for locked undefined-length bursts, masters eventually withdraw their request to access the bus. This assumption is not explicit in the ARM standard, so we modify it, by requiring that masters withdraw only their request for the lock, *not* for bus access. Weakening the assumptions also allowed abstracting the array $HLOCK$ of N lock requests by the two bits that are only referenced in it. Another difference is that a Mealy game is solved here, and a Moore in [5].

In Section 2, we describe how the specification was revised, referring to the technical ARM specification. In Section 3, we present the experiments, measurements, observations and conclusions based on the measurements and the GR(1) synthesis algorithm.

Acknowledgments This work was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

2 Revising the formal specification

Declarative specifications for this case study were presented in [1, 3, 5, 7]. These papers do not mention the processors used as hardware, so the difference in processor clock speed is unknown. As a result, the speedups can be compared only as absolute time intervals.

We base our specifications on those of [5]. In that case study, each requirement was obtained either:

- by formalizing the AMBA AHB standard, or
- by adding some desired (e.g., fair arbiter) or auxiliary properties (e.g., auxiliary scheduling variables `start`, `decide`), that are left unspecified in the standard.

When initially formalized in LTL, some of the requirements yield formulae that are not in the GR(1) fragment. Nonetheless, there do exist deterministic Büchi automata for these particular properties. The next step in [5] is to translate these properties to (symbolic) deterministic Büchi automata, whose states are represented by auxiliary variables that become part of the problem's alphabet. This translation is not always trivial.

The translation step can be done either manually or automatically. However, with automatic translation the user has little control over the form of the determinized automaton. This ability to modify the automaton itself proves important during development and, especially, while debugging the specification. By changing directly the automaton, one can observe what effects small modifications have on realizability, and understand better the problem's structure.

Another aspect of writing LTL formulae, instead of attempting to express the design intent as an automaton, is that the former may require nesting of operators, which quickly becomes unmanageable. Writing imperatively a Büchi automaton for the same property can prove easier. Moreover, as the reduction of progress conditions in the AMBA example demonstrates, the availability of imperative elements can *encourage* modifications that significantly reduce the problem's complexity, while preserving the required property.

For this reason, describing directly automata can benefit the synthesis process.

The properties from [5] have been either:

- changed, because they did not correspond to the ARM standard, or
- expressed *directly* as automata, in case they do not belong to GR(1) (similarly to what was done in [5] by defining the automata directly in LTL), or
- equivalently expressed as automata, because this allows significant reduction in synthesis time, or
- reformulated to an equivalent, but syntactically simpler formula, in several cases due to the availability of array arithmetic, or
- remained unchanged.

The correspondence is noted inline, for example, "G 5" is the guarantee with index 5 in [5]. In what follows, we consider each specification in incremental order, using the numbering in [5]. Also, note that a Mealy game is solved here, whereas a Moore game is solved in [5].

2.1 A1 vs ARM standard

Consider A1, stating that for locked indefinite duration bursts, the master that controls the bus must eventually withdraw its request for *access* to the bus. In logic, this property is expressed as:

$$\Box\Big(\big(HMASTLOCK \wedge (HBURST = INCR)\big) \rightarrow \bigcirc\Diamond\neg HBUSREQ[HMASTER]\Big) \tag{1}$$

The ARM standard states that:

- Sec.3.11.2, paragraph 2, p.3-29 and Sec. 3.11.4, p.3-33: A master can lose ownership of the bus early, because the arbiter decides to limit their access time.
- Sec.3.11.2, paragraph 3, p.3-29 and Sec. 3.11.5, p.3-34: If a master requests both:
  1. bus access (by setting $HBUSREQ[i]$ HIGH), and
  2. locking of its access (by setting $HLOCK[i]$ HIGH)

  then: if the arbiter decides to give to the master the bus, *and* lock the access, then the arbiter guarantees that it will maintain the lock,

  > "until the *locked sequence* has completed".

The question becomes what completion of a locked sequence means. If a sequence has predefined length (determined by *HBURST* and some other signals), then the arbiter knows how long the request will last (G3 addresses this for length-4 locked bursts).

However, for indefinite length bursts, the arbiter does not know a priori when the locked sequence will terminate. In [5], the "locked sequence" is interpreted as the time interval until the master who was granted locked access stops requesting bus *access*. The master withdraws its request for access by setting $HBUSREQ[HMASTER]$ to false. As a consequence, the progress assumption A1 requires that a master who requested indefinite locked access, eventually withdraw its request to communicate.

However, it is not clear from the standard whether this behavior is intended. In Table 2-2, entry *HLOCK*, p.2-5, it is stated that

> "When HIGH, this signal indicates that the master requires locked access to the bus, and no other master should be granted the bus *until* this signal is LOW."

The "until" above suggests that the arbiter *can* grant the bus to *another* master, *after* master j deasserts the signal $HLOCK[j]$. This implies that deassertion of $HLOCK[j]$ by the current master suffices to allow the arbiter to reassign the bus. It does not refer to the signal $HBUSREQ[j]$.

2.2 Weakening A1

Based on the previous note, we can arrive at a weaker assumption, that decouples the:

1. lock requests *HLOCK*, from the
2. bus access requests *HBUSREQ*.

As soon as a master withdraw its request for locking its access, the arbiter is allowed to assign the bus to another master. This does not oblige the current master to withdraw its request for bus access. So the master is free to keep requesting bus access ($HBUSREQ$), but is obliged to stop requesting the lock ($HLOCK$). In other words, a master is not allowed to lock the bus forever (but may keep requesting bus access forever, without ever lowering $HBUSREQ[j]$, in agreement with Sec. 3.11.2, paragraph 1, p.3-29 "A bus master …may request the bus during any cycle").

The new assumption A1 is

$$\Box((HMASTLOCK \wedge (HBURST = INCR)) \rightarrow \bigcirc\Diamond\neg HLOCK[HMASTER]) \tag{2}$$

We will remove also $(HBURST = INCR)$, for reasons explained later.

Note that up to here, this allows the master to stop requesting bus access (LOW $HBUSREQ$), but keep requesting the lock (during an indefinite length burst). Sec 3.11.2, paragraph 6, p.3-29 assumes that the master never does this. It is formulated as assumption A3. As commented later, we remove assumption A3, because it is not necessary for realizability (and simplifying it has no adverse effect – weakening of assumptions is desirable).

2.3 Updating G2

The replacement of $HBUSREQ[HMASTER]$ by $HLOCK[HMASTER]$ in A1 implies that the same replacement applies to G2 also.

2.4 A3 becomes trivially true

The replacement of $HBUSREQ[HMASTER]$ by $HLOCK[HMASTER]$ in A1 allows each master i to keep $HBUSREQ[i]$ always HIGH. As a result, assumption A3 does not constrain the environment in a way essential for realizability (since the environment can set HIGH all the elements of array $HBUSREQ$ forever). Therefore, we drop assumption A3.

2.5 Coupling between A1 and G2

2.5.1 Coupling

In this section, we analyze the dependence of G2 on A1. The conclusion motivates simplifying A1.

An initial (wrong) attempt to express the LTL property A1 in PROMELA (without using Fig.9(a) [5]) resulted in the code

```
assume active env proctype withdraw_lock(){
    progress:
    do
    :: lock && (burst == INCR);
        do
        :: ! master_lockreq; break
        :: true /* wait */
        od
    :: else
    od
}
```

We have used `master_lockreq` in place of $HMASTLOCK[HMASTER]$, for reasons explained later.

The correct automaton for property A1 is Fig.9(a), and is expressed by the code

```
assume active env proctype withdraw_lock(){
progress0:
    do
    :: lock && (burst == INCR); goto S1;
    :: else
    od;
    S1:
    do
    :: (! master_lockreq) && ! (lock && (burst == INCR));
        goto progress0
    :: (! master_lockreq) && (lock && (burst == INCR));
        goto progress1
    :: else
    od;
    progress1:
    do
    :: master_lockreq; goto S1;
    :: (! master_lockreq) && (lock && (burst == INCR));
    :: else;
        goto progress0
    od
}
```

The specification with the first automaton as A1 is unrealizable. Using the second automaton, the specification is realizable.

Consider the first automaton and the automaton for G2:

```
assert active sys proctype maintain_lock(){
    do
    :: lock && start && (burst == INCR);
        do
        :: ! start && ! master_lockreq; break
        :: ! start
        od
    :: else
    od
}
```

Each of them has two states: an "initial" one (outer `do`), and a "waiting" one (inner `do`). The product of the two automata has 4 states:

State 0: both automata at the initial states. The system has not yet lost at this state.

State 1: both automata at the "waiting" states. As soon as the environment sets `master_lockreq`, the system returns to the initial state and is released, so that it can start a new bus access later. (The environment also returns to its initial state).

State 2: the environment at the initial state, and the system at the waiting state. The system has lost, because the initial A1 state is a progress state, and the environment can keep *HBURST* different from *INCR*, to remain at its initial state. So the environment is not obliged to withdraw the lock request `master_lockreq`, leaving the system indefinitely trapped at its waiting state.

State 3: the system at the initial state, and the environment at its waiting state. Note that state 3 is reachable, because the environment can reach its waiting state as soon as `lock && (burst == INCR)`, whereas the system has to wait until `start`, and `start` can be delayed arbitrarily long by the environment keeping `ready` LOW (G1).

If the system transitions to its waiting state, but the environment remains at its waiting state, then state 1 is reached. If the environment transitions to its initial state, but the system remains at its initial state, then state 0 is reached. These two transitions cause no problems.

The transition that makes the system lose is if the environment returns to its initial state, and the system transitions to its waiting state, both at the same time. If this happens, then state 2 is reached, and the system has lost. This transition is possible, only if `burst == INCR` can hold when `! master_req`.

This execution is not possible with the automaton of Fig.9(a), because in order for the environment to return to `progress0` there, it must set both `burst != INCR` *and* `! master_req` at the same time. This implies that the environment will return to its initial state, without letting the system transition from its initial, to its waiting state. So it prevents the transition from state 3 to the losing state 1.

2.5.2 Modifying A1 to remove the coupling

The coupling analyzed previously requires that the original automaton be used for assumption A1 (which can be rewritten, using structured programming constructs). The resulting specification is more fragile (for example, the initial attempt at manually writing an automaton contained the error above). Moreover, it results in longer synthesis time.

For these reasons, we simplify assumption A1 to

$$\square(HMASTLOCK \rightarrow \bigcirc\Diamond\neg HLOCK[HMASTER]) \tag{3}$$

Using the replacement bit variable `! master_req`, this is equivalent to the (much simpler) automaton

```
assume active env proctype withdraw_lock(){
    progress:
    do
    :: lock;
        do
        :: ! master_lockreq; break
        :: true /* wait */
        od
    :: else
    od
}
```

Note that the only modification from the erroneous initial attempt is to remove the conjunct `burst == INCR` from the guard `lock && (burst == INCR)`.

The modified assumption A1 states that if the current master is granted the lock, then next it must eventually withdraw the locking request. This assumption is reasonable, because no master that has been granted locked bus access is entitled to continue requesting the lock indefinitely. Moreover, it decouples the type of burst from the lock requests.

2.6 Reducing the number of variables modeling the environment

2.6.1 Substitutions

Earlier, we used `master_lockreq` in place of $HLOCK[HMASTER]$. The reason is that assumption A3 does not constrain $HLOCK$ any more (and was dropped, as explained earlier), so each bit in the array $HLOCK$ can take any value in its domain (no constraint). Therefore, it is not significant *which* bit this is, so $HMASTER$ can be removed, abstracting the value $HLOCK[HMASTER]$ by a single environment bit `master_lockreq`.

This implies the same replacement, of $HLOCK[HMASTER]$ by the bit `master_lockreq`, also in G2.

The only remaining use of array $HLOCK$ is in G7. Using arrays, the guarantee G7 can be expressed as

```
[] ( decide -> (lockmemo' <-> HLOCK[grant']) )
```

We observe that `HLOCK[grant']` above is again an element of the array $HLOCK$. Before replacement with bit `master_lockreq`, the only constraint on $HLOCK$ was A1. A1 required that $HLOCK[HMASTER]$ eventually become LOW, if locked access is granted.

In general, `grant'` will be a master different than the current (so as to serve all masters), therefore `HLOCK[grant']` takes values independently of $HLOCK[HMASTER]$. So we abstract `HLOCK[grant']` by a single environment bit `grantee_lockreq`. The bit `grantee_lockreq` represents the lock request of that master that is selected as the next grantee. The values of `grantee_lockreq` are not constrained in any way.

The initial condition assumption A4 implies that `master_lockreq` and `grantee_lockreq` must be false initially.

2.6.2 Removing array HLOCK

The replacements:

1. $HLOCK[HMASTER]$ by the single bit `master_lockreq`, and of
2. $HLOCK[HGRANT[i]']$ by the single bit `grantee_lockreq`

weaken the assumptions. Therefore, they are desirable.

After these replacements, the array $HLOCK$ does not appear in any guarantee or assumption (except the initial condition A4). Therefore, the array $HLOCK$ can now be removed.

In the best case ($N = 16$ masters), the replacement of the bit array $HLOCK$ by the 2 bits `master_lockreq` and `grantee_lockreq` resulting in a reduction by 16 - 2 = 14 bits. So it reduces by 28 (14 unprimed and 14 primed copies) the number of environment variables in the binary decision diagram (BDD). This reduction is significant, because environment variables are universally quantified, leading to an exponential increase in the number of possible next inputs. This reduction reduced synthesis time.

Instead of 32 possible next environment valuations for $HLOCK$, this replacement reduces them to 4, so a reduction of universal branching by a factor of 8.

2.7 Merging progress guarantees

The second significant change was the merging of the N weak fairness guarantees in G9:

$$\bigwedge_{i=0}^{N-1} \Box\Diamond(HBUSREQ[i] \rightarrow (HMASTER = i)) \tag{4}$$

into an equivalent Büchi automaton with a single progress state. This reduces the N inner fixed point computations in the GR(1) synthesis algorithm, to only a single fixed point computation. As a result, the number of masters N does not increase any more the number of progress goals that the system must satisfy.

The Büchi automaton used is

```
/* G9: weak fairness */
assert active proctype fairness(){
    int(0, N) count;
    do
    :: ! request[count] || (master == count);
        if
        :: (count < N) && (count' == count + 1)
        :: (count == N) && (count' == 0);
            progress: skip
        fi
    :: else
    od
}
```

This automaton is *equivalent* with the conjunction of weak fairness guarantees, because it

- "looks" at each one of them in turn,
- waits until that progress requirement is satisfied (`! request[count] || (master == count)`)
- then increments the counter `count` and waits for the next fairness guarantee to be satisfied,

until it reaches the last fairness guarantee. At this point it has completed a round of each fairness guarantee being satisfied, so it visits the progress state, and starts a new round.

The automaton is equivalent to the LTL property

$$\begin{aligned} \Box\Bigg(\Diamond\Big(\Big(HBUSREQ[0] \rightarrow (HMASTER = 0)\Big)\wedge \\ \bigcirc\bigcirc\Diamond\Big(\big(HBUSREQ[1] \rightarrow (HMASTER = 1)\big) \wedge \ldots\Big)\Big)\Bigg) \end{aligned} \tag{5}$$

This demonstrates the merits of using a language that allows directly writing sequential composition. The result allowed synthesizing arbiters for up to 16 masters, when the previously possible number was 12 (with the specification from [3]).

Note that this automaton does *not* constrain in any way the order in which the arbiter chooses masters to grant the bus. In other words, it does not restrict the master to using a particular priority for choosing masters.

Note that, after all these changes, the result is a fully parameterized program, where one need only change a single preprocessor definition (`#define N 15`), in order to define a different number of masters in the bus.

2.8 Changes to variables

Note that we use an integer variable to model the $HGRANT$ signal. This conforms to the standard, because the standard requires that exactly one $HGRANT_i$ bit be true at a time. So we reduce the number of BDD variables by a factor exponential in the number of masters. The synthesized integer values of $HGRANT_i$ can readily be mapped to patterns of 7 bits, where only a single one is true.

3 Experimental results

3.1 Experimental configurations

The experiments were performed with the open PROMELA compiler [10] and the SLUGS GR(1) synthesizer as back-end [11]. The hardware used has:

1. Intel(R) Xeon® X5550 processing core
2. runs Ubuntu 14.04.1
3. 11 GB RAM (3 experiments in the last batch were run after an upgrade to 27GB RAM).

The modifiable parameters are listed in the table below. The parameters that are fixed are assigned values in the table. The remaining parameters vary over the experiments. The number of variables varies by the size of the masters, and is given for the different specifications in Fig. 32 to Fig. 39.

In addition, there are two versions of the specification: the original from [5], and the revised one proposed here. The original specification is obtained by translating to PYTHON the PERL generator script for AMBA specifications that is distributed on the ANZU homepage. The new generator produces the same specifications, but in open PROMELA syntax (as `ltl` blocks for assumptions and assertions). For each specification, there are two equivalent variants that we evaluate: conjoining the fairness formulae, or merging them into a single Büchi automaton.

In the experiments, we are interested in the effect of the following parameters: strategy reordering, machine (memory, CPU frequency), conjunction vs BA, original vs revised AMBA specification, initial variable ordering.

3.1.1 Enabling reordering during strategy construction

In all experiments, reordering is enabled during realizability. Checking realizability without reordering causes a memory blowup very fast, going from 309K BDD nodes in the strategy for 2 masters, to 1.8M BDD nodes for 3 masters. For comparison, when dynamic reordering during realizability is enabled, in the worst case[1], the strategy BDD for 2 masters has about 20K nodes, and about 200K nodes for 3 masters.

In SLUGS, reordering during strategy construction is disabled by default. Enabling reordering during this second phase proved the enabling factor for scalable synthesis of the AMBA case study when the (weak) fairness guarantees are conjoined. In contrast, the revised specification with the fairness requirements represented as a single Büchi automaton does not require that reordering be enabled during construction of the strategy.

The price is longer runtimes. Nonetheless, when strategy reordering is deactivated, the runtimes using a BA do not scale as quickly as memory does when using a conjunction of fairness formulae. Without reordering during strategy construction, the shared BDD quickly blows up, for both the original and revised specifications, as shown in Fig. 31 and Fig. 19, respectively.

Note that in experiments with the same specification, the realizability phase has the same parameters, independently of whether the strategy is constructed with reordering enabled or not. Typically, realizability terminates, whereas construction of a strategy can cause a memory blowup

[1]The worst case in terms of strategy size blowup is the original spec with reordering disabled during strategy construction.

Table 1: Parameters of the experiments (both those constant and variable).

parameter	value
mem	10GB
cache	uint-1
min hit	1
blowup	1.2
reorder	group sifting
reorder during strategy construction	varies
number of variables	varies
universal branching (number of env vars)	varies
initial var order	natural order (1, 10, 2, …)

or time out, limiting the number of masters for which we could synthesize an arbiter, for certain combinations of specification with parameters.

3.1.2 Reordering using group sifting

Rudell's sifting [12] is the reordering algorithm used by Slugs. After experiments with different reordering algorithms, group sifting [13] was found as the algorithm that produces the best results for this case study. Group siting is an extension of sifting that automatically detects affinity between variables, and creates temporary groups. Grouped variables are shifted together during swaps of levels. This can improve the results by avoiding a situation where shifting one of the two tightly coupled variables finds no better place, because the other variable is "pulling it back" (known as rubber-band effect), and the same happening to the second variable results in no overall change. In contrast, moving both variables together enables finding a better position for both of them.

3.1.3 Number of variables

Original vs revised specification The number of variables differs between the original and revised specifications. It also differs for each variant of these specifications. In the original specification, the number of variables grows faster with the number of masters, than it does in the revised one. The reasons are:

- The use of one lock request bit for each master. The lock requests have been abstracted by using two bits, without any conflict with the requirements described in the AMBA technical manual [2]. This change leads to $N-2$ fewer environment variables, where N is the number of masters. It is a significant gain, because not only does it reduce the state space size, but also the degree of universal branching (at each state, the environment has fewer "next moves").
- The one-hot encoding of the system variables "master" and "grantee" in the original specification. In the revised specification, these variables are encoded as bitvectors. The two formulations are equivalent. As a result, in the revised specification, the number of system variables increases only when the number of masters reaches boundaries of powers of 2. The effect is reduction of state space size.

Table 2: Overview of results.

	Strategy reordering	Specification original	Specification revised
Conjunction of fairness	with	slow	fast
	w/o	memory blowup	memory blowup
Büchi automaton	with	very slow	ok
	w/o	slow	ok (slower)

Fig. 32 to Fig. 39 show how the number of variables scales for the original and revised specificantion variants considered.

Conjunction vs Büchi automaton The number of variable differs also between a specification that includes fairness in the form of conjoined recurrence formulae, and as a Büchi automaton. The representation of a Büchi automaton requires introducing an auxiliary bitvector for representing the current node of the automaton. This is also discussed later, in the section Section 3.4 analyzing the trade-off of using a BA, instead of handling weak fairness at the level of the strategy construction algorithm.

Note that when using a BA in the original specification, there is one extra fairness guarantee from one of the automata (Eq.G2.4 in [5]. This results in one extra point in the "combined strategy" plots. The number of points in a "combined strategy" plot is equal to the number of recurrence goals in the game, plus one (the initial point). This guarantee is not present in the revised specification, because it describes a safety property that can be represented by a process without progress states.

Optimizing away unused auxiliary variables Note that if the processes of a player do not include any atomic blocks, then no auxiliary variables are added for requesting and granting atomic execution of processes. In the specification of the AMBA arbiter used here, no atomic blocks are necessary, so the auxiliary variables ex_s and pm_s are not used, thus the compiler does not define them. This avoids unnecessarily increasing the number of states.

3.2 Instrumentation of the GR(1) synthesis algorithm

3.2.1 The different configurations

The experiments were performed for 8 different combinations: original and revised specification, using conjunction or a BA, and with strategy reordering enabled and disabled. An overview of the results is shown in Table 2, and in detail in Fig. 4a to Fig. 31c.

3.2.2 Measurements for each phase

The measurements were obtained by inserting in Slugs statements that print the information and dumping this output to a log file. The most relevant set of changes can be found in a fork of Slugs on `github`. There are three distinct phases of computation:

1. Fixed-point iteration that decides realizability and stores the interant sets X, Y, Z.

2. Construction of individual strategies, one for each recurrence goal.

3. Combination of the individual strategies into a single one that iterates through them.

Each phase involves different quantities to be measured, so it is instrumented slightly differently.

The realizability phase involves three nested iterations, each one computing a fixed point. The three variables are X, Y, Z, and at a high level, the iteration has the structure of the following μ-calculus formula $\nu Z.\mu Y.\nu X$. So X, Z are greatest fixed points and Y is a least fixed point. Moreover, X is in the innermost loop, so it is the most frequently updated variable, with variable Y less frequently updated, and variable Z the least frequently updated one.

Each one of the variables X, Y, Z represents a set symbolically, by reference to a BDD in the shared BDD managed by CUDD. Therefore, the size of these variables can be quantified as the number of BDD nodes in the referenced BDD. This number is obtained by calling the function `Cudd_DagSize`.

In addition, the total number of nodes in the shared BDD is recorded with `Cudd_ReadNodeCount`. The total size of the shared BDD corresponds to the current memory use of CUDD (though they are not identical entities). Reordering is triggered based on growth of the shared BDD, as measured by its number of nodes. The total number of nodes also quantifies the randomness with which new nodes have been added [14], and so how inefficiently representable the intermediate results of the fixed point computation are, compared to the final result (that is typically much smaller, as has been observed in the literature [15]).

Plotting the sizes of the variables X, Y, Z provides a view into the fixed-point iteration. However, it is difficult or impossible to tell when a loop starts of ends, by only inspecting how X, Y and Z change. For this reason, the indices of the:

1. current recurrence assumption, and

2. current recurrence guarantee

are also recorded. Finally, the total runtime is measured with the function `gettimeofday` from `sys/time.h`, and the time spent reordering so far is measured with the function `Cudd_ReadReorderingTime`.

The remaining measurements are taken during construction of the strategy. During construction of the individual strategies, the quantities recorded are the total time, reordering time so far, goal pursued by the strategy under construction, the total number of nodes in the shared BDD, the number of nodes in the BDD of the current strategy, as well as the number of BDD nodes in the sets of new and accumulated states.

The final phase combines the individual strategies into a single one. Besides the total and reordering time, the goal whose strategy is currently combined in the overall strategy, the total BDD nodes, and the nodes in the combined strategy (so far) are recorded.

3.3 Observations

3.3.1 General

The figures can be understood as follows. There are two sets of figures:

1. the top four figures extend through the whole computation, whereas

2. the bottom three figures contain more details for each of the individual phases.

In most runs, reordering takes a high, to very high, percentage of the total runtime. A sense of this ratio can be obtained visually by comparing the bisector in[2] Fig. 4a-(i) (dotted) to the cumulative reordering time (solid line). The final values of each run are collected in Fig. 32 to Fig. 39.

The periods of constancy of the reordering curve correspond to BDD computations. The highlighted period corresponds to the construction of individual strategies.

Fig. 4a-(ii) shows the goal that is currently pursued during realizability, and the goal whose strategy is currently being constructed during the individual strategy construction phase. For the revised specification, there is only a single recurrence goal, so this curve is constant. For the original specification, the successive goals appear as increments.

Fig. 4a-(iii) shows the recurrence assumption that is currently being considered during the fixed-point computation of Y. For every iteration of the middle fixed-point Y, the computation takes a disjunction over all assumption, which results in fast cycling in the plot.

The total number of nodes in the shared BDD is shown in Fig. 4a-(iv). We can identify the following features that correspond to the computation. Initially, the total number of nodes grows slowly, and several reorderings occur (flat or slightly decreasing periods). These are triggered by the initial exploration of the state space. The new states added to the BDD trigger changes to the variable ordering, which tends to "adapt" to the sets that need to be represented.

After the first outer fixed point is completed, the stored iterants (X, Y over the inner iterations, to be used for constructing the strategies in the final fixed-point iteration) are deleted, if a fixed-point was not reached in the current iteration. This dereferencing appears as abrupt reductions of the number of nodes to almost zero (forming "teeth"). So each such reduction corresponds to one iteration of the Z greatest fixed-point.

During construction of the strategy, the number of nodes increases monotonically, if no reordering is enabled. If reordering is enabled in the strategy construction phase, then the number of nodes may be observed to decrease, and in those cases, its rate of growth is always limited (with the trade-off of long interruptions for reordering). Note that the allocated memory never decreases, because CUDD does not release memory, even when nodes are deleted.

In some total node count plots, a dashed line is visible, starting from the end of the highlighted period. This corresponds to the time between the completion of individual strategy construction, and the first iteration of combining strategies. It is included, because reordering during that period can defer the initial iteration of constructing the combined strategy.

In the second set of figures, Fig. 4a-(v) shows the sizes of the fixed-point variables X, Y, Z. We can observe that X, Y are quite similar in size. This is expected, because Y is assigned the union of X over the recurrence assumptions (and there are only two recurrence assumptions). The variable Z exhibits more interesting behavior. In the initial iteration, Z is always 1 ($\top$), by definition. Its size changes in the following iterations, remaining constant in each one of those iterations. As a result, the curve of variable Z indicates the successive loops of the outer fixed-point iteration. We can also observe "humps" of X, Y during the intermediate results of each outer iteration. Note that these humps of X, Y, and steps of Z correspond to the teeth in Fig. 4a-(vi).

Fig. 4a-(vi) shows the BDD size of the strategies for the individual goals, as they are being constructed. It also shows the number of accumulated and new states. It is interesting that these curves are typically close to constant. Another observation is that, in most cases, this phase is brief, unless reordering is triggered. When strategy reordering is disabled, this phase is always of short duration.

[2]Counting over the graphics within each subfigure proceeds from top to bottom.

Finally, the size of the cumulative strategy as the individual ones are being combined is shown in Fig. 4a-(vii). Note that in the revised specification, when using a BA, there is only one recurrence goal, so only a single point in this plot (there is only a single sub-strategy, so not multiple ones to be combined). In cases without reordering and with conjunction, the rapid growth of the combined strategy can be observed in this plot. Another observed feature is the triggering of reordering during this last phase, and that if triggered, reordering takes most of the time in this phase.

Fig. 32 to Fig. 39 summarize the previous measurements over the number of masters, per experimental configuration. The top plot shows how the numbers of variables change. The number of variables determines the size of the state space, and this is a difference between specification variants. The number of environment variables gives an upper bound on the amount of universal branching in a problem.

The second plot summarizes the total number of nodes in the shared BDD, and the size of the synthesized strategies. This is a measure of how larger the intermediate BDDs are from the desired result.

The third and fourth plots show how total, realizability, and reordering times scale with the number of masters, and how they compare to each other.

Fig. 40 to Fig. 50 are the most interesting, and compactly represent the conclusions from the all the experiments. The show ratios of quantities between different selected pairs of configurations, those that we think are the most interesting and useful.

3.3.2 Comparison of variants

From the measurements, we observe that

- The revised specification is orders of magnitude better than the original, as seen in Figs. 49 and 50.
- Using a Büchi automaton, reordering during strategy construction is not needed, and undesired, as seen in Figs. 40 and 41.
- Using conjunction of fairness goals, reordering is necessary, as seen in Figs. 42 and 43.
- Conjunction with reordering is clearly better than the BA, by about an order of magnitude, as seen in Figs. 44 to 46.
- Using a Büchi automaton is clearly better than conjunction without reordering, by several orders of magnitude, as seen in Figs. 47 and 48. In absence of strategy reordering, only the BA scales.

3.4 Trade-off of counter in state space

In the following, by *realizability phase*, we refer to the fixed-point computation that stratifies the set of states to produce the layers that comprise the attractors [9, 16]. This phase stores these sets in memory as an array of BDD nodes. By *construction phase*, we refer to the construction of a strategy that satisfies the original specification, using the layers produced by the realizability phase. So the realizability phase precedes the construction phase.

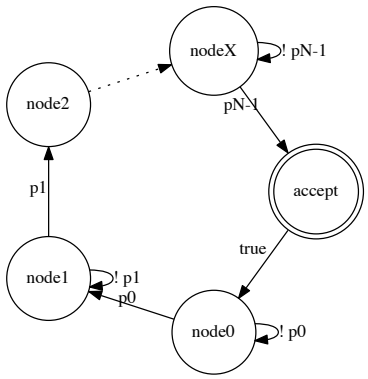


Figure 1: The form of a single Büchi automaton with one accepting state that is equivalent to N weak fairness goals.

3.4.1 The two alternatives

In this section, we evaluate the two (equivalent) alternatives for representing liveness goals:

1. as a conjunction of the form $\square\diamondsuit p_0 \wedge \square\diamondsuit p_1 \wedge \cdots \square\diamondsuit p_{N-1}$
2. as a single Büchi automaton with the structure shown in Fig. 1.

Representing the Büchi automaton augments the system state by the integer counter that describes the current node. This counter is linear in the number of fairness guarantees.

Conjoining multiple fairness goals leads to the construction of a sub-trategy for each goal. In order to obtain the overall strategy, these sub-strategies are combined, using a counter to keep track of the currently active sub-strategy. This other counter comprises the transducer's memory.

Note that both counters have the same range. In the final strategy, both counters appear as BDD variables. The difference is that the first counter is present during the attractor computations (realizability phase), whereas the second is introduced only at the end (strategy construction phase).

This means that using a Büchi automaton shifts the transducer memory from the construction, to the realizability phase. As a result, the state space of the game is multiplied by a factor of k, the number of fairness goals. In the experiments presented here, the default implementation of the synthesis algorithm in Slugs was used. In the worst case, the time complexity increases by a factor $O(k^3)$, measured in number of symbolic controllable preimage computations. An improved implementation employing fixpoint memoization [17, 5] is available as a Slugs option, but was not used. The improved implementation would be affected by a factor of $O(k^2)$.

Nonetheless, an improvement is observed, because synthesis remains scalable, even without reordering during the construction phase. This can be understood by considering three effects.

Including the counter in the state space encodes the problem symbolically, instead of explicitly. It avoids the combination of individual strategies later, which is an enumerated procedure that iterates over the transducer's memory (the counter introduced during the construction phase). The number of sub-strategies to disjoin increases linearly with the number of liveness goals. This can

affect the suitability of the variable order significantly, and it does. The measurements indicate an exponential growth of the total number of nodes in the shared BDD, as the number of masters grows.

The second effect is the reduction of intermediate results [15] of BDD computations. As has been observed in the literature [14], unstructured breadth-first search of the state space creates intermediate sets that have "a lot of ad hoc detail". For a fixed variable ordering, this semi-random detail decreases the likelihood of these intermediate sets being efficiently representable as BDDs. As described in [14], an interesting analog is the entropy of random strings in information theory. The Büchi automaton introduces structure in the state space that is relevant to the fairness constraints.

The third effect is the abruptness of changes to how far from optimal the current order is. The experimental results suggest that disjoining individual strategies at the end is disruptive to the variable ordering, necessitating reordering to avoid BDD blowup. In contrast, shifting the counter to the state space allows the order to adapt over a longer number of iterations, while the BDD is changing less abruptly (i.e., during the attractor computations).

Another effect is the reduction of size of the goal set. This can lead to a simpler BDD for the Z variable in the fixed-point computation. It is observed as small BDD size for Z for a specification that uses a Büchi automaton. For example, in Fig. 4a to Fig. 12a Z is much smaller than X and Y. In contrast, a conjunction of recurrence formulae leads to large Z values, much larger in many cases than the variables X, Y. This difference can be seen in Fig. 13a to Fig. 20b.

3.4.2 Effect of reordering and memoization

In most cases, 65%„ or more, of the total runtime is spent reordering the shared BDD. A similar observation has been made in [5]. Therefore, reordering is a controling factor of overall runtime. Reordering by sifting considers all BDDs in a "neighborhood" of the BDD being reordered. This allows it to cross boundaries between basins of attraction associated with different local minima.

However, it also makes sifting (and reordering in general) quite sensitive to the details of intermediate results during the fixed-point computation. In turn, these intermediate results can depend on the order that individual goals are visited, as well as variations in the encoding, e.g., of a program graph by integers. Overall, these variations can cause reordering to "touch" expensive neighborhoods, resulting in significant runtime outliers being observed. The sensitivity of BDD computations is a common observation in the symbolic model checking literature [15].

This effect can be further amplified by the cache, where results are memoized (mapping arguments to results of the operation being performed). As more memory is required, results are overwritten, causing regeneration, thus deviating from the practically bilinear cost of BDD operations, to the worst-case exponential [18], Programmer's Guide.

Configuring Cudd memory limit Initially, fragile behavior was observed with CUDD. This was due to the maximum memory limit of CUDD that by default[3] is set to 3GB in SLUGS. Increasing this limit to that available on the machine resulted in improved performance, and very significantly reduced fragile behavior. This limit can be set with the function `Cudd_SetMaxCacheHard`. A relevant suggestion can be found in the CUDD Programmer's manual, Sec. "Modifiable parameters".

Effect of initial order Another factor that introduced significant variability and adversely affected runtime was the initial order of variables. In the initial implementation, the initial order

[3] In `BFCuddManager.h` as of `f9eed21813e7e6d23c0bd63563b587cc1cee95c6` (line 30).

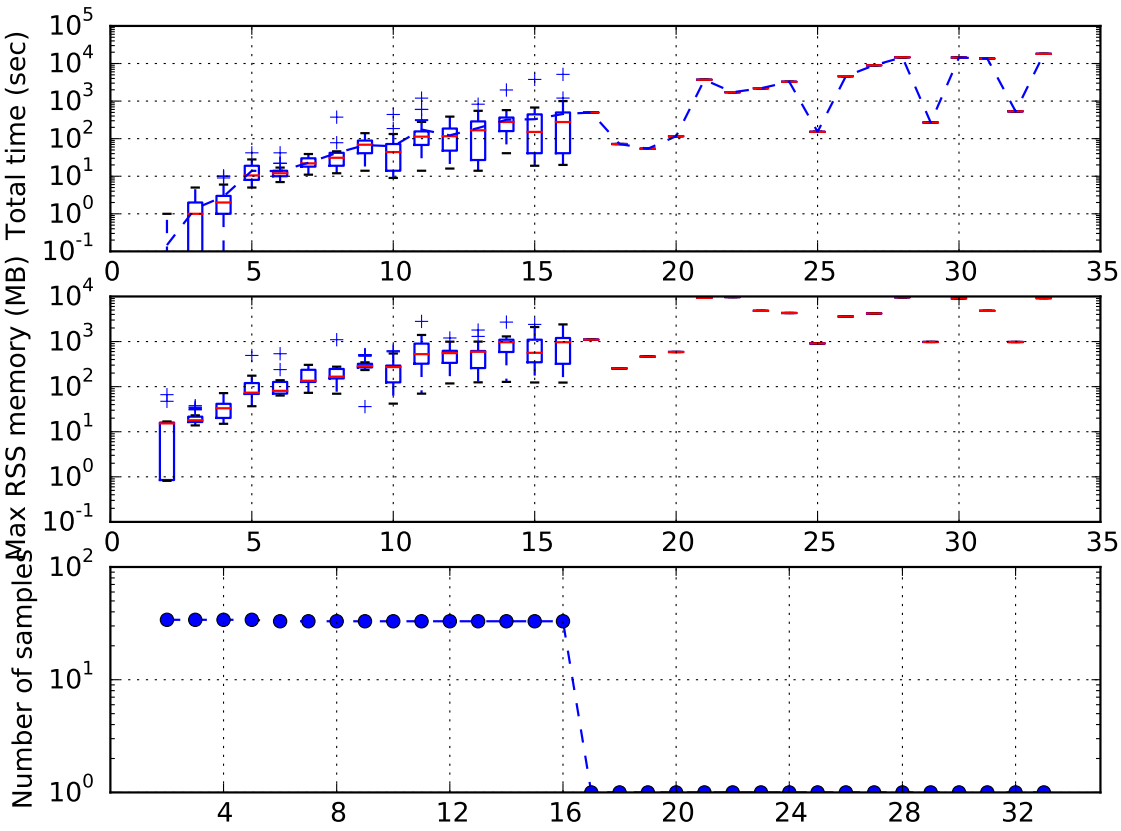


Figure 2: Total run time to synthesize an arbiter using the revised specification, with fairness as a Büchi automaton, and reordering enabled during strategy construction. These repeated experiments are different runs than those from which the detailed tracing plots were generated.

of variables was not controlled. The result was semi-random, because the enumeration of items in PYTHON sets and dictionaries can vary arbitrarily.

Later, the BDD variables were initially sorted in natural order by their bitblasted "flat" name (i.e., after translation to logic and bitblasting). The flat name of a variable includes scope information in the form of a prefix. This brings together in ordering variables that are defined in the same namespace, bits in the same bitvector, and variables with similar names (with respect to natural ordering). Initial sorting by natural order improved the results.

Using a natural lexical ordering to obtain the initial variable order brings together variables in the same scope. The revised specification contains more information in the form of variable scopes, because local variables in the same process have the same prefix in the logic formulae (for example, a local variable `x` will become `pid0_x`). The original specification is flat (no processes in the syntactic description), so it contains less scoping information.

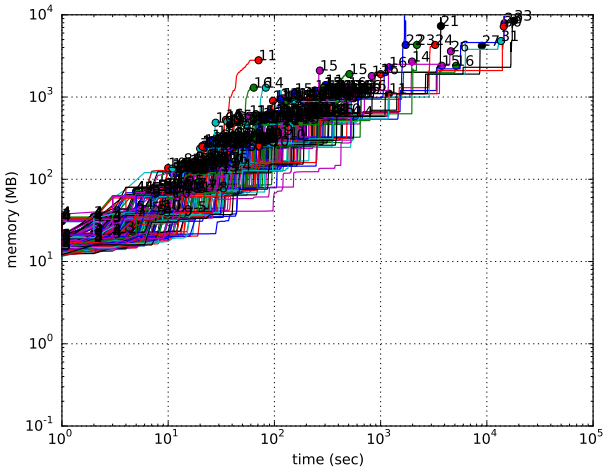
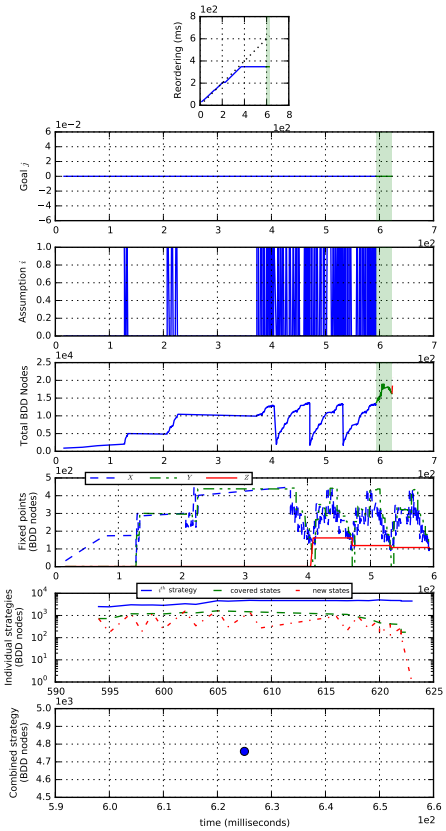
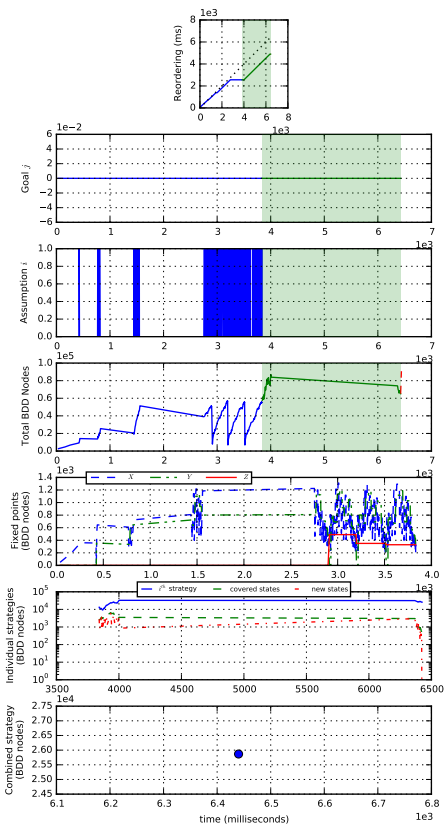


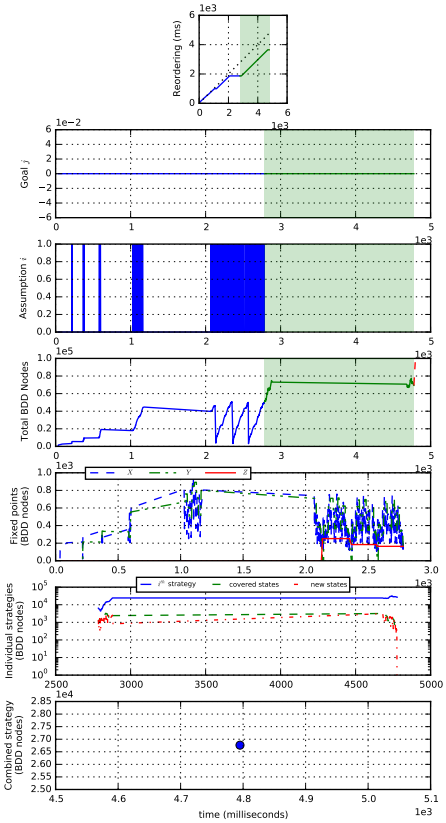
Figure 3: Peak memory consumption for the experiments of Fig. 2



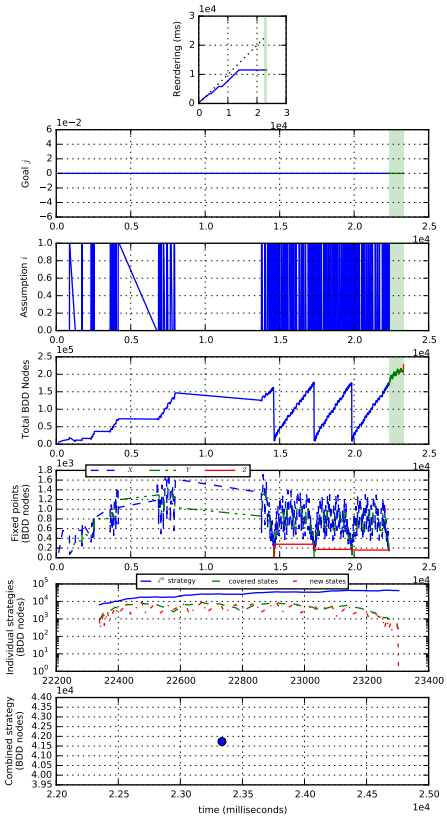
(a) 2 masters



(b) 3 masters

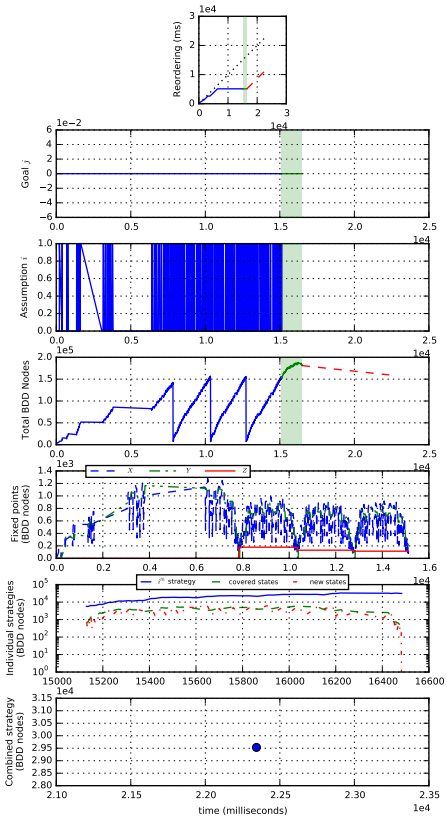


(c) 4 masters

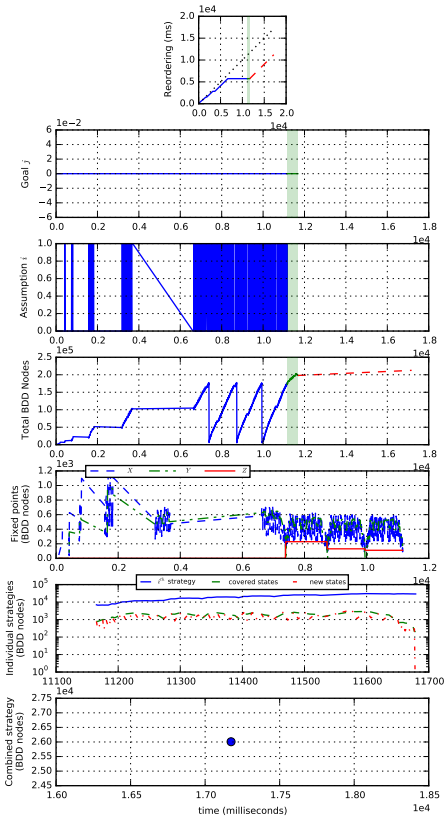


(d) 5 masters

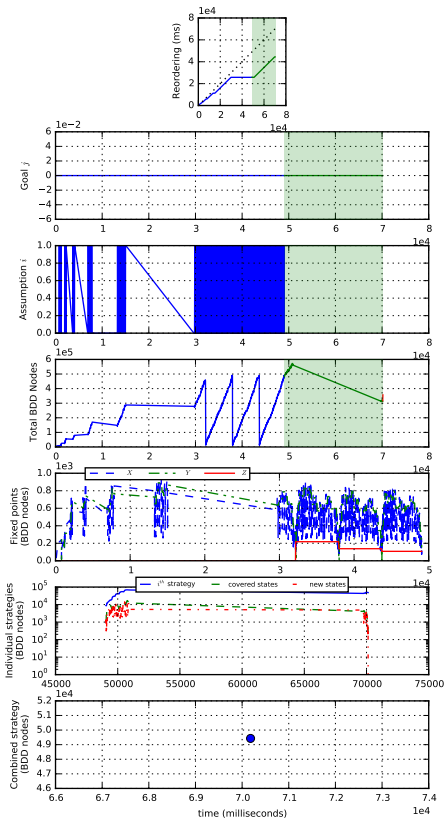
Figure 4: Revised spec with BA and strategy reordering.



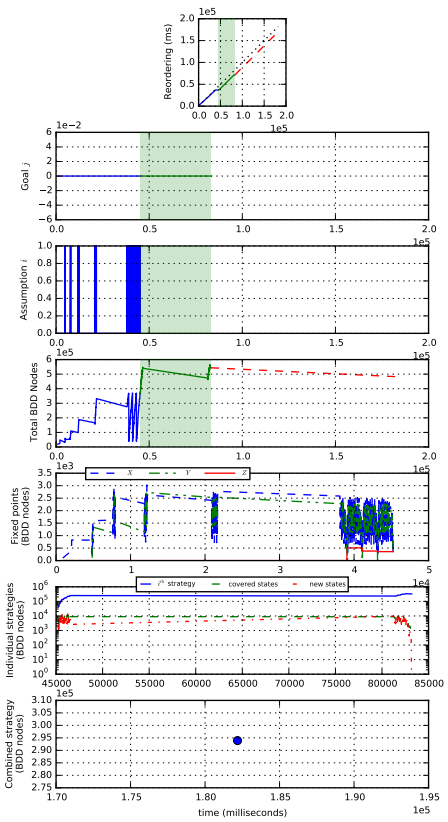
(a) 6 masters



(b) 7 masters

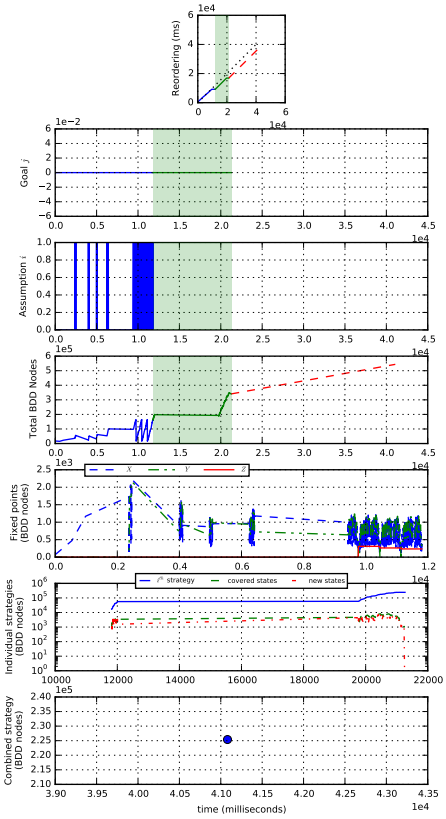


(c) 8 masters

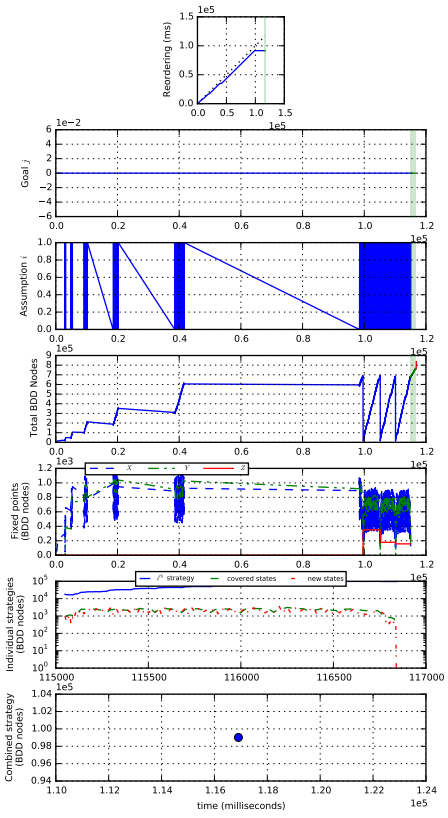


(d) 9 masters

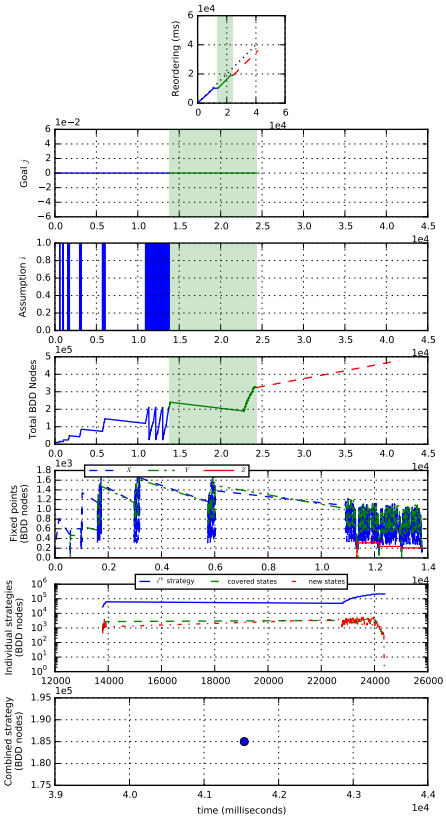
Figure 5: Revised spec with BA and strategy reordering.



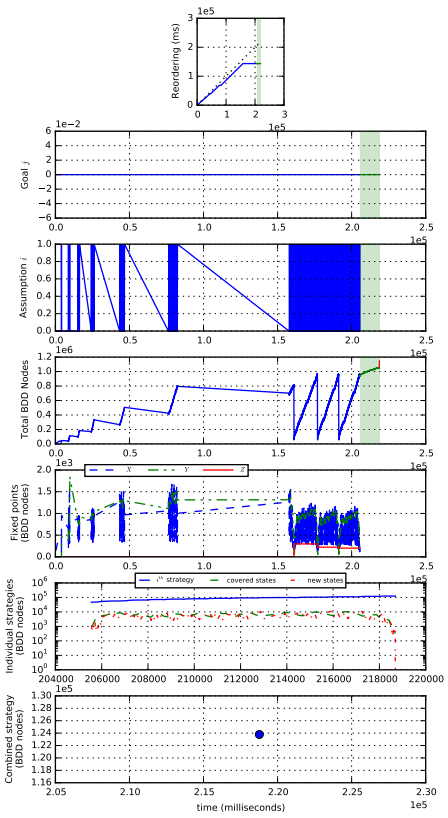
(a) 10 masters



(b) 11 masters

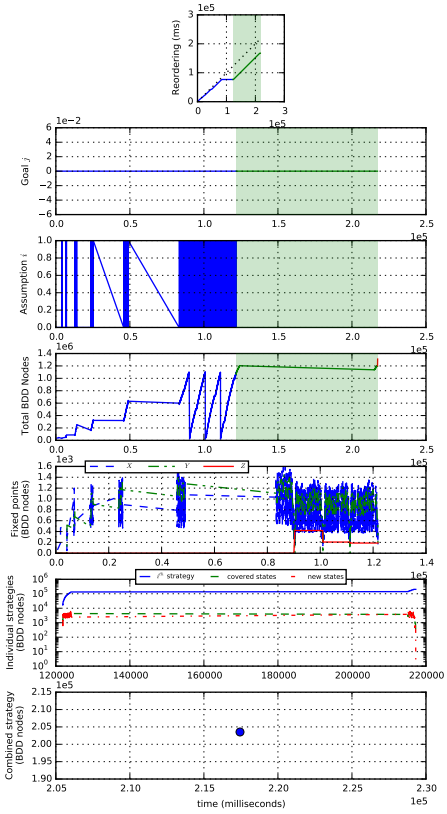


(c) 12 masters

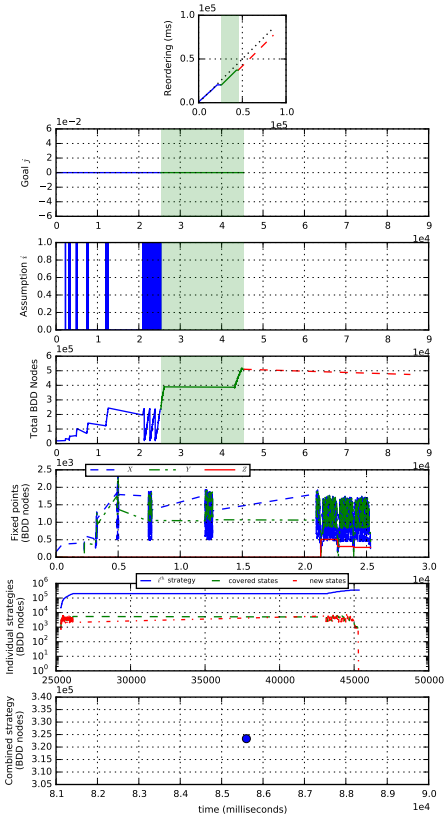


(d) 13 masters

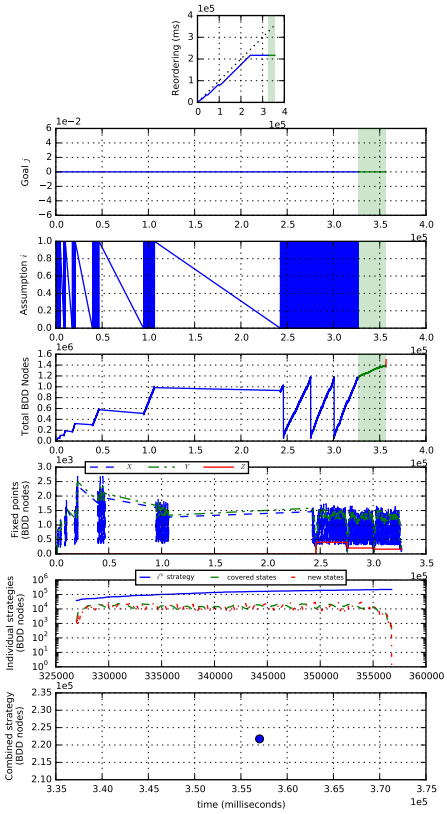
Figure 6: Revised spec with BA and strategy reordering.



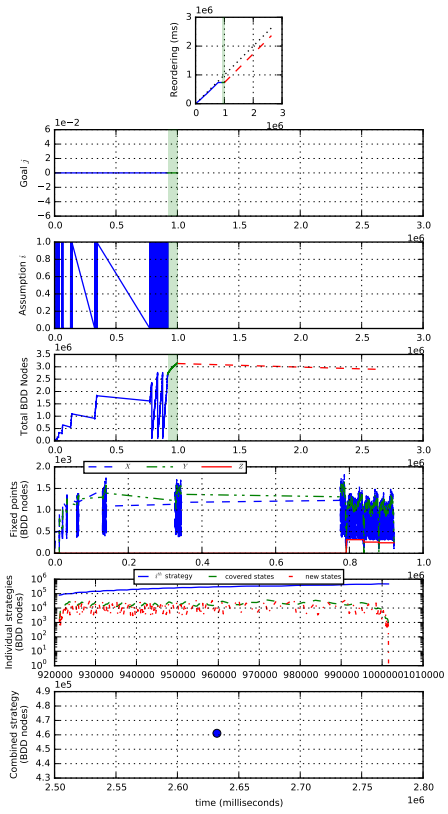
(a) 14 masters



(b) 15 masters

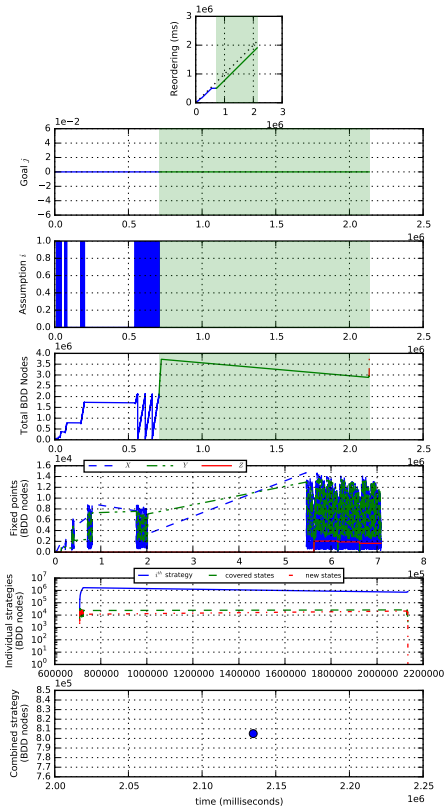


(c) 16 masters

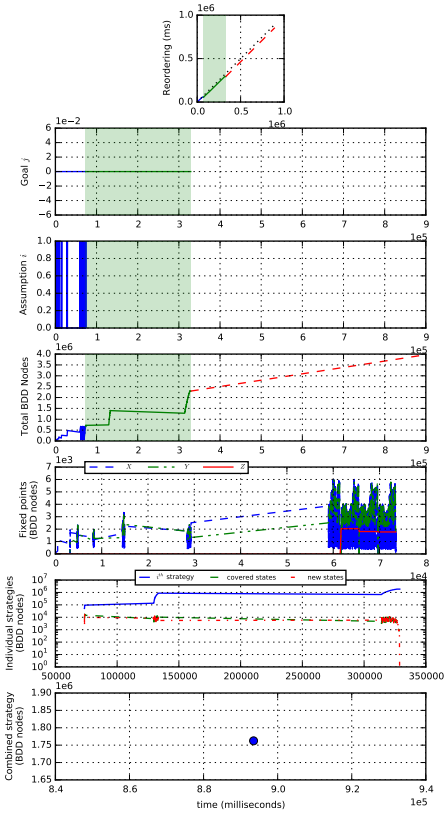


(d) 17 masters

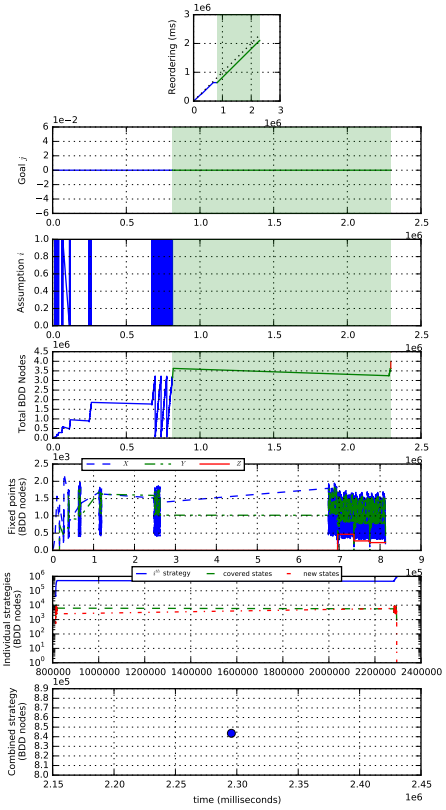
Figure 7: Revised spec with BA and strategy reordering.



(a) 18 masters

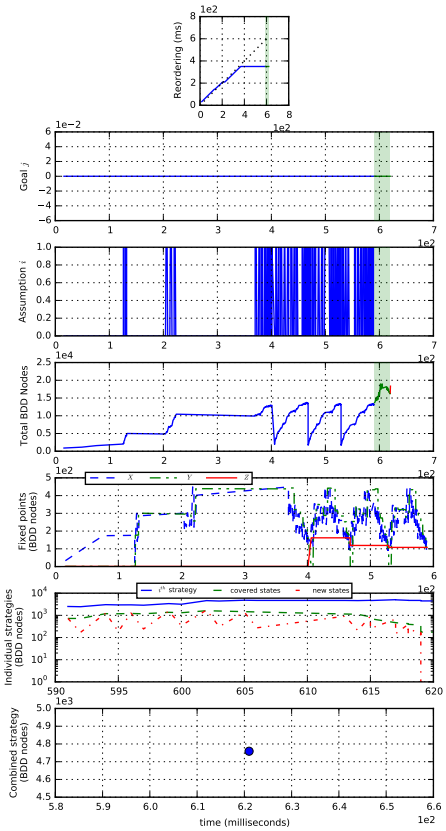


(b) 19 masters

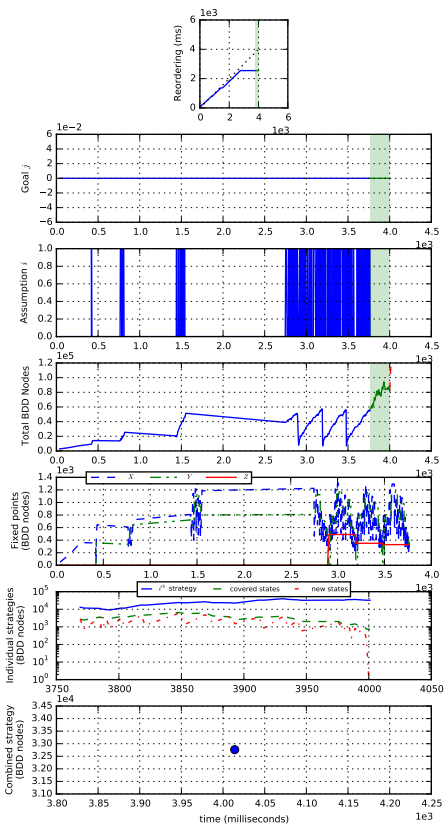


(c) 20 masters

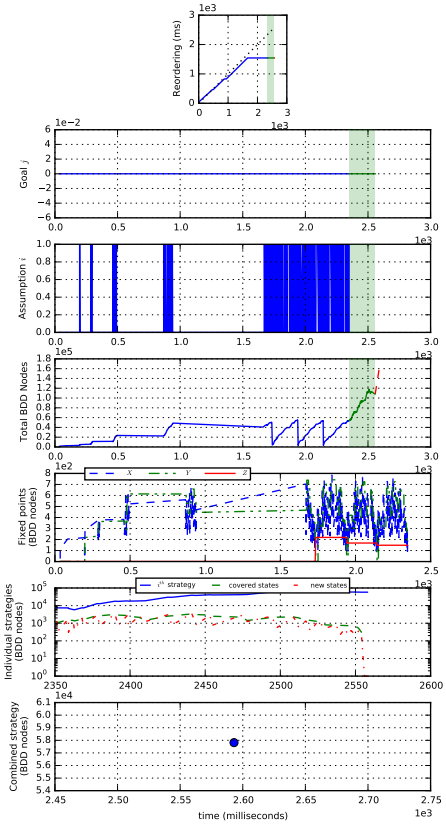
Figure 8: Revised spec with BA and strategy reordering.



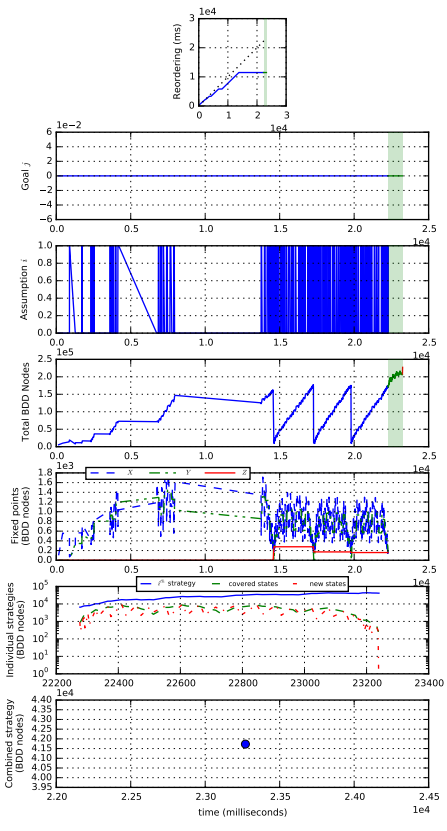
(a) 2 masters



(b) 3 masters

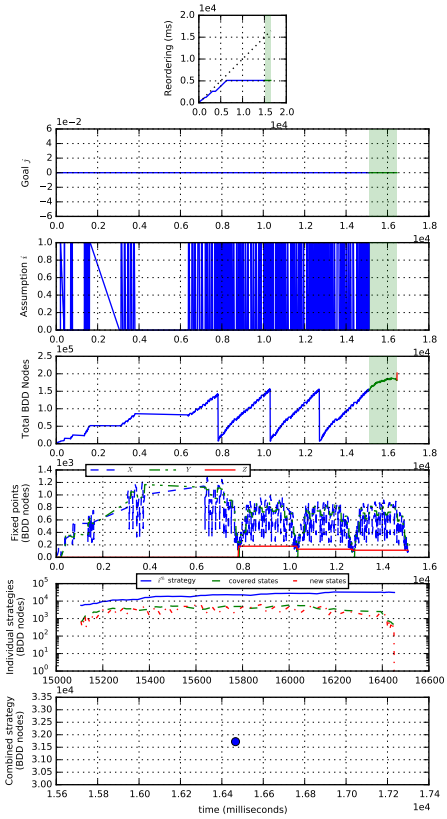


(c) 4 masters

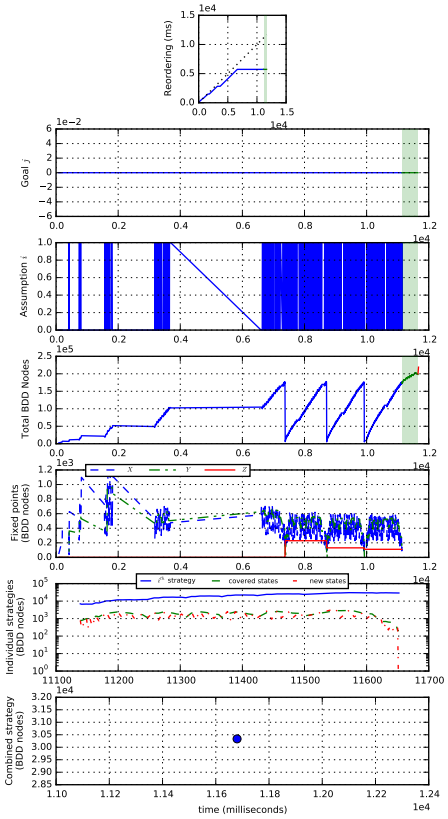


(d) 5 masters

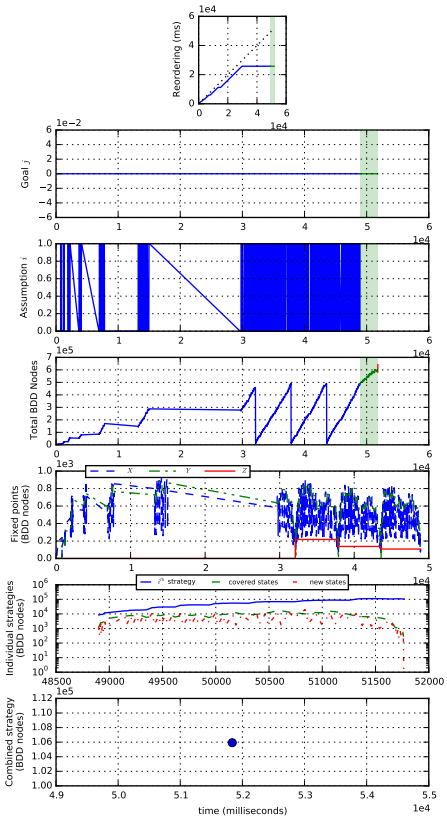
Figure 9: Revised spec with BA but no strategy reordering.



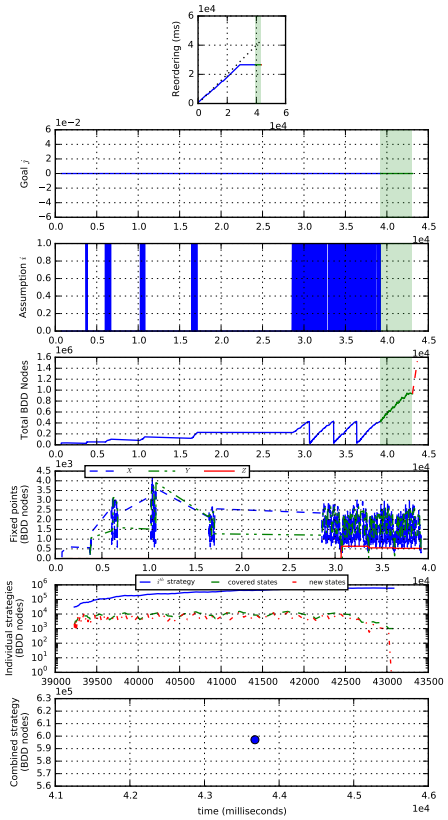
(a) 6 masters



(b) 7 masters

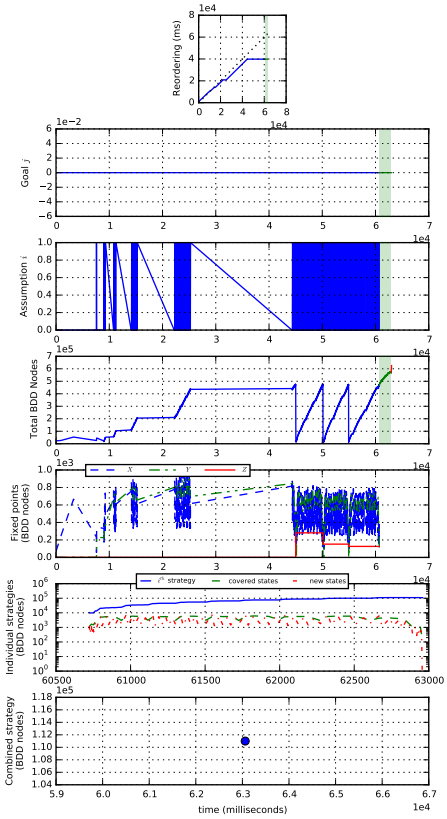


(c) 8 masters

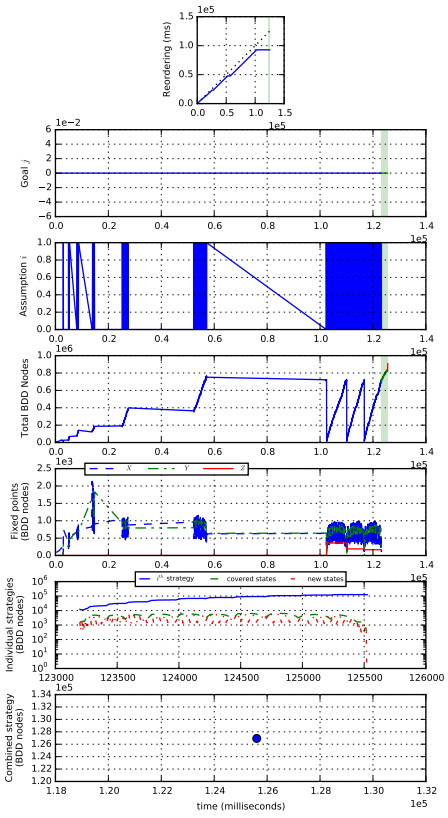


(d) 9 masters

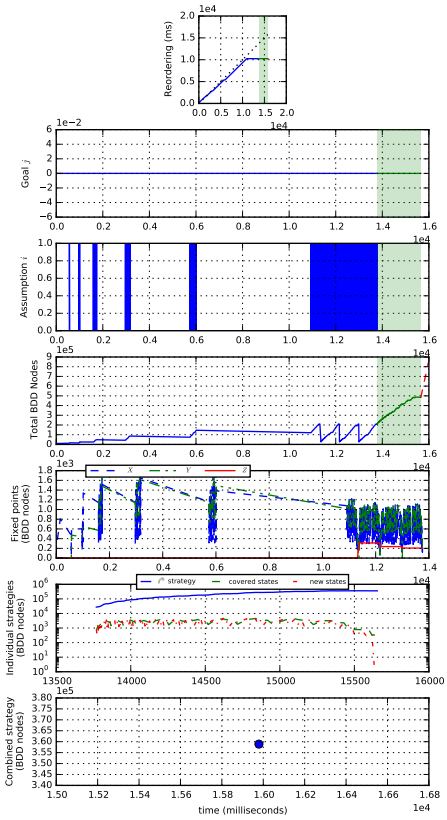
Figure 10: Revised spec with BA but no strategy reordering.



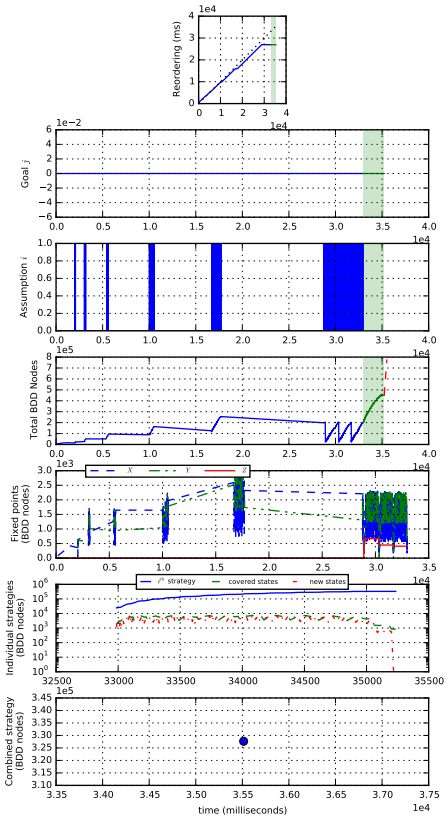
(a) 10 masters



(b) 11 masters

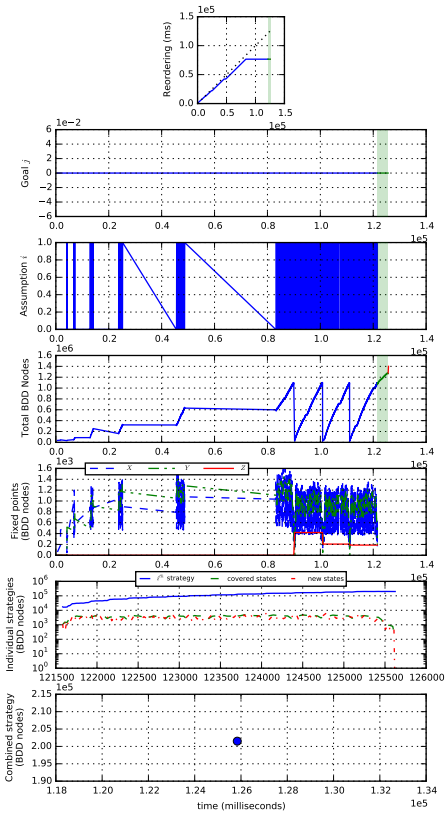


(c) 12 masters

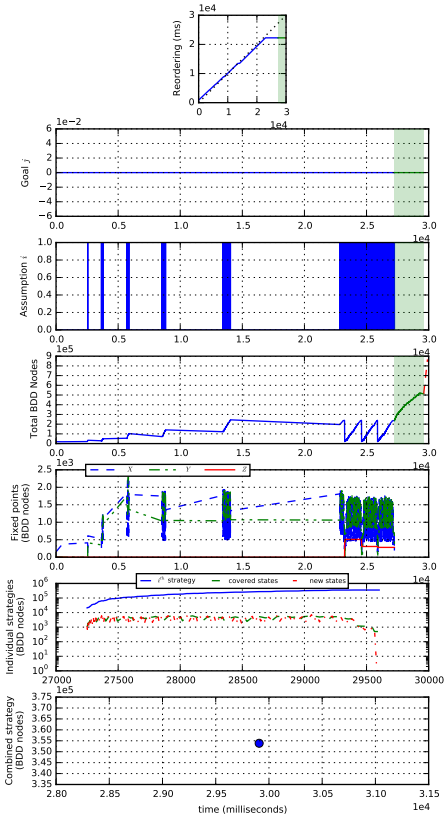


(d) 13 masters

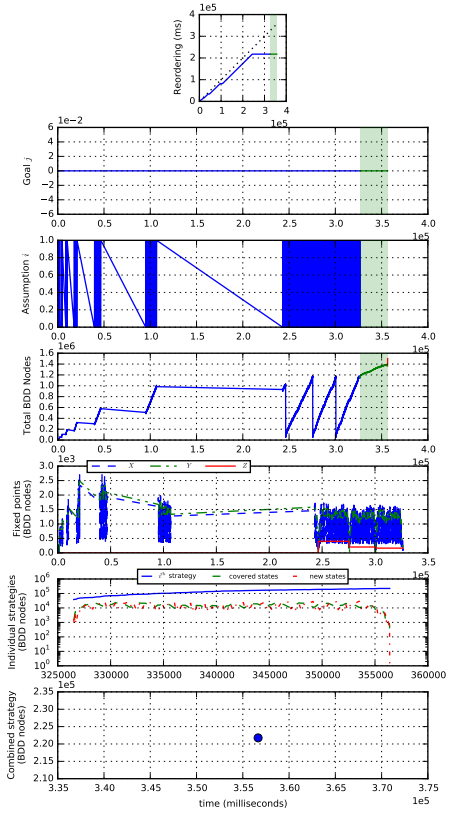
Figure 11: Revised spec with BA but no strategy reordering.



(a) 14 masters

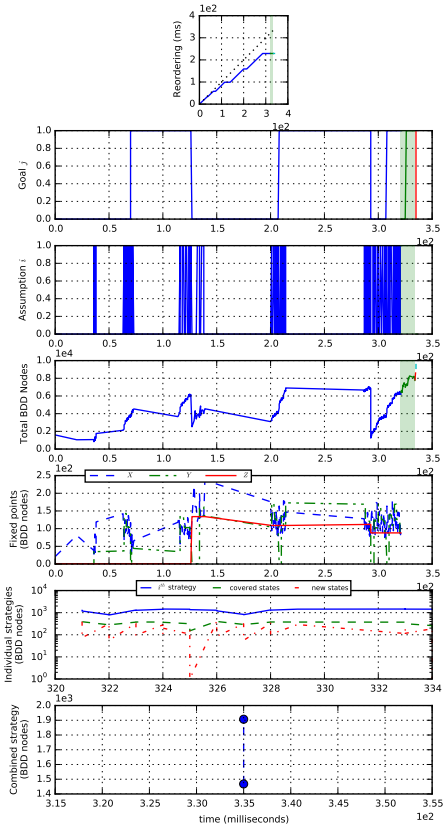


(b) 15 masters

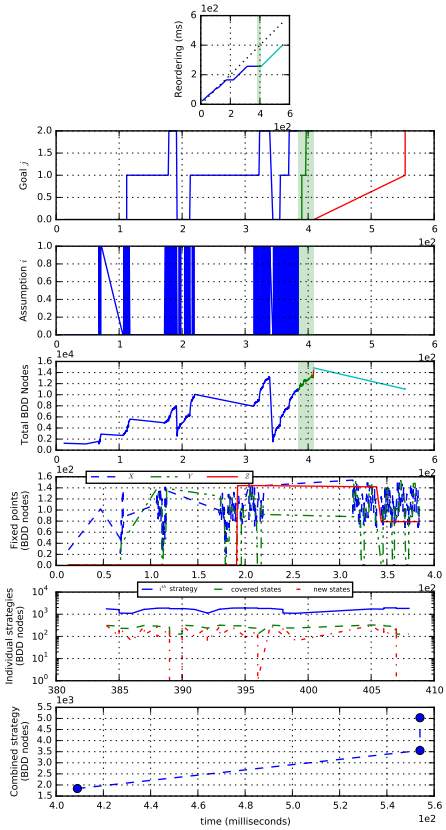


(c) 16 masters

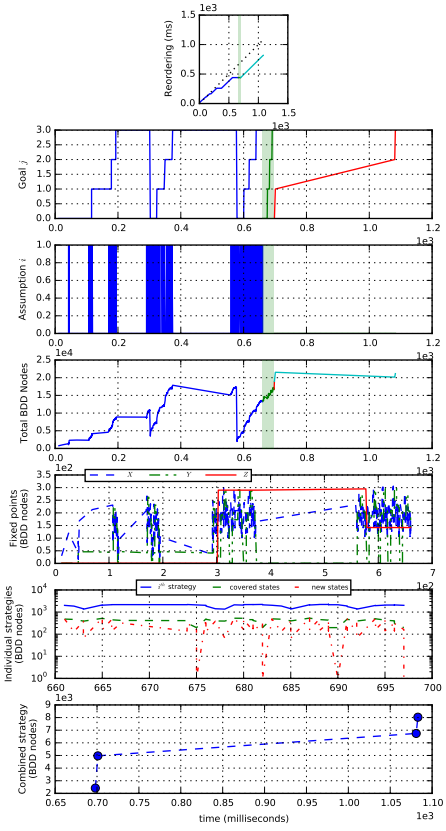
Figure 12: Revised spec with BA but no strategy reordering.



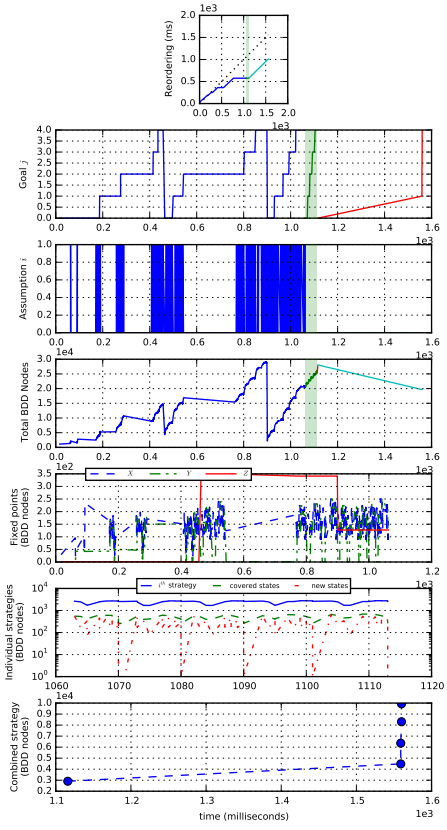
(a) 2 masters



(b) 3 masters

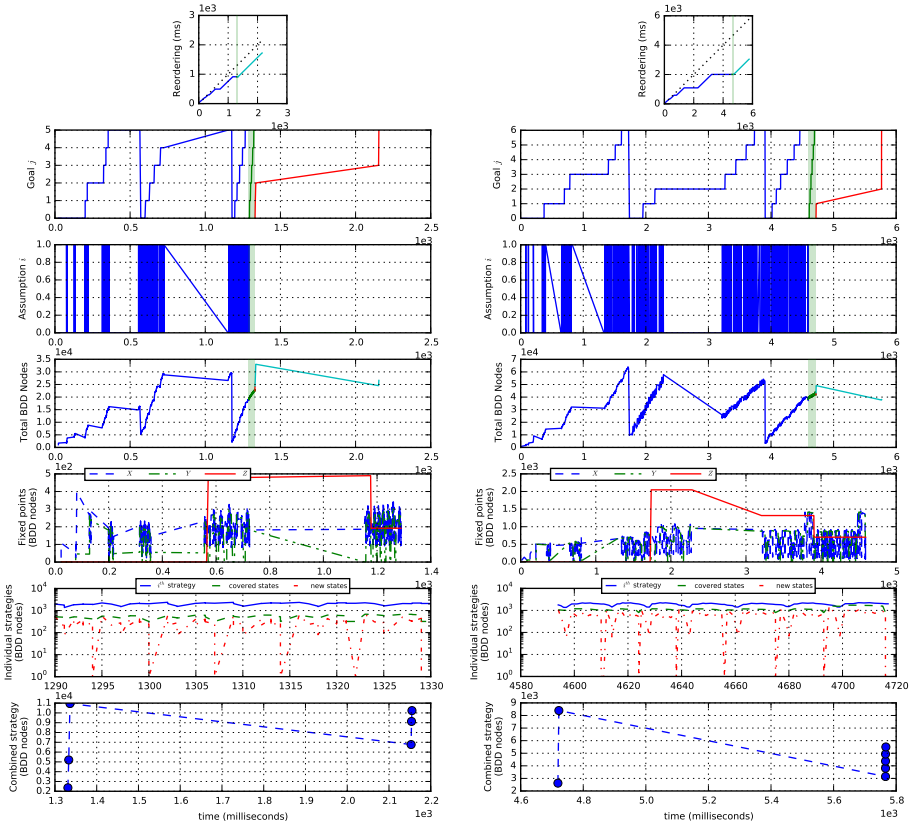


(c) 4 masters



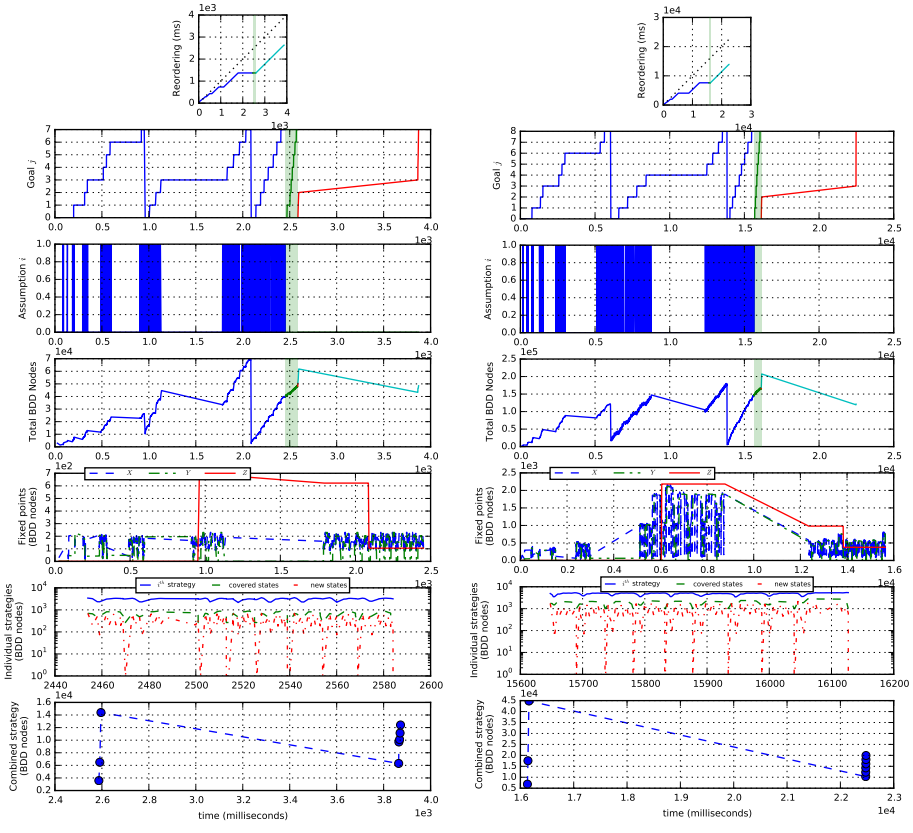
(d) 5 masters

Figure 13: Revised spec with conjunction and strategy reordering.



(a) 6 masters

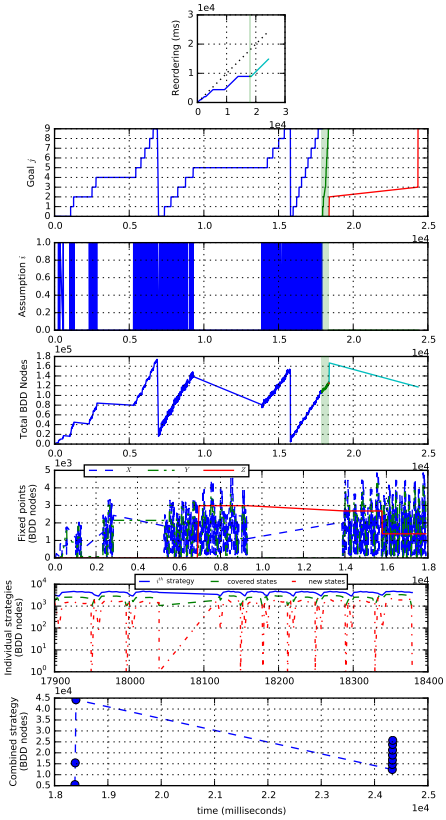
(b) 7 masters



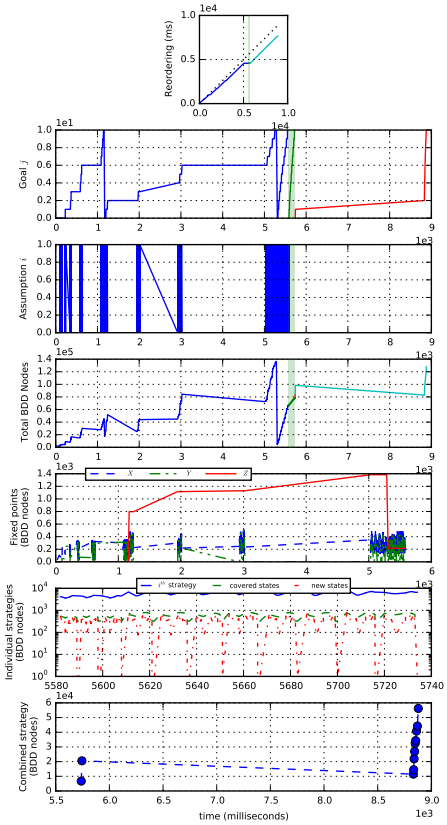
(c) 8 masters

(d) 9 masters

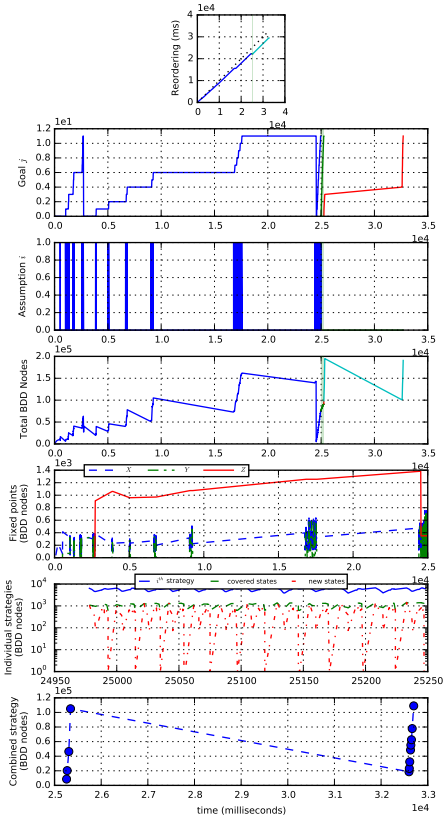
Figure 14: Revised spec with conjunction and strategy reordering.



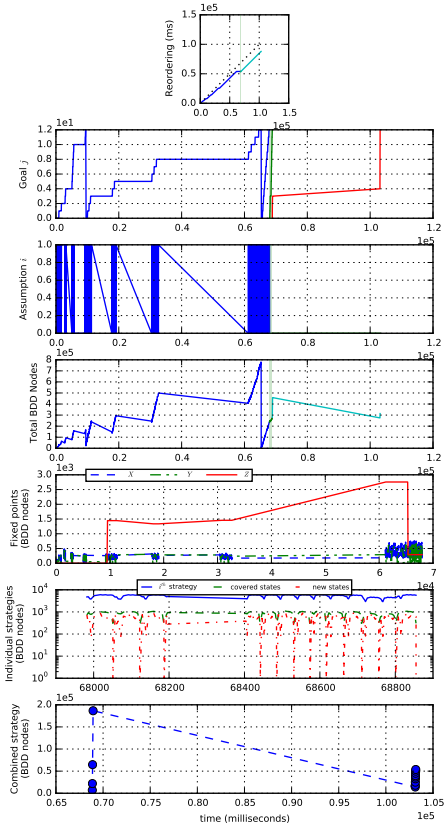
(a) 10 masters



(b) 11 masters

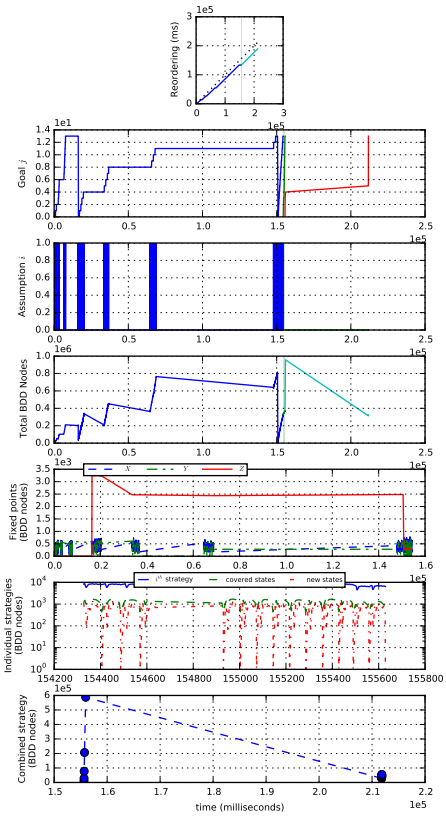


(c) 12 masters

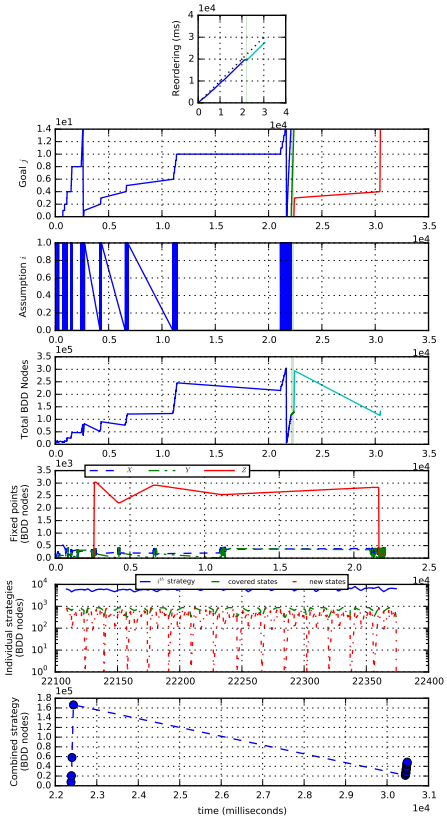


(d) 13 masters

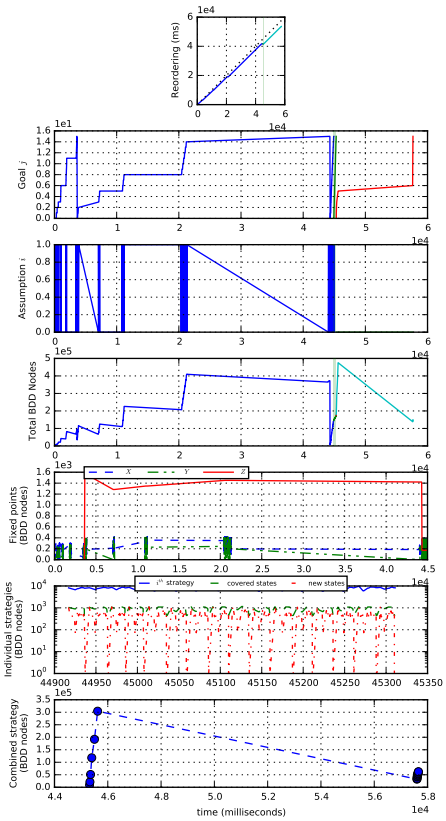
Figure 15: Revised spec with conjunction and strategy reordering.



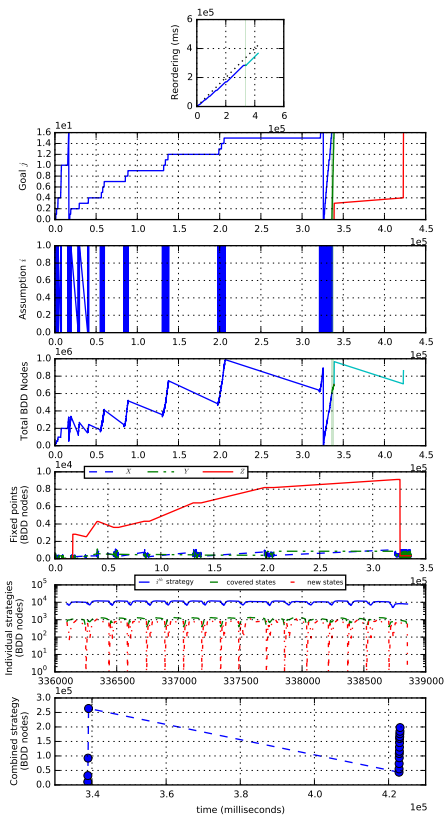
(a) 14 masters



(b) 15 masters

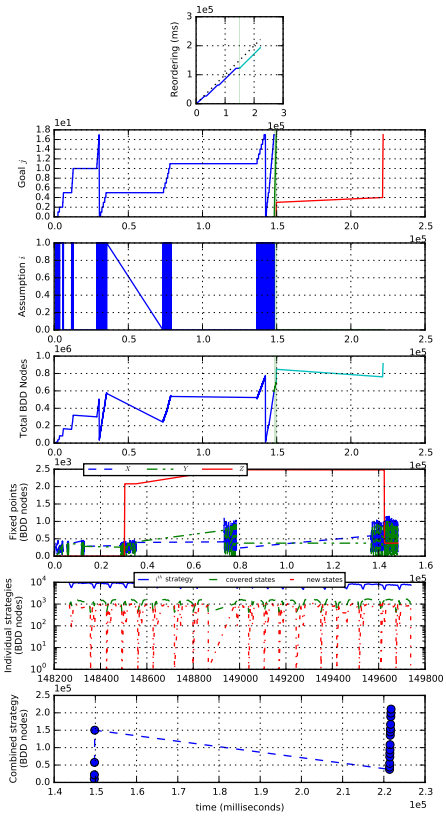


(c) 16 masters

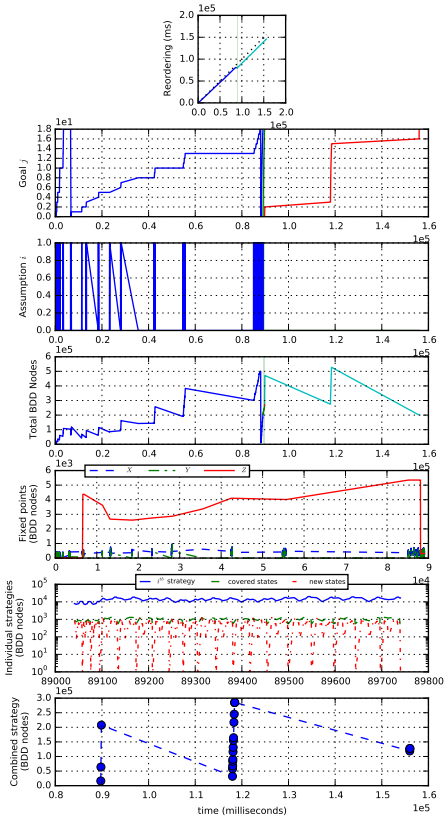


(d) 17 masters

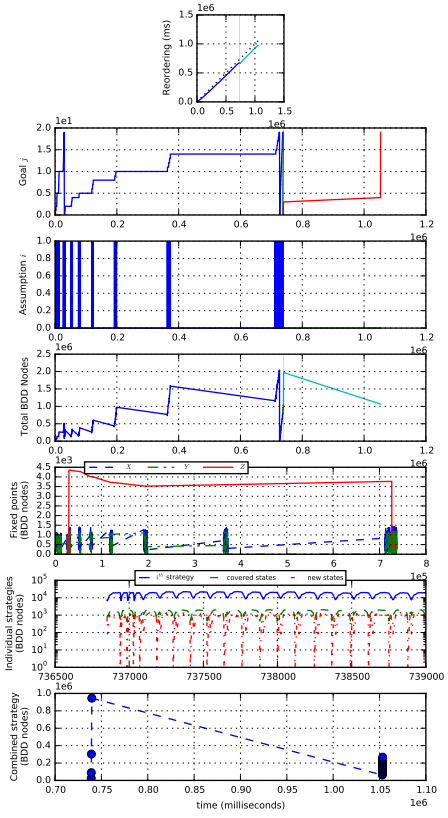
Figure 16: Revised spec with conjunction and strategy reordering.



(a) 18 masters

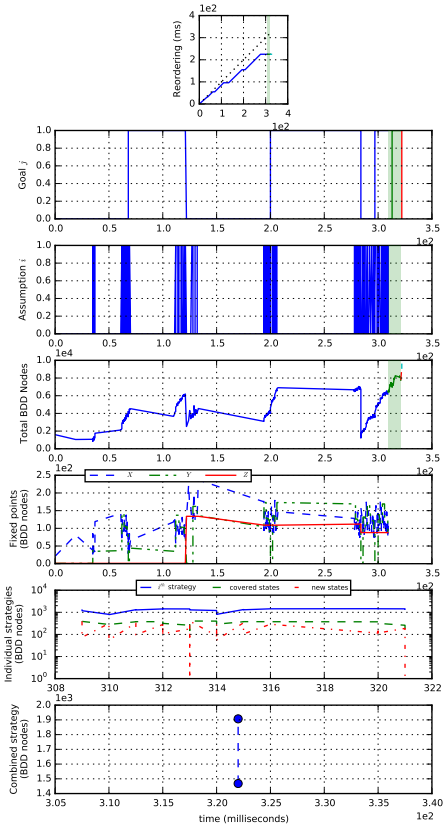


(b) 19 masters

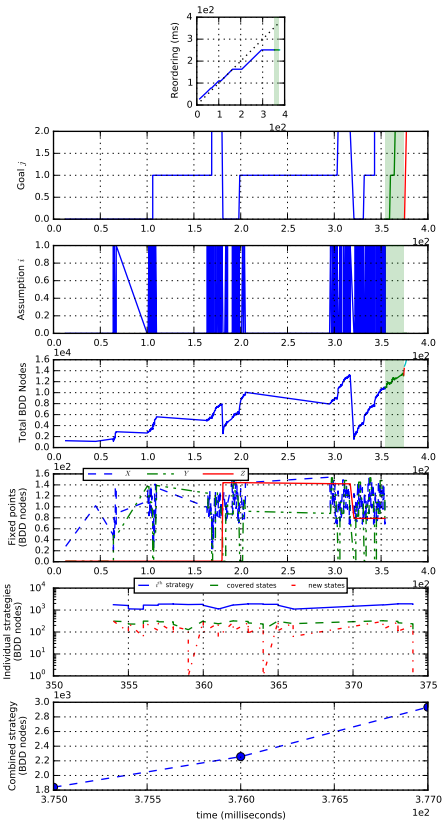


(c) 20 masters

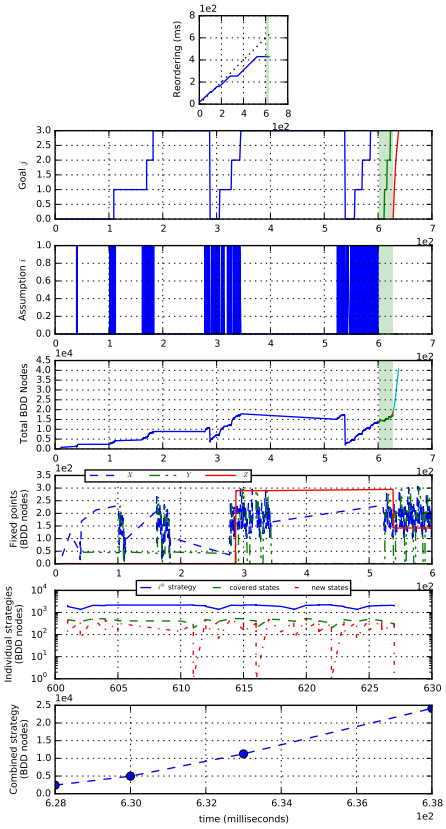
Figure 17: Revised spec with conjunction and strategy reordering.



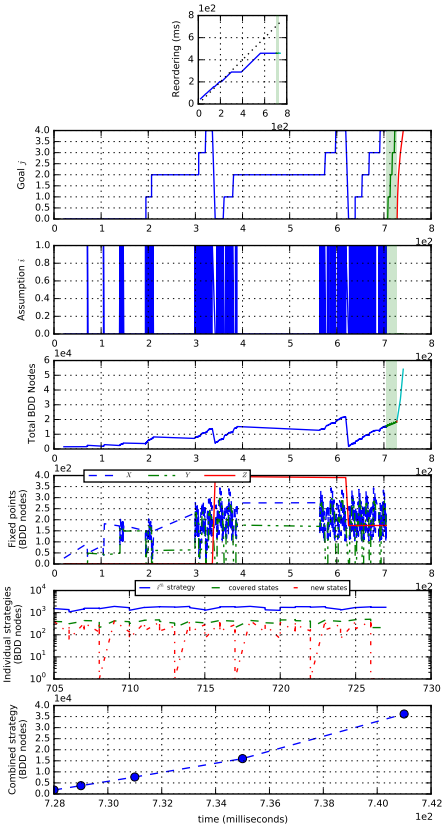
(a) 2 masters



(b) 3 masters

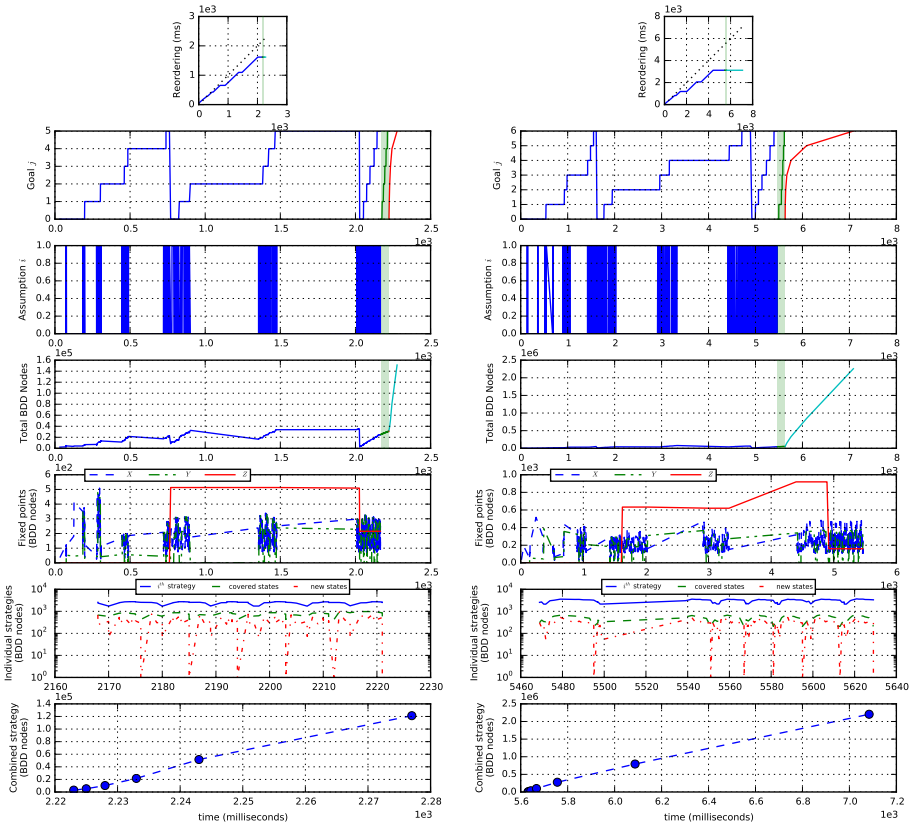


(c) 4 masters



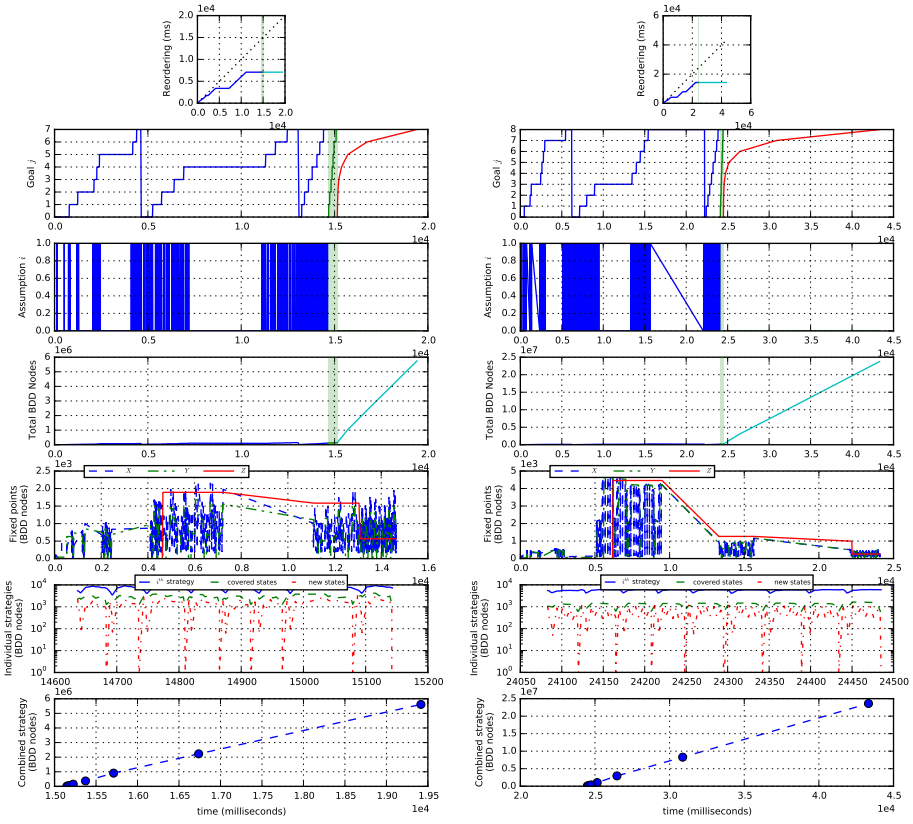
(d) 5 masters

Figure 18: Revised spec with conjunction but no strategy reordering.



(a) 6 masters

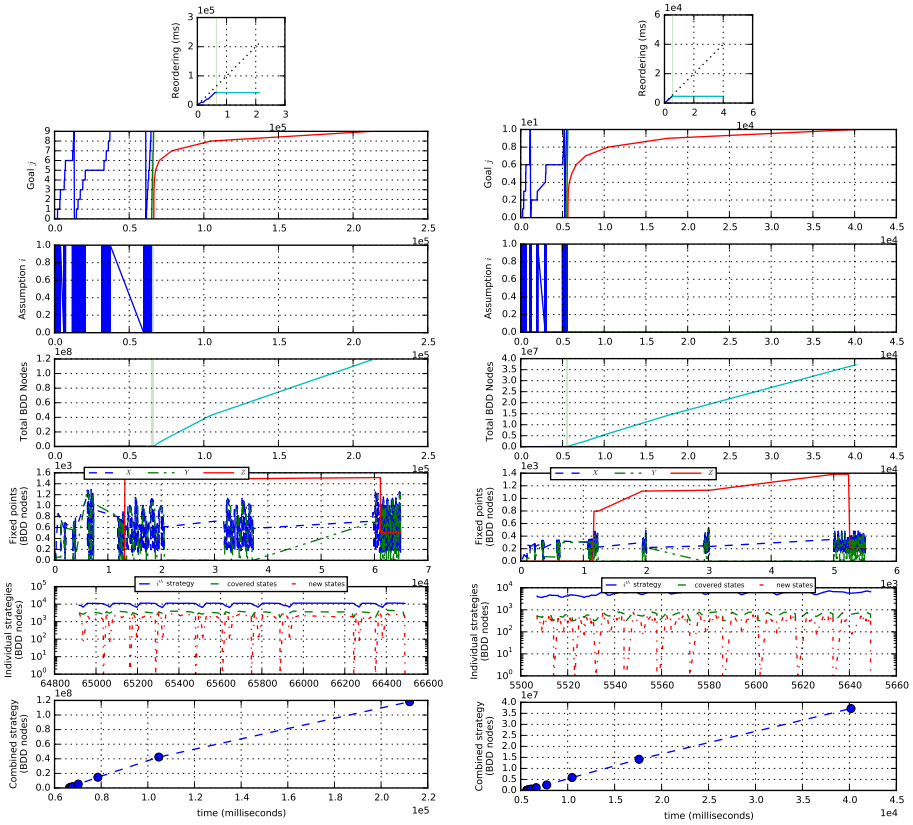
(b) 7 masters



(c) 8 masters

(d) 9 masters

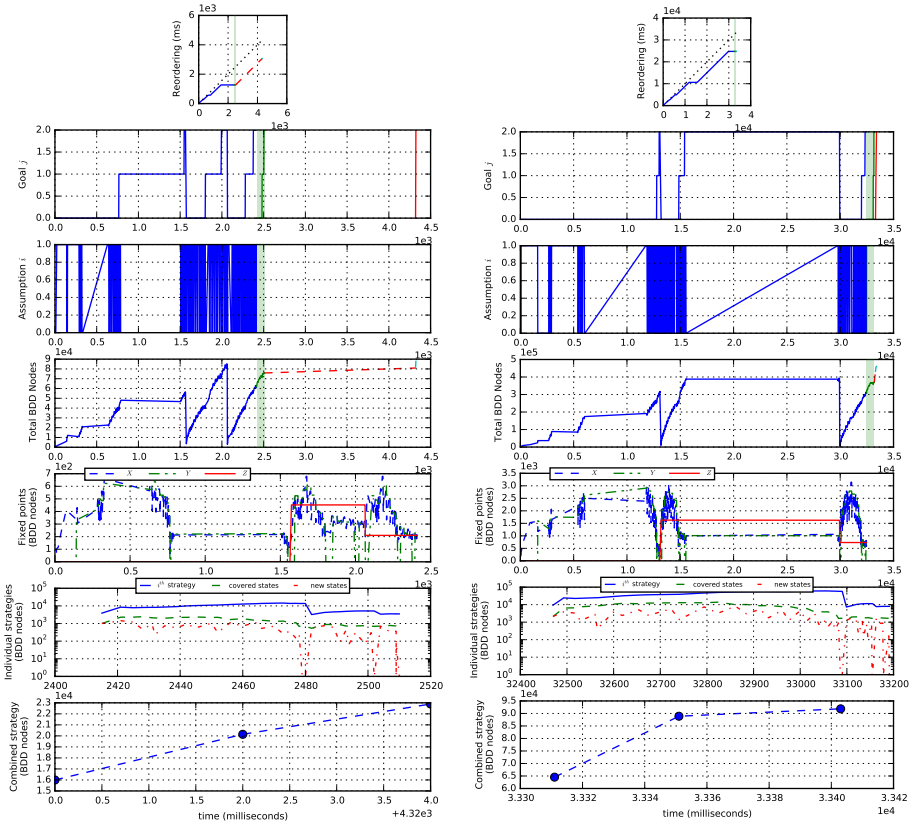
Figure 19: Revised spec with conjunction but no strategy reordering.



(a) 10 masters

(b) 11 masters

Figure 20: Revised spec with conjunction but no strategy reordering.



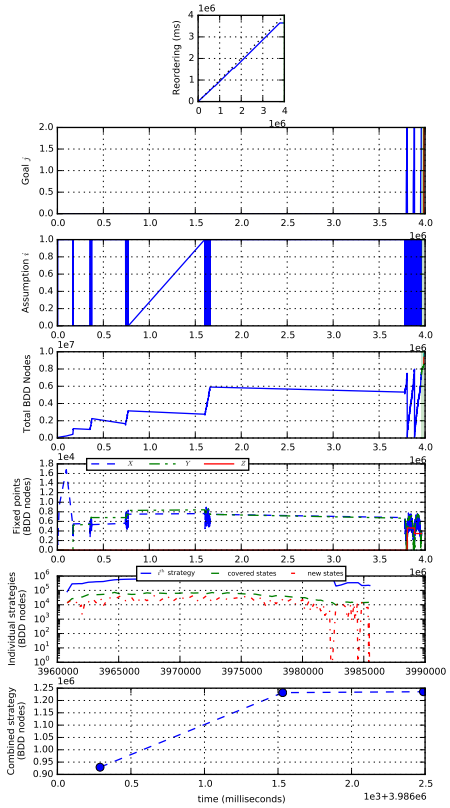
(a) 2 masters

(b) 3 masters

(c) 4 masters

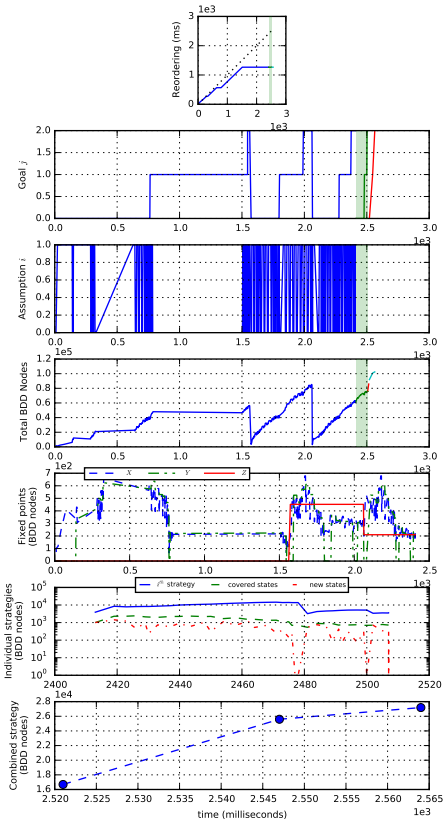
(d) 5 masters

Figure 21: Original spec with BA and strategy reordering.

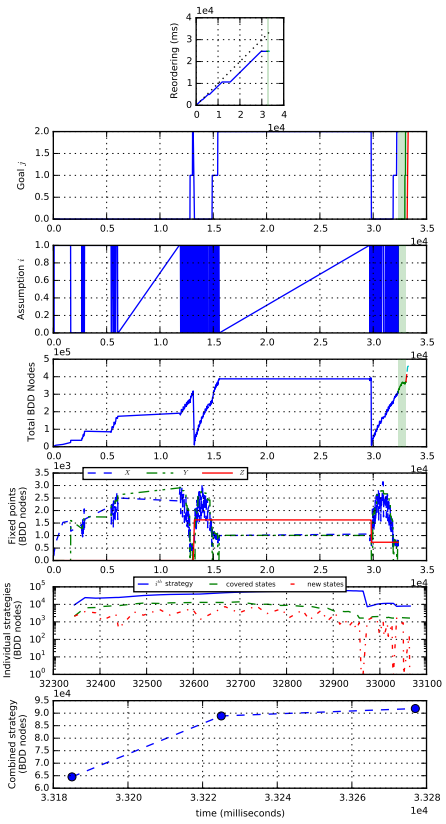


(a) 6 masters

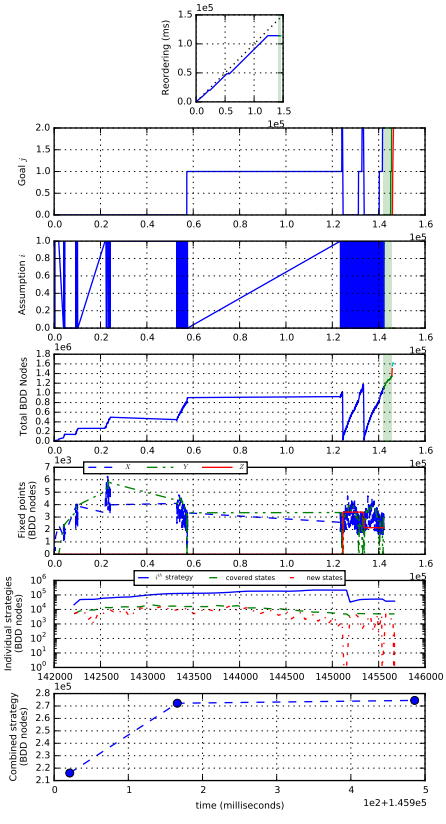
Figure 22: Original spec with BA and strategy reordering.



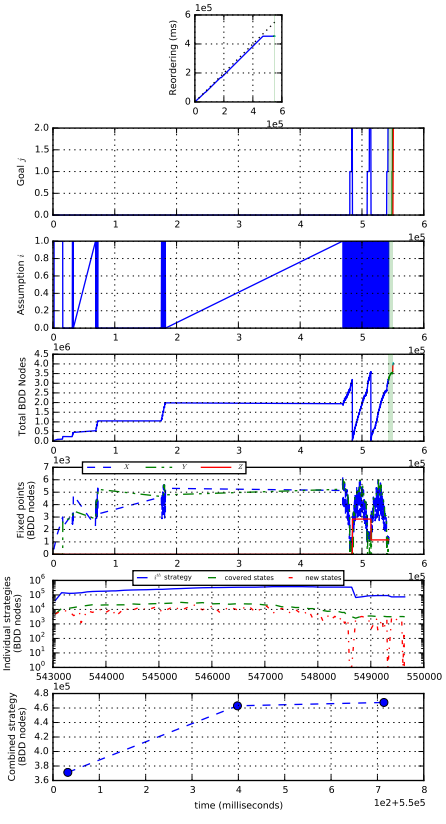
(a) 2 masters



(b) 3 masters

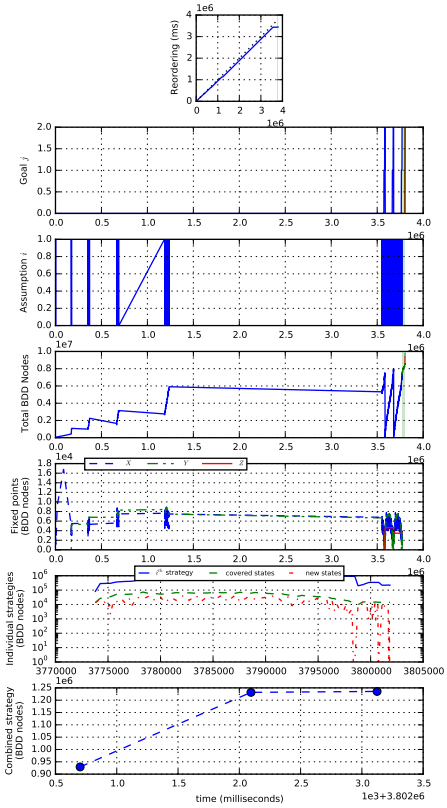


(c) 4 masters

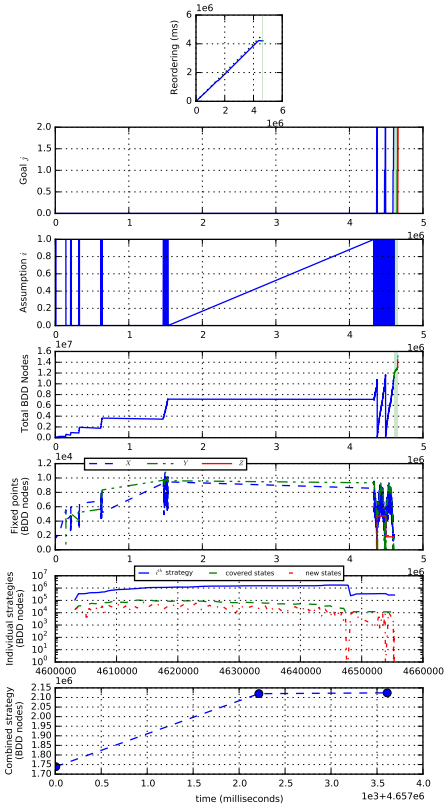


(d) 5 masters

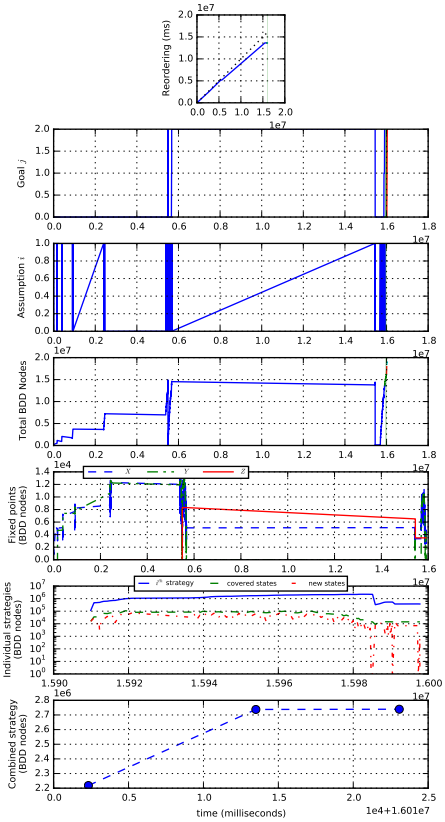
Figure 23: Original spec with BA but no strategy reordering.



(a) 6 masters

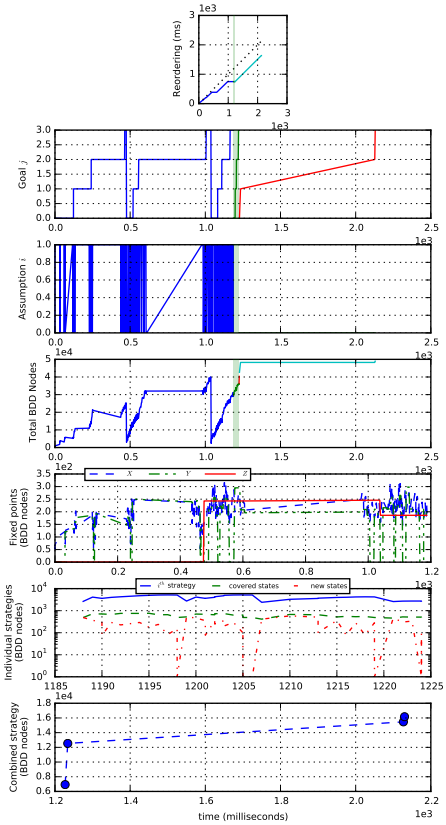


(b) 7 masters

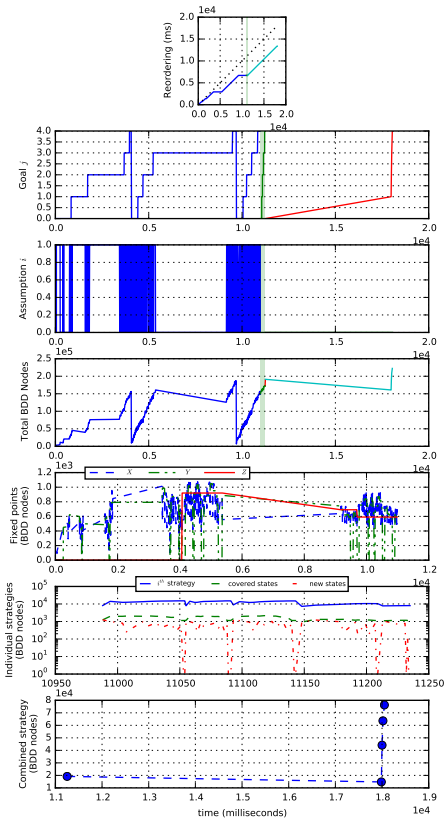


(c) 8 masters

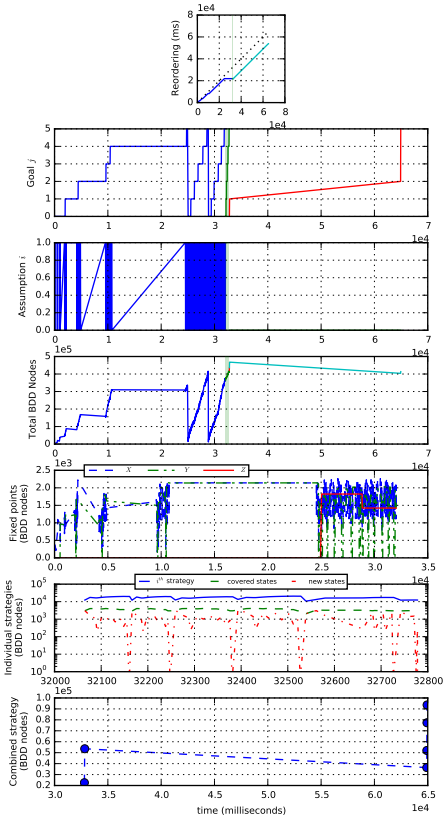
Figure 24: Original spec with BA but no strategy reordering.



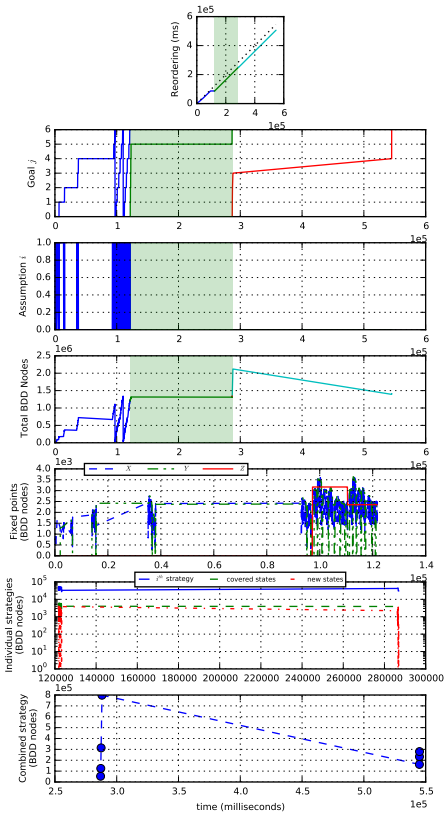
(a) 2 masters



(b) 3 masters

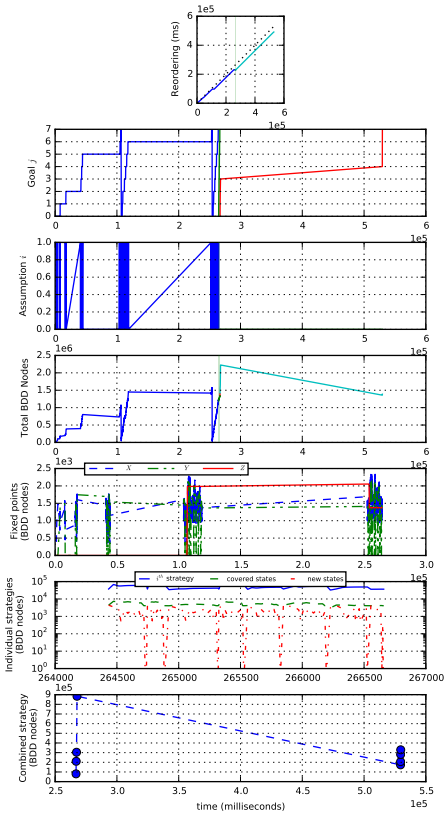


(c) 4 masters

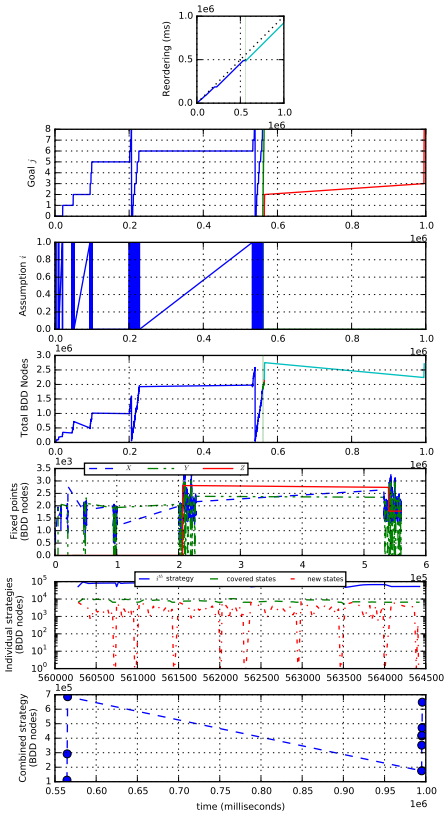


(d) 5 masters

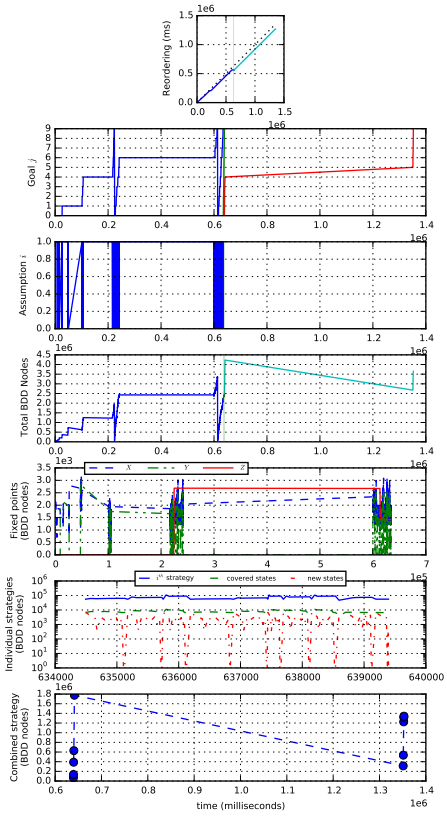
Figure 25: Original spec with conjunction and strategy reordering.



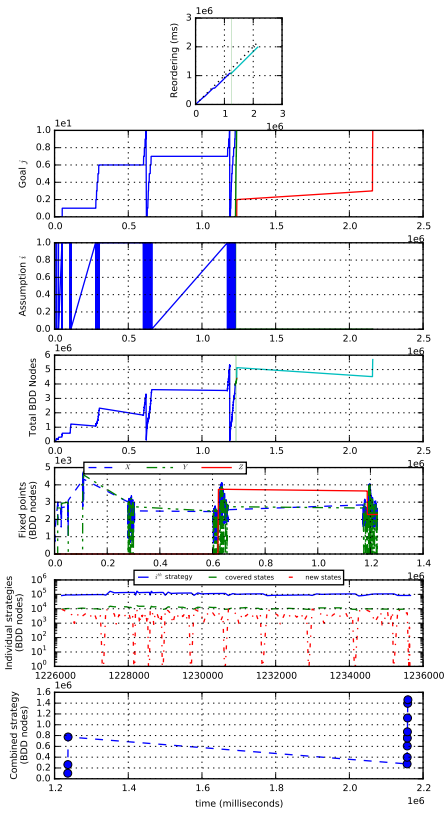
(a) 6 masters



(b) 7 masters

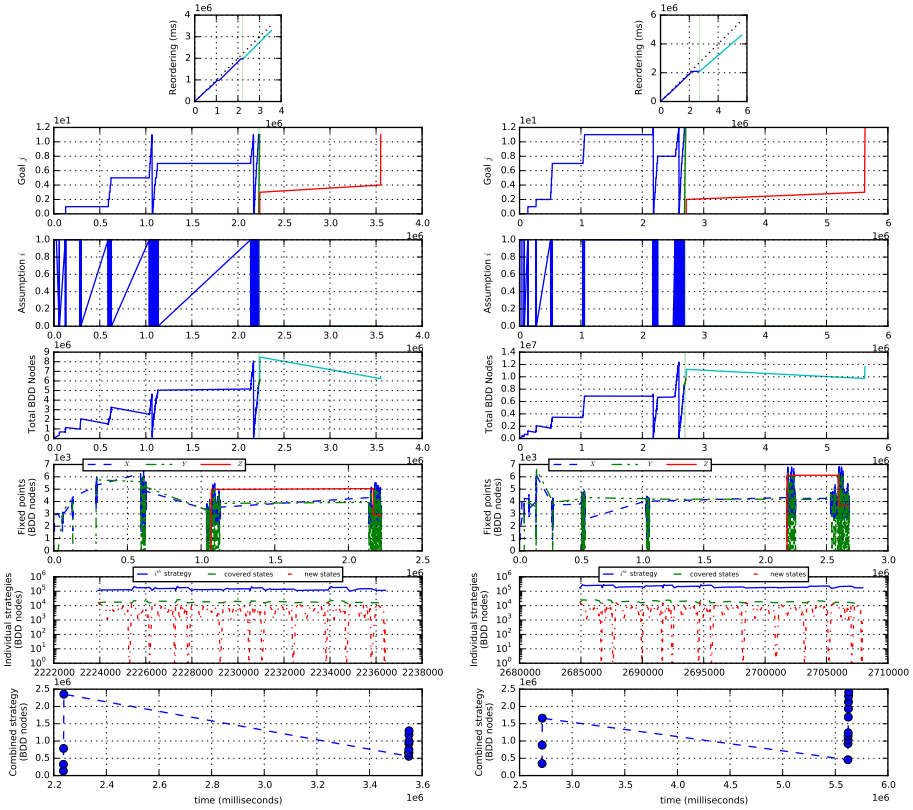


(c) 8 masters

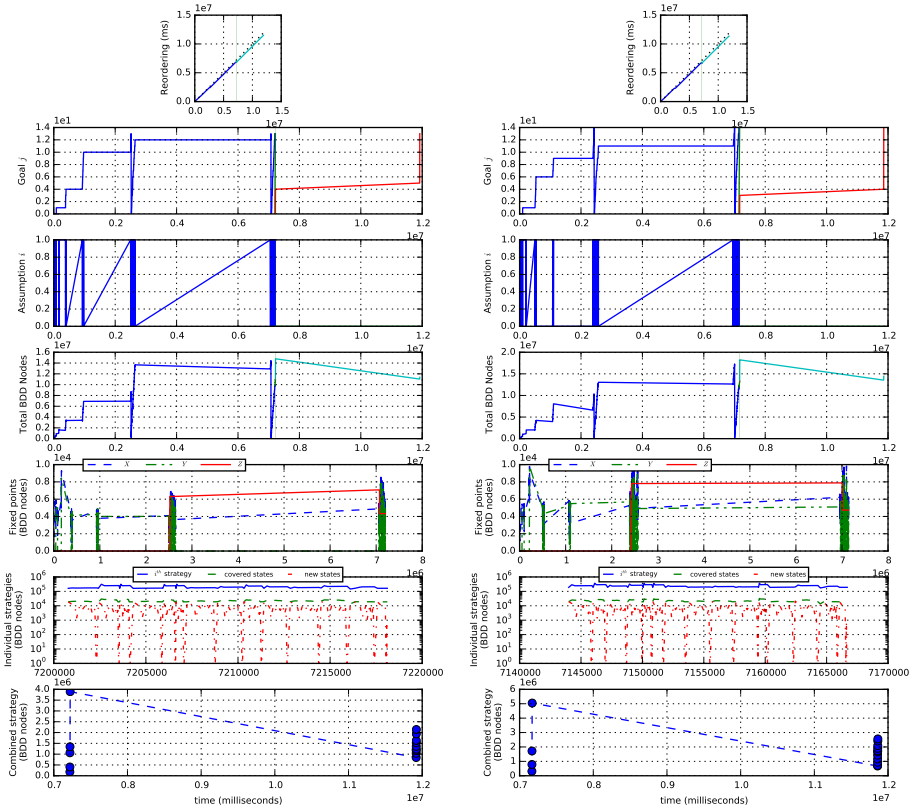


(d) 9 masters

Figure 26: Original spec with conjunction and strategy reordering.

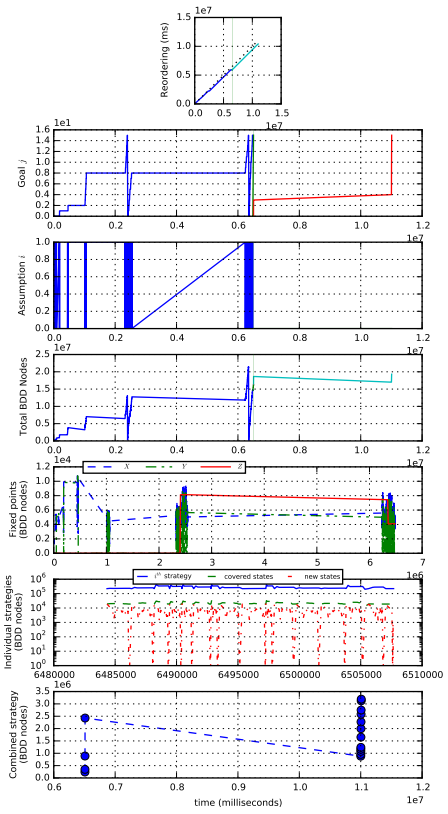


(a) 10 masters (b) 11 masters

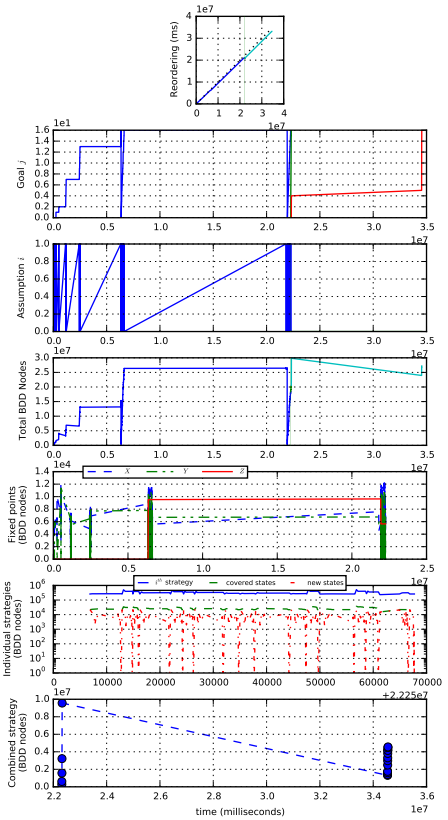


(c) 12 masters (d) 13 masters

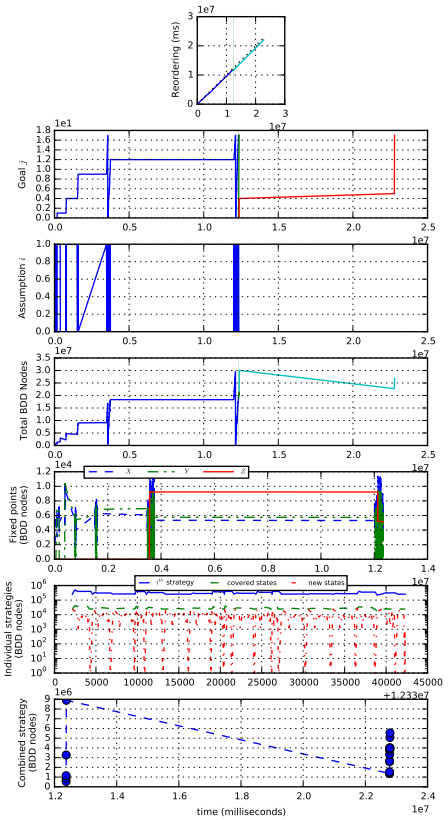
Figure 27: Original spec with conjunction and strategy reordering.



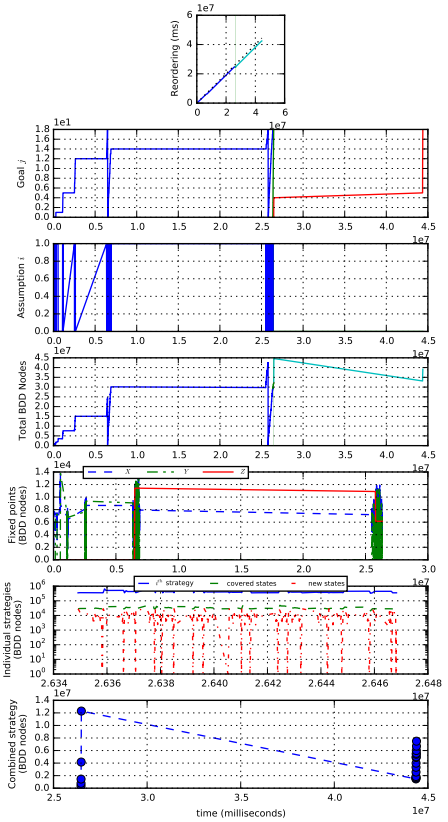
(a) 14 masters



(b) 15 masters

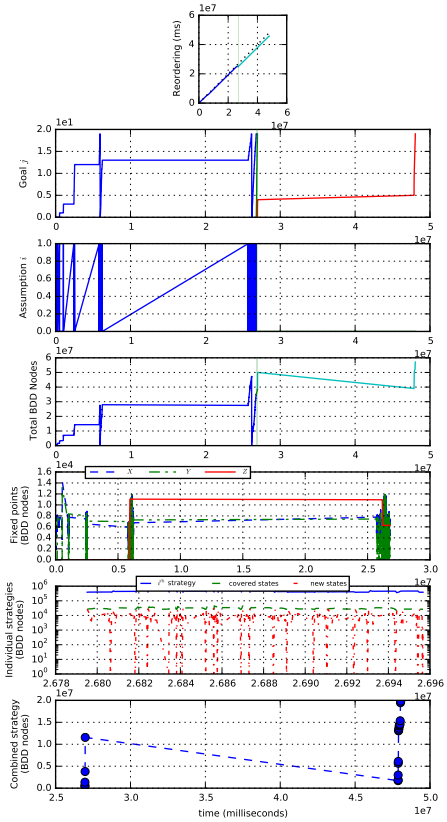


(c) 16 masters



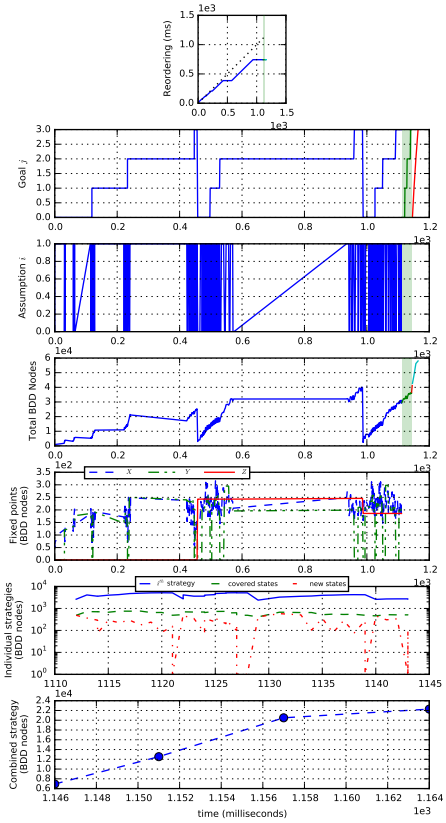
(d) 17 masters

Figure 28: Original spec with conjunction and strategy reordering.

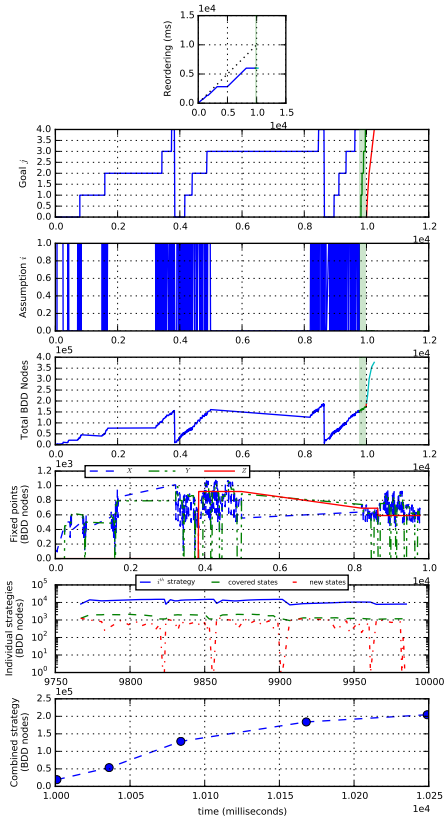


(a) 18 masters

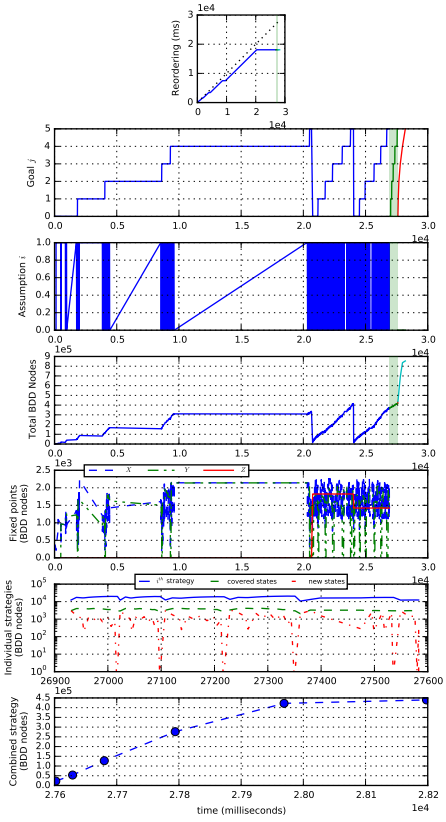
Figure 29: Original spec with conjunction and strategy reordering.



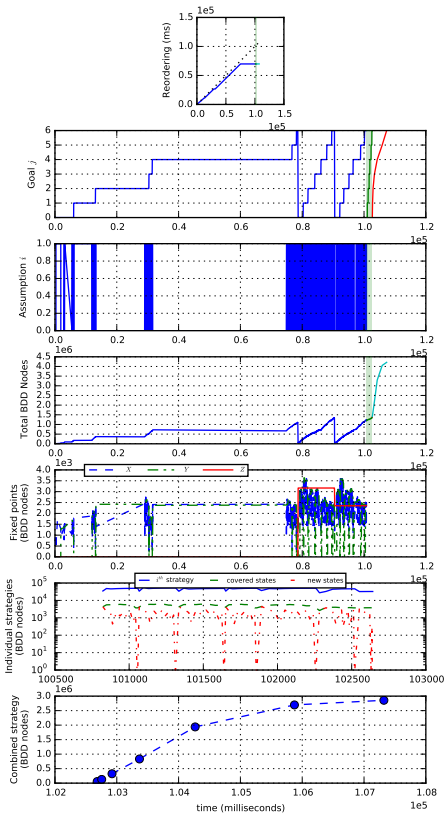
(a) 2 masters



(b) 3 masters

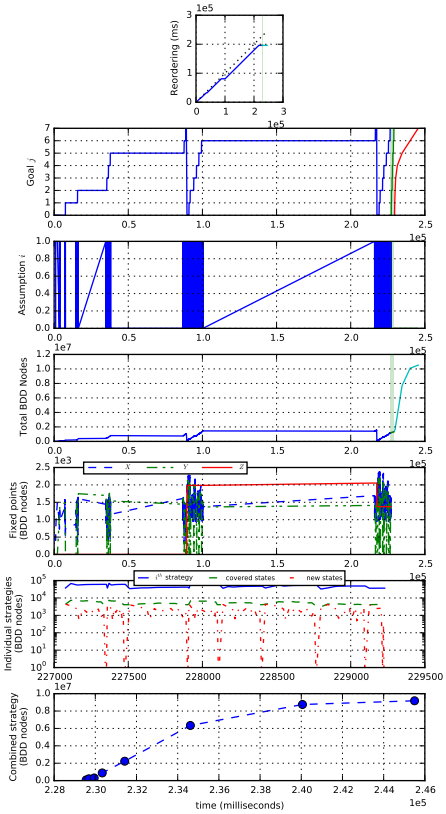


(c) 4 masters

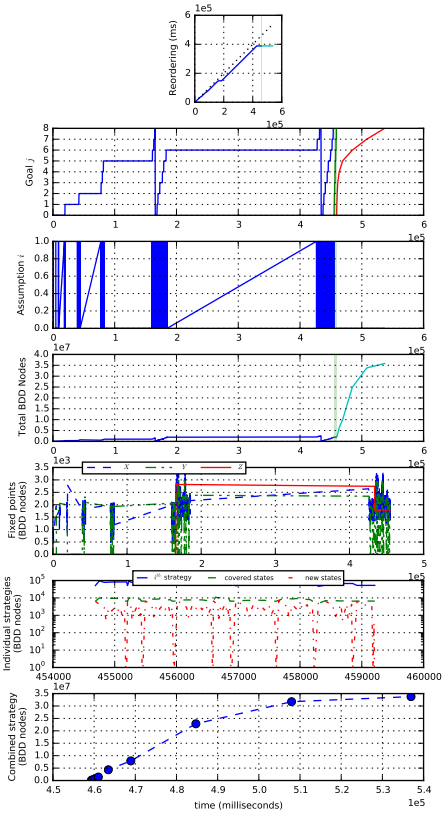


(d) 5 masters

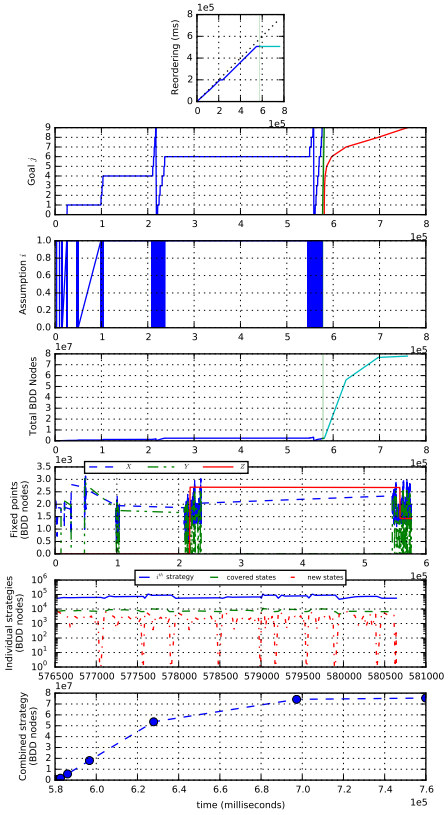
Figure 30: Original spec with conjunction but no strategy reordering (the last runs were run after an upgrade from 11GB RAM to 27GB RAM.



(a) 6 masters



(b) 7 masters



(c) 8 masters

Figure 31: Original spec with conjunction but no strategy reordering (the last runs were run after an upgrade from 11GB RAM to 27GB RAM.

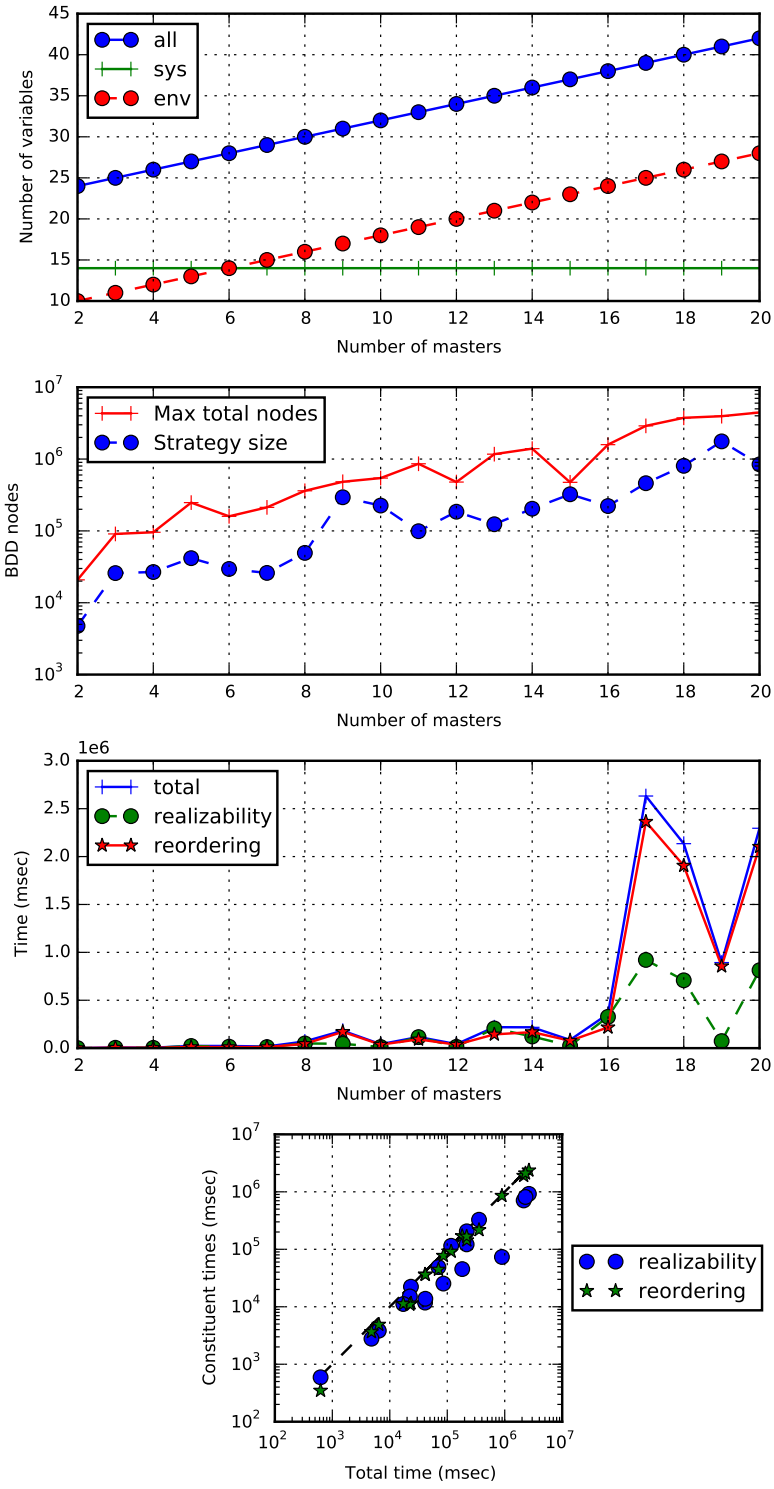


Figure 32: Revised spec with BA and strategy reordering.

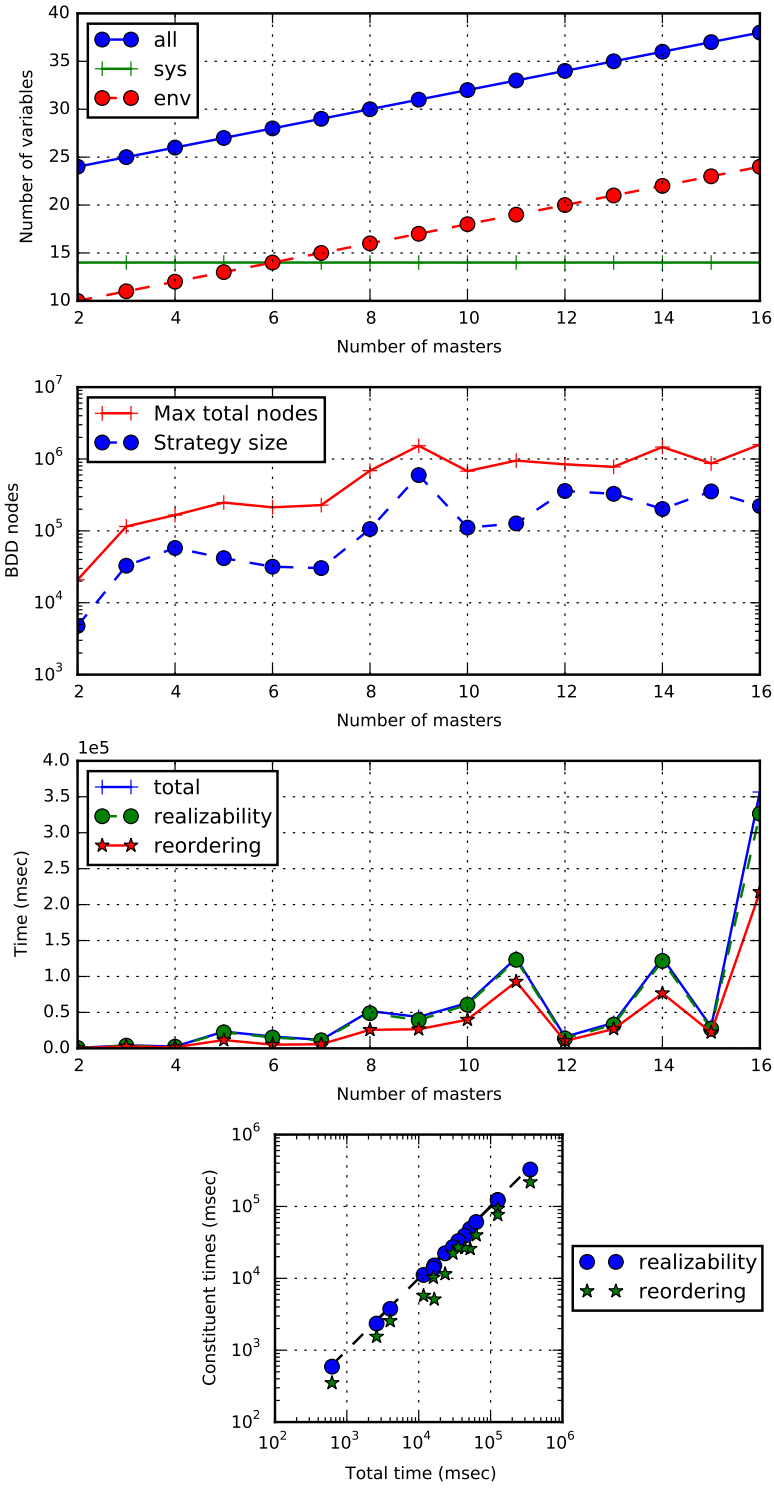


Figure 33: Revised spec with BA but no strategy reordering.

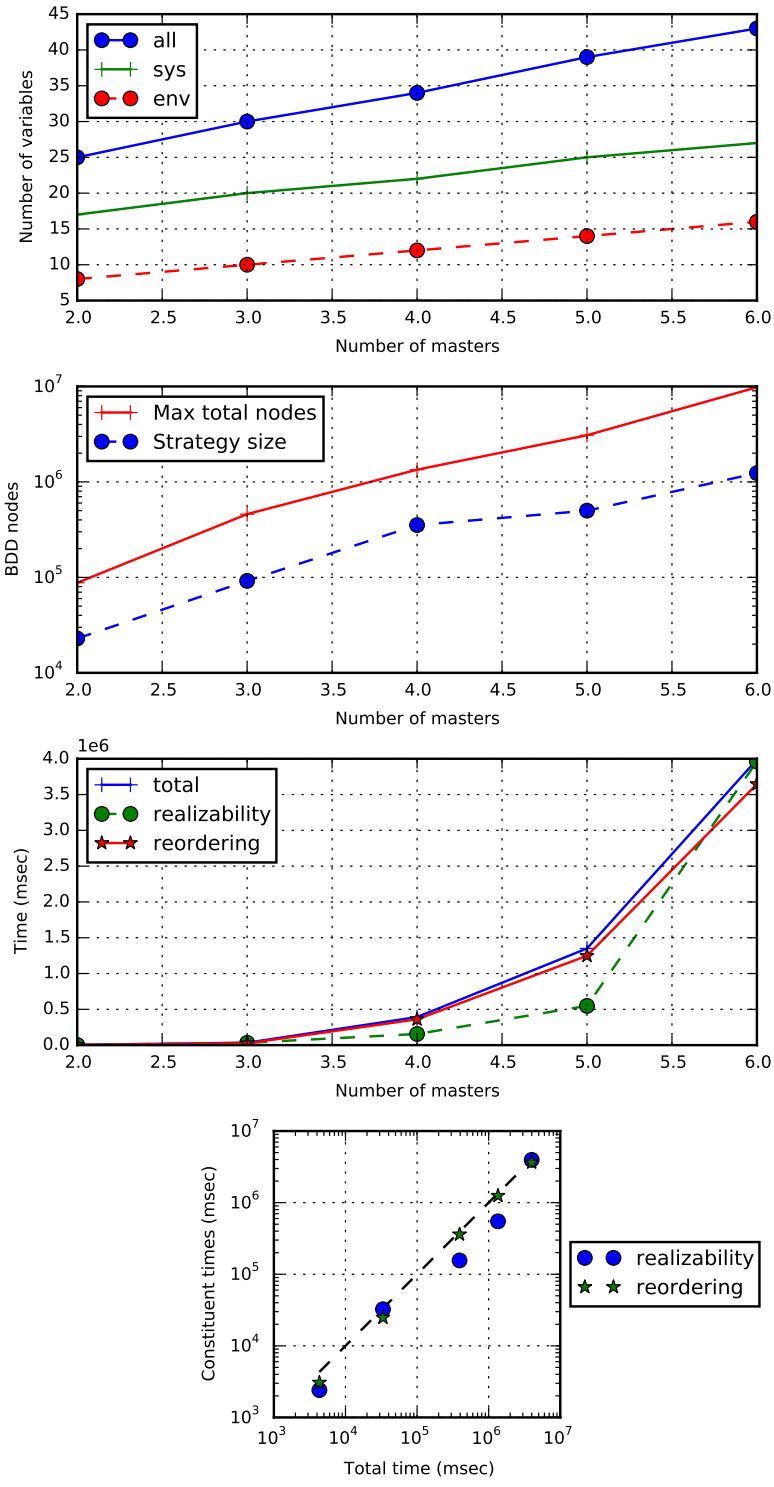


Figure 34: Original spec with BA and strategy reordering.

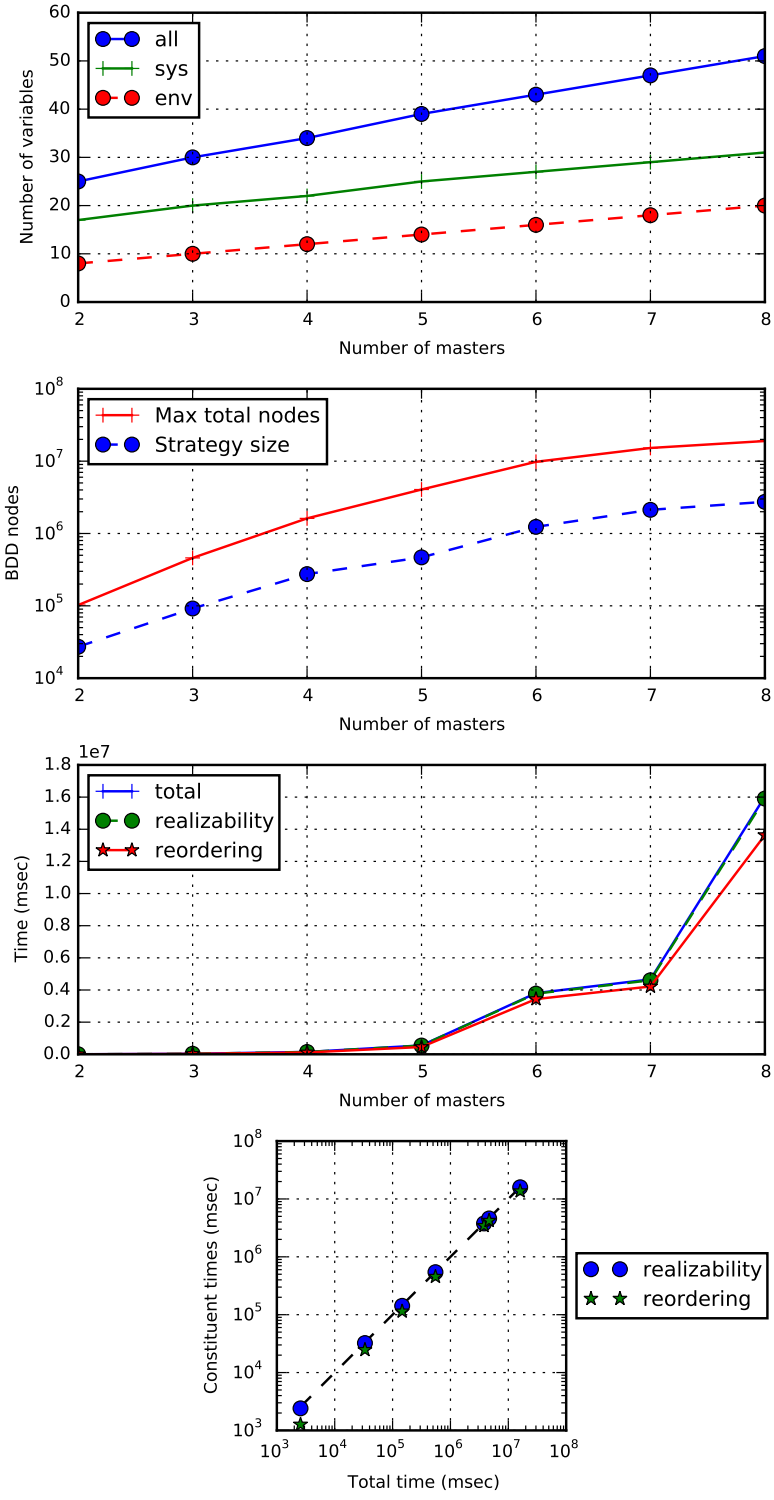


Figure 35: Original spec with BA but no strategy reordering.

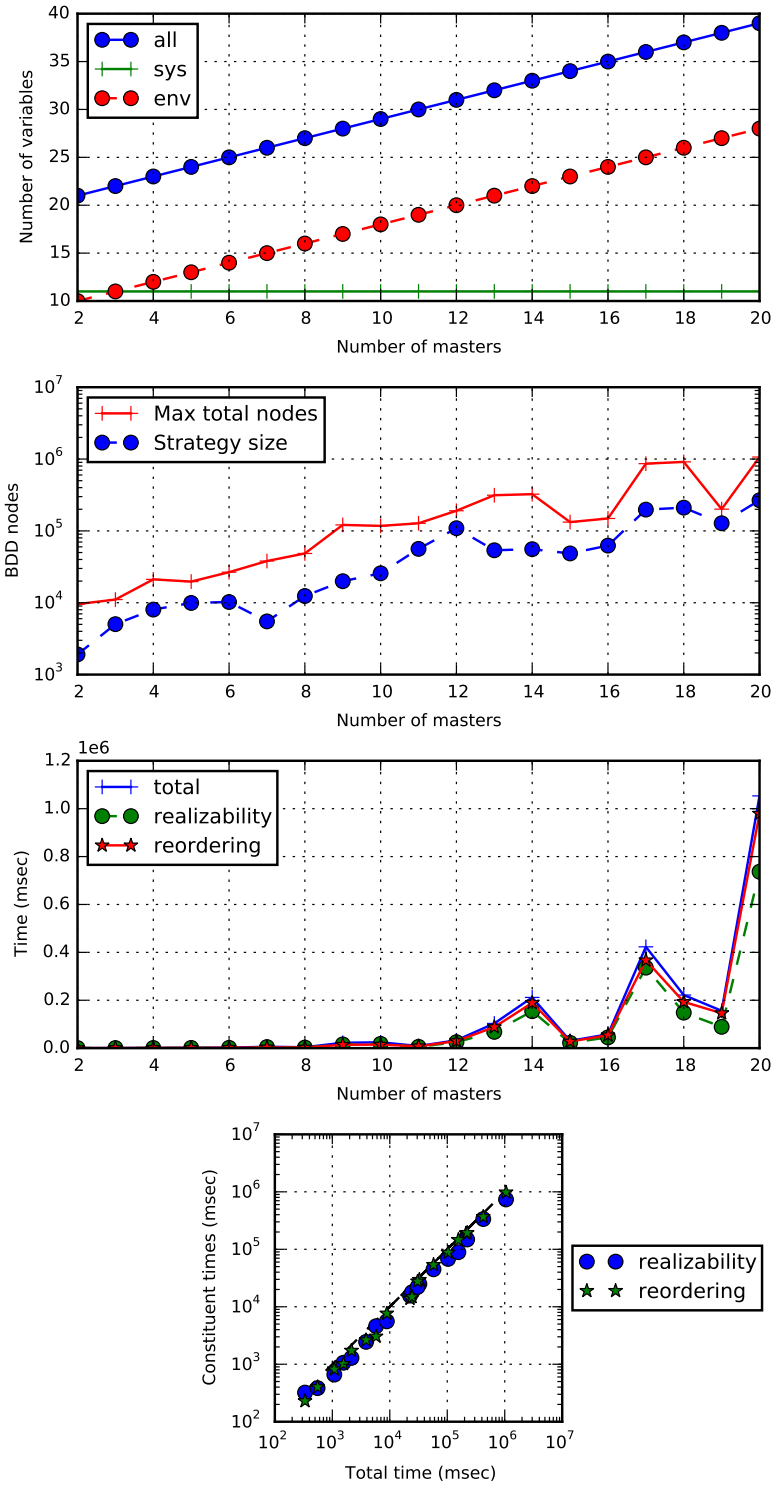


Figure 36: Revised spec with conjunction and strategy reordering.

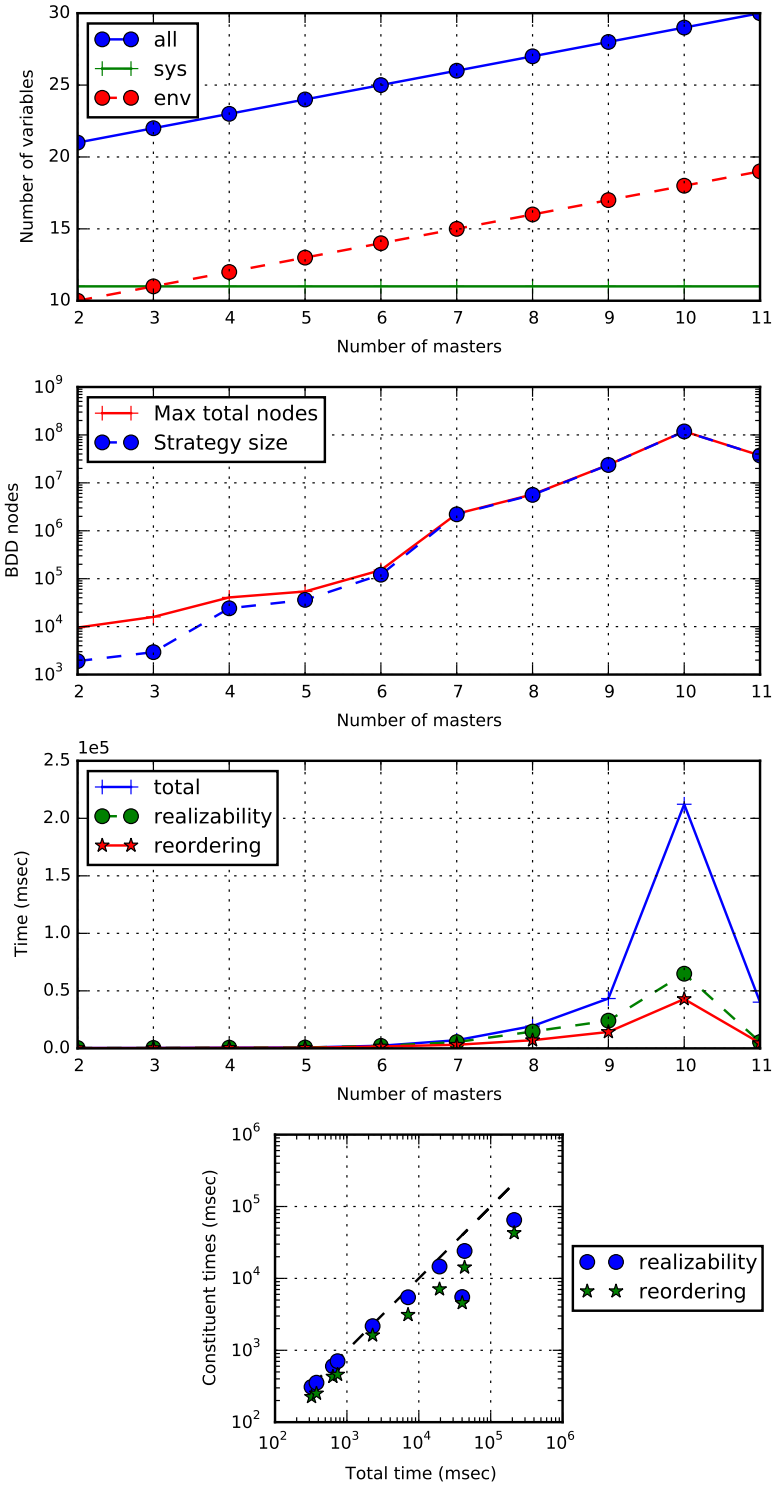


Figure 37: Revised spec with conjunction but no strategy reordering.

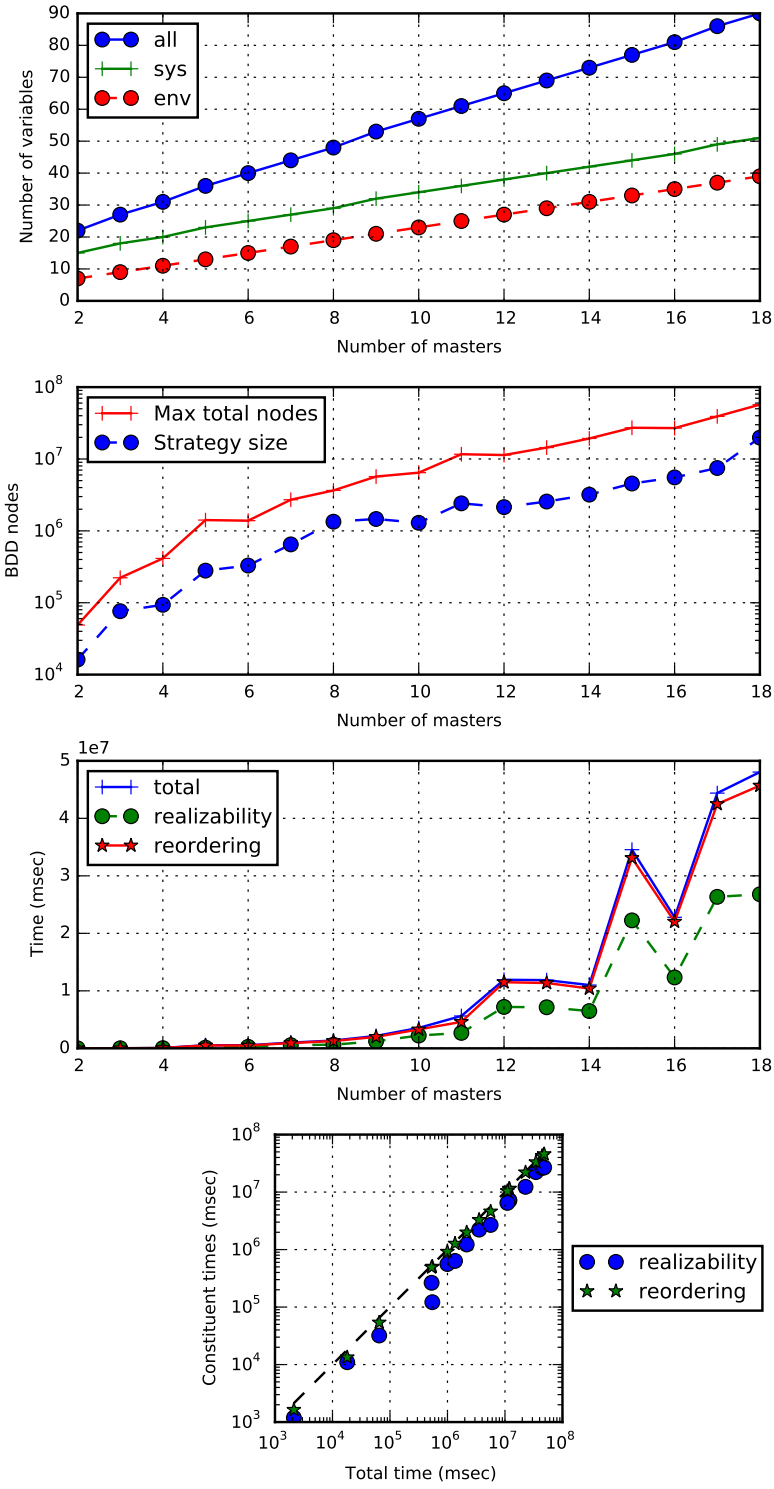


Figure 38: Original spec with conjunction and strategy reordering.

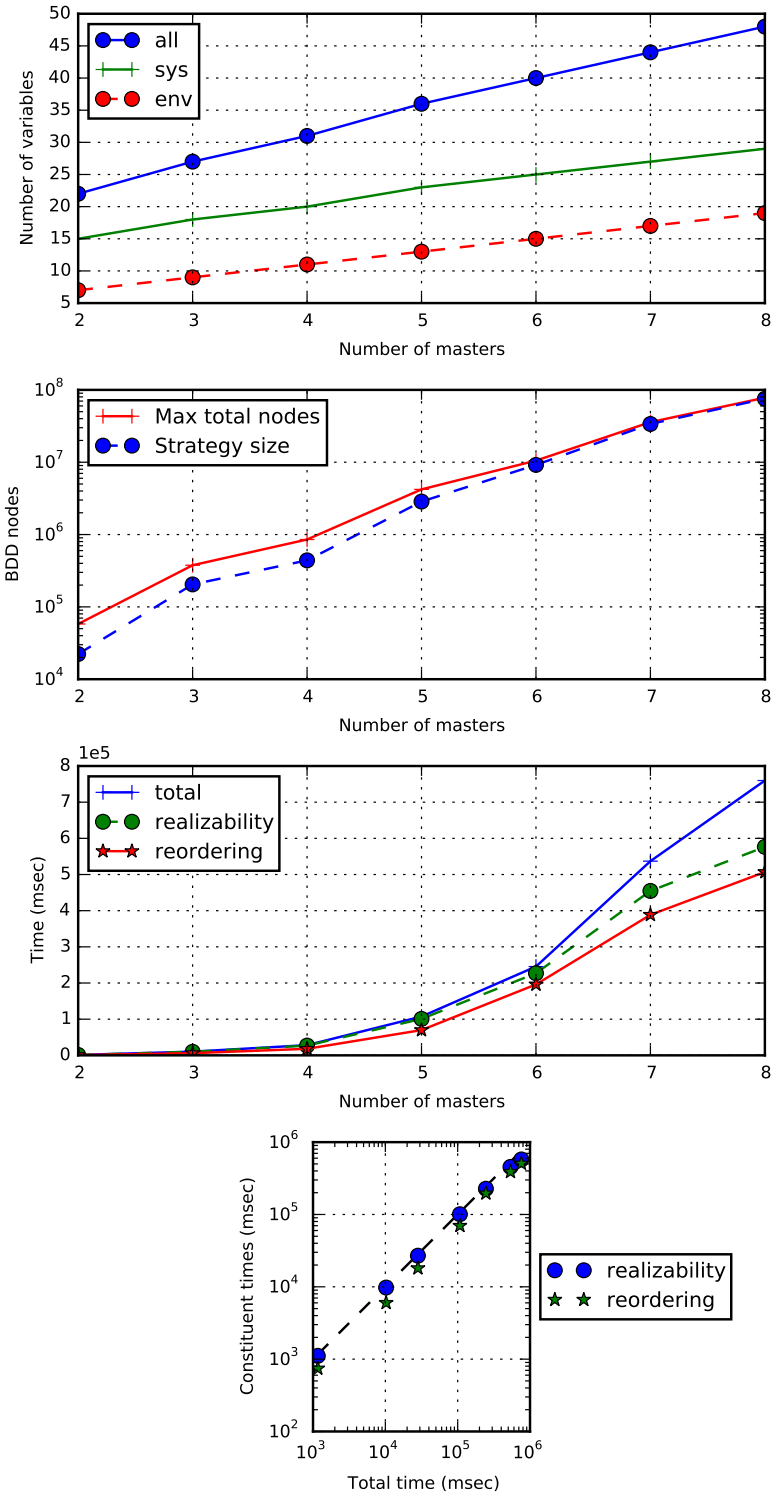


Figure 39: Original spec with conjunction but no strategy reordering (the last runs were run after an upgrade from 11GB RAM to 27GB RAM.

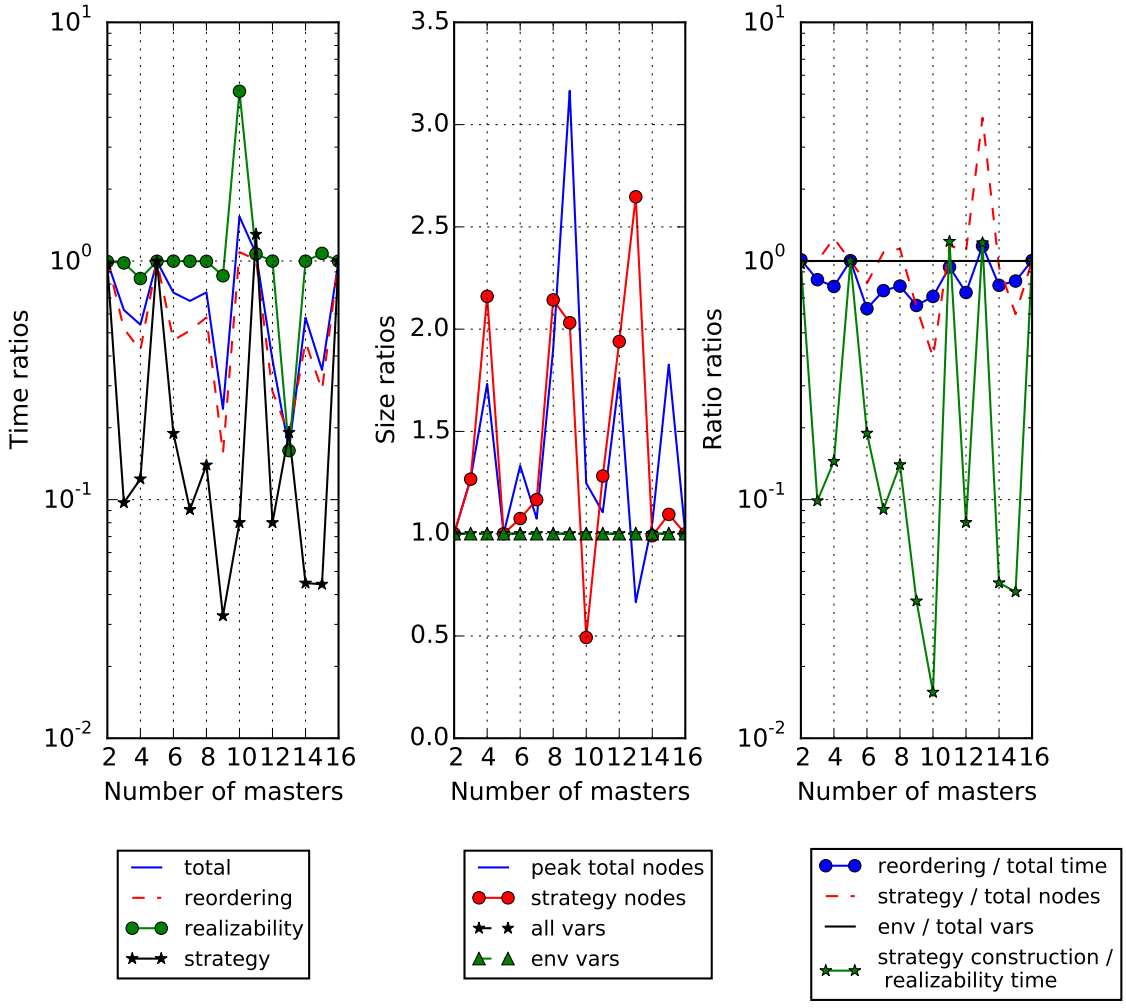


Figure 40: Revised with BA, w/o divided by w/ reordering.

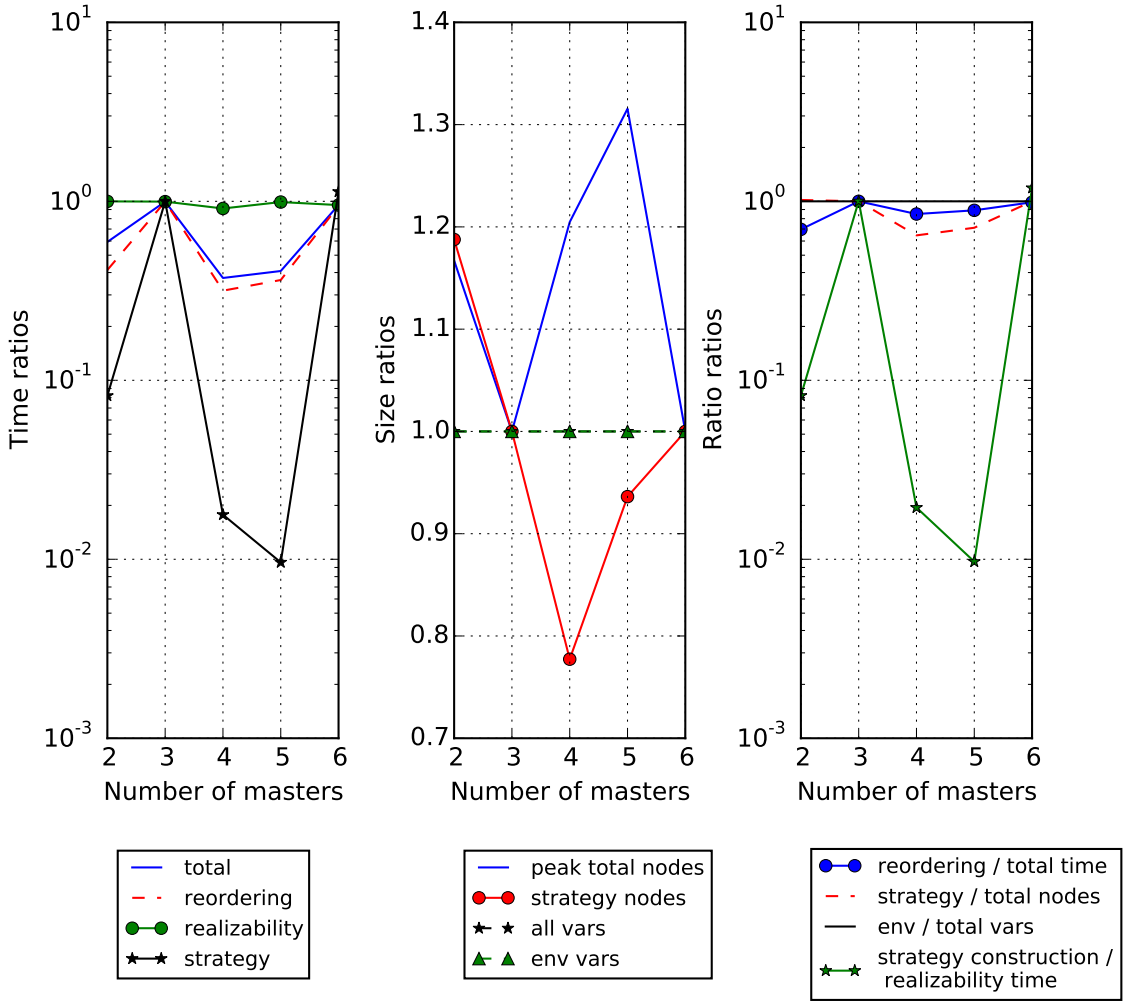


Figure 41: Original with BA, w/o divided by w/ reordering.

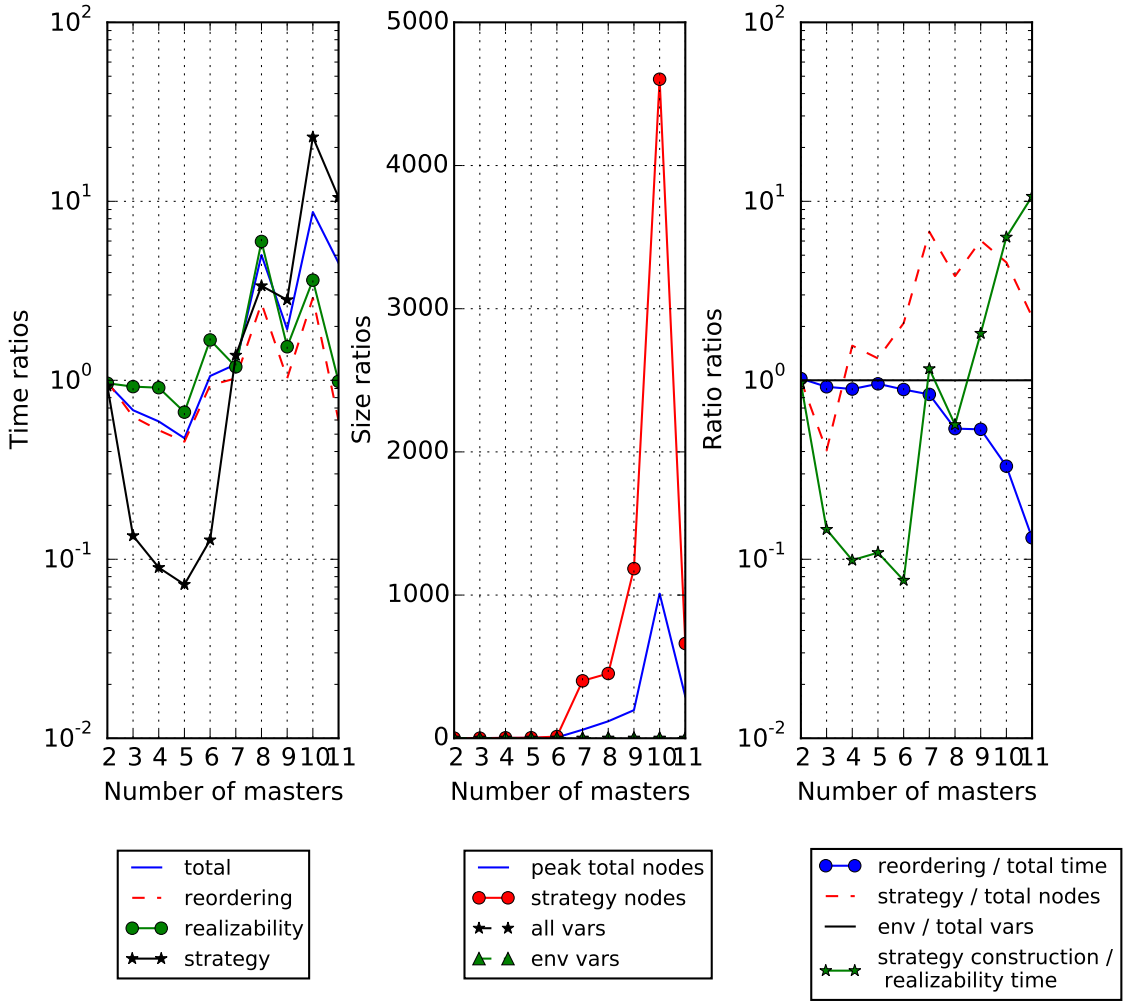


Figure 42: Revised with conjunction, w/o divided by w/ reordering.

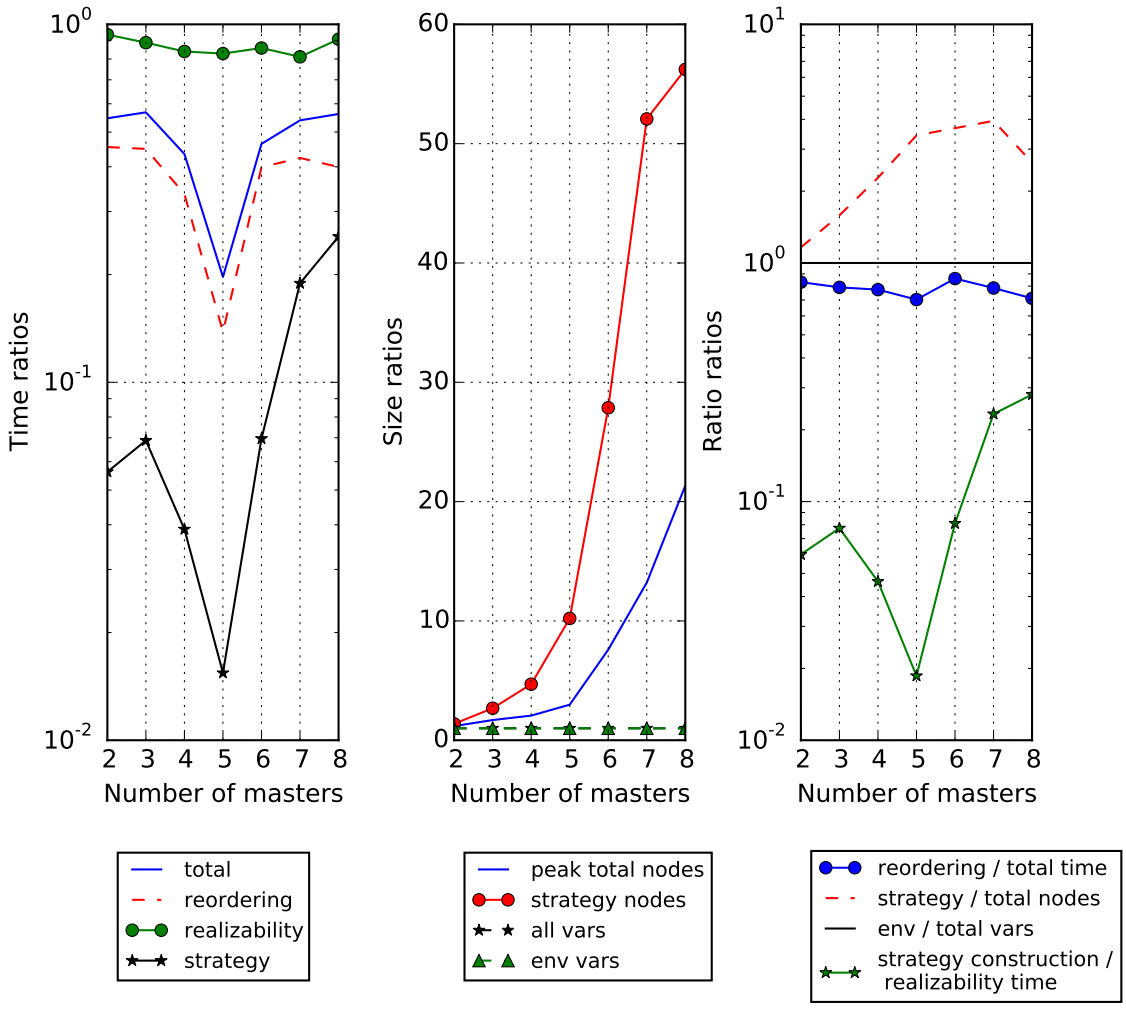


Figure 43: Original with conjunction, w/o divided by w/ reordering.

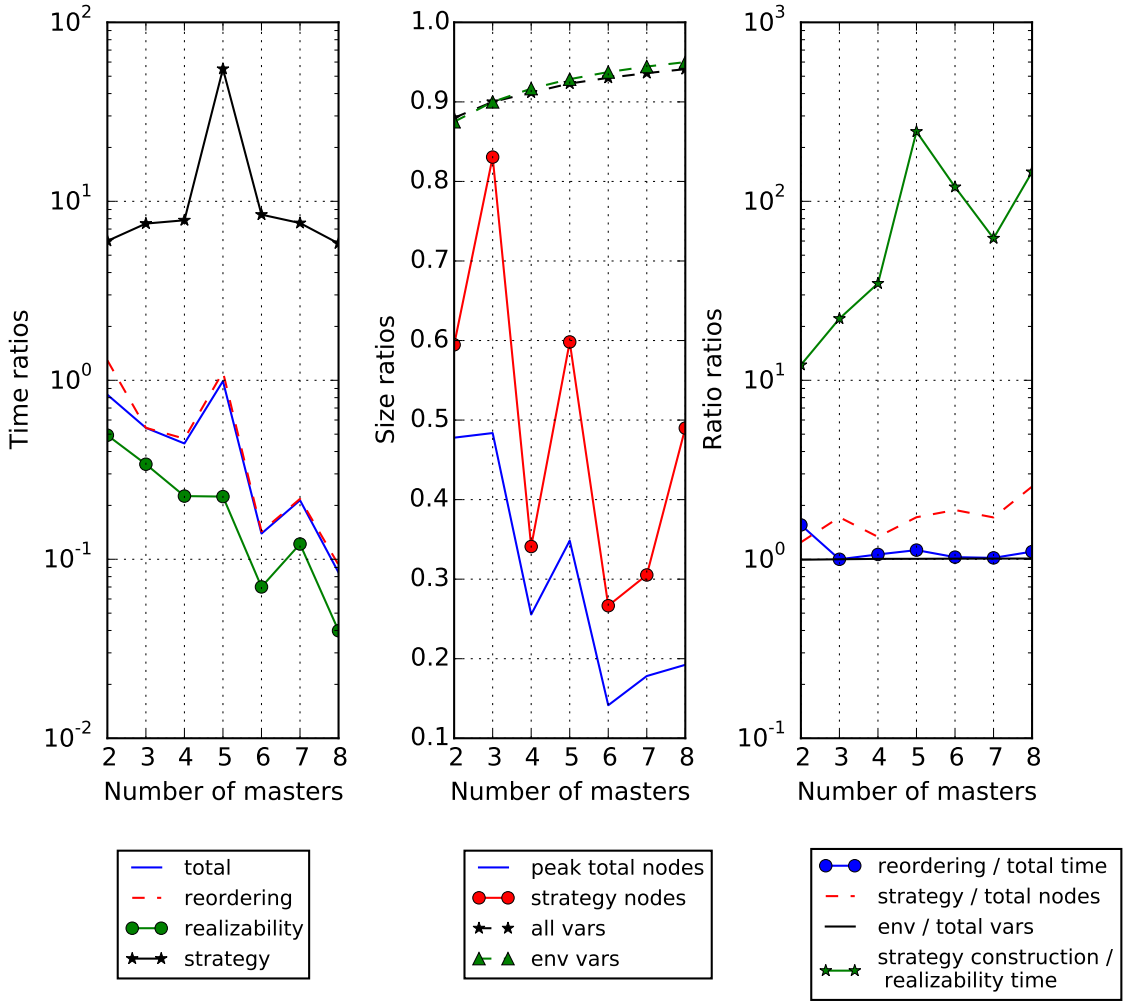


Figure 44: Original with conjunction and with reordering, divided by original BA w/o reordering.

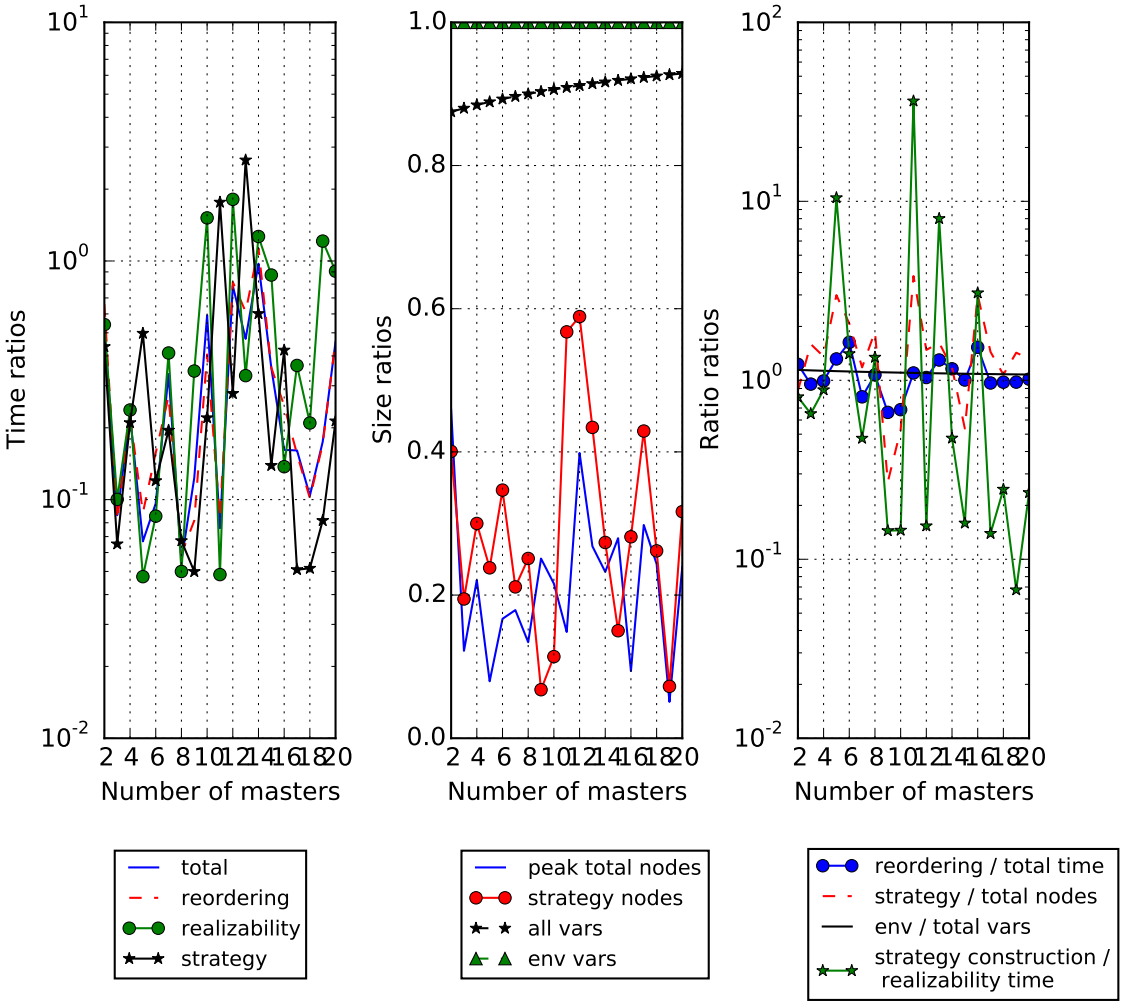


Figure 45: Revised conjunction divided by BA (both with reordering).

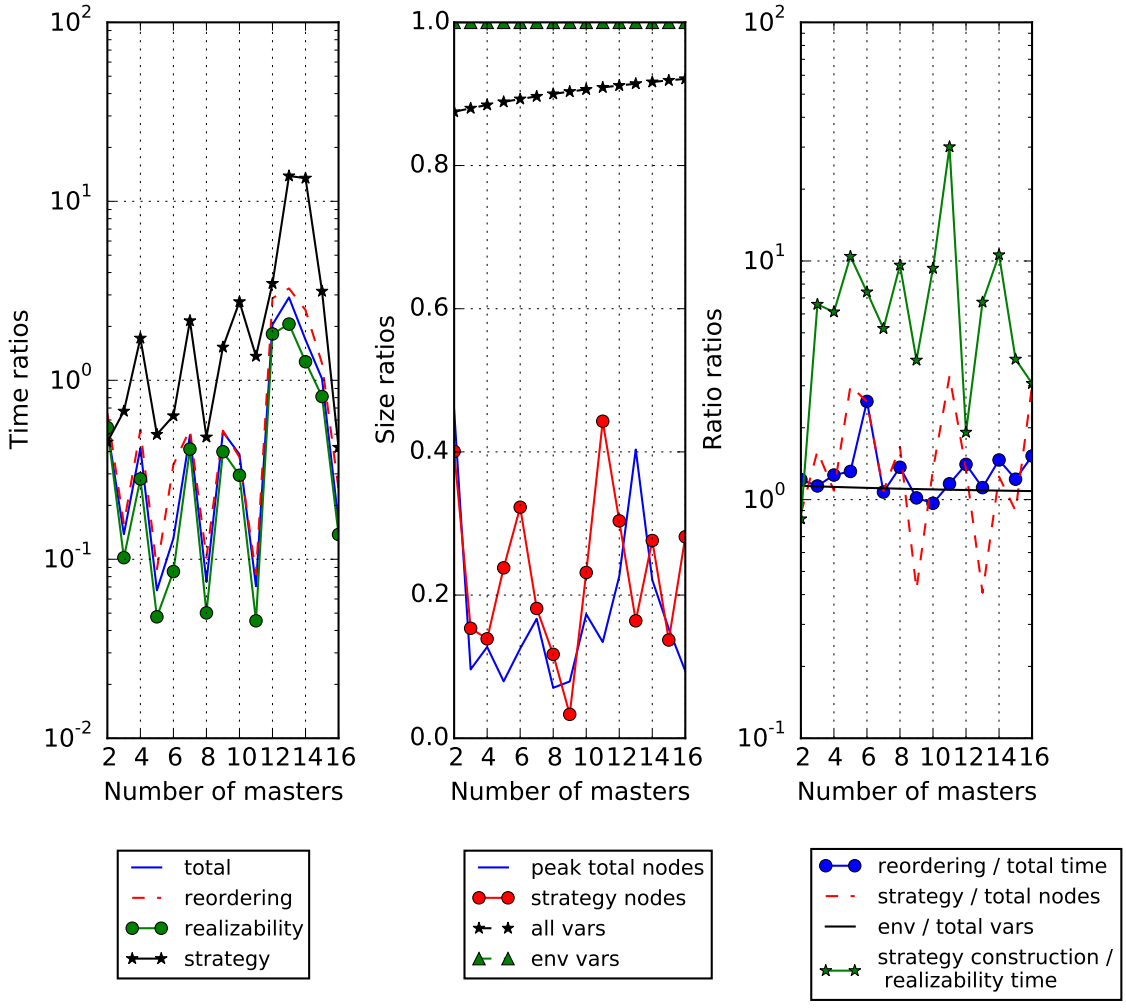


Figure 46: Revised conjunction with reordering, divided by BA w/o reordering.

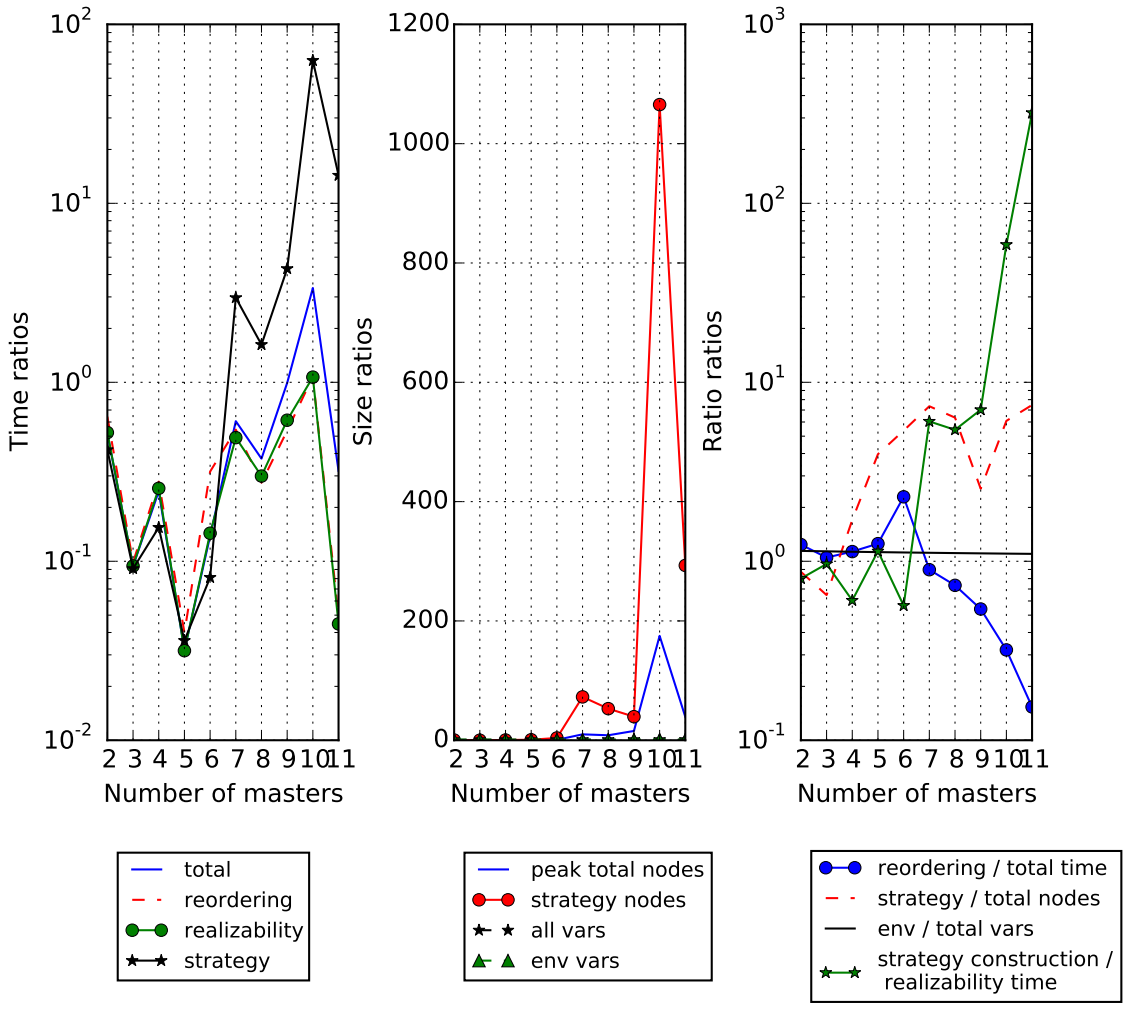


Figure 47: Revised conjunction divided by BA (both w/o reordering).

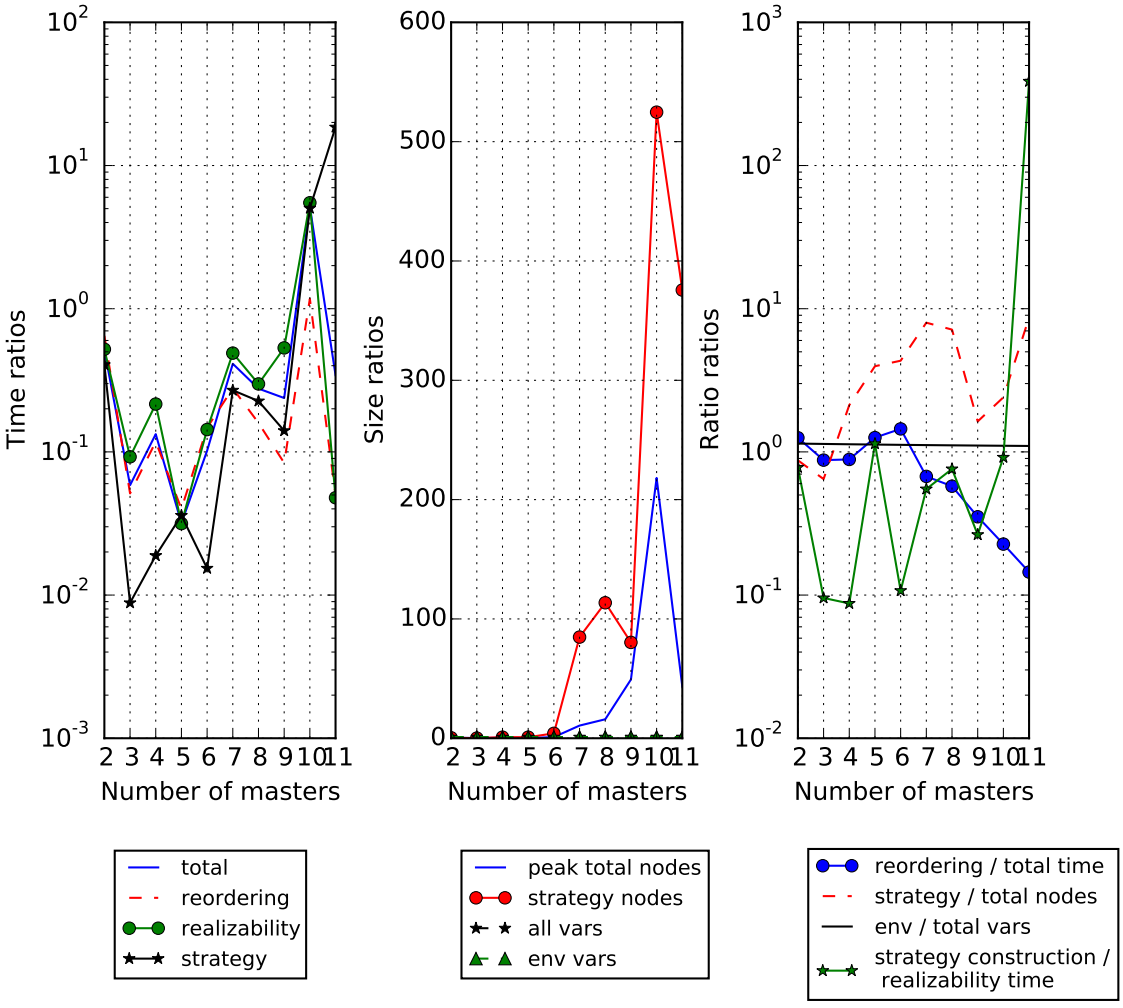


Figure 48: Revised conjunction w/o reordering, divided by BA with reordering.

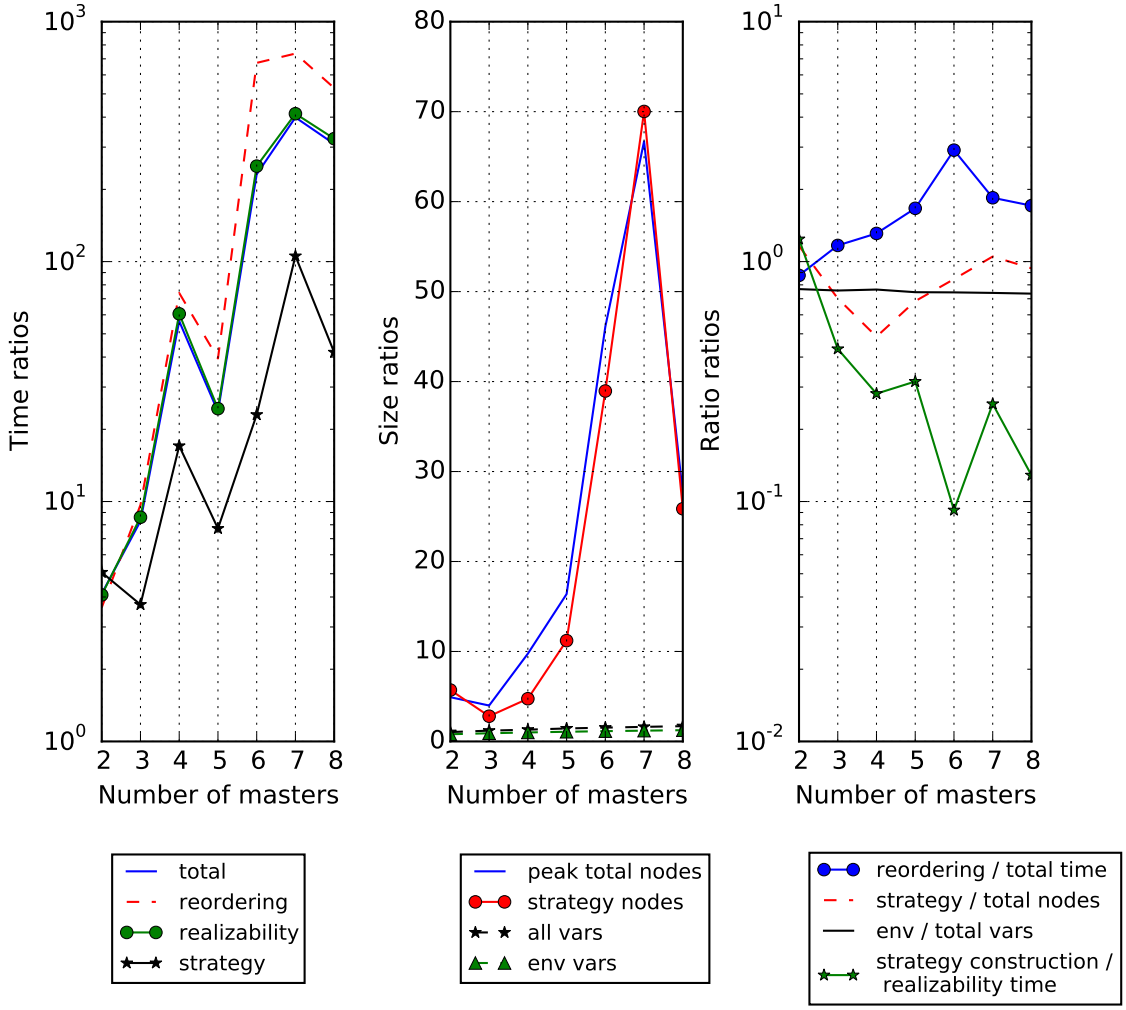


Figure 49: Original with BA divided by revised with BA (both w/o reordering).

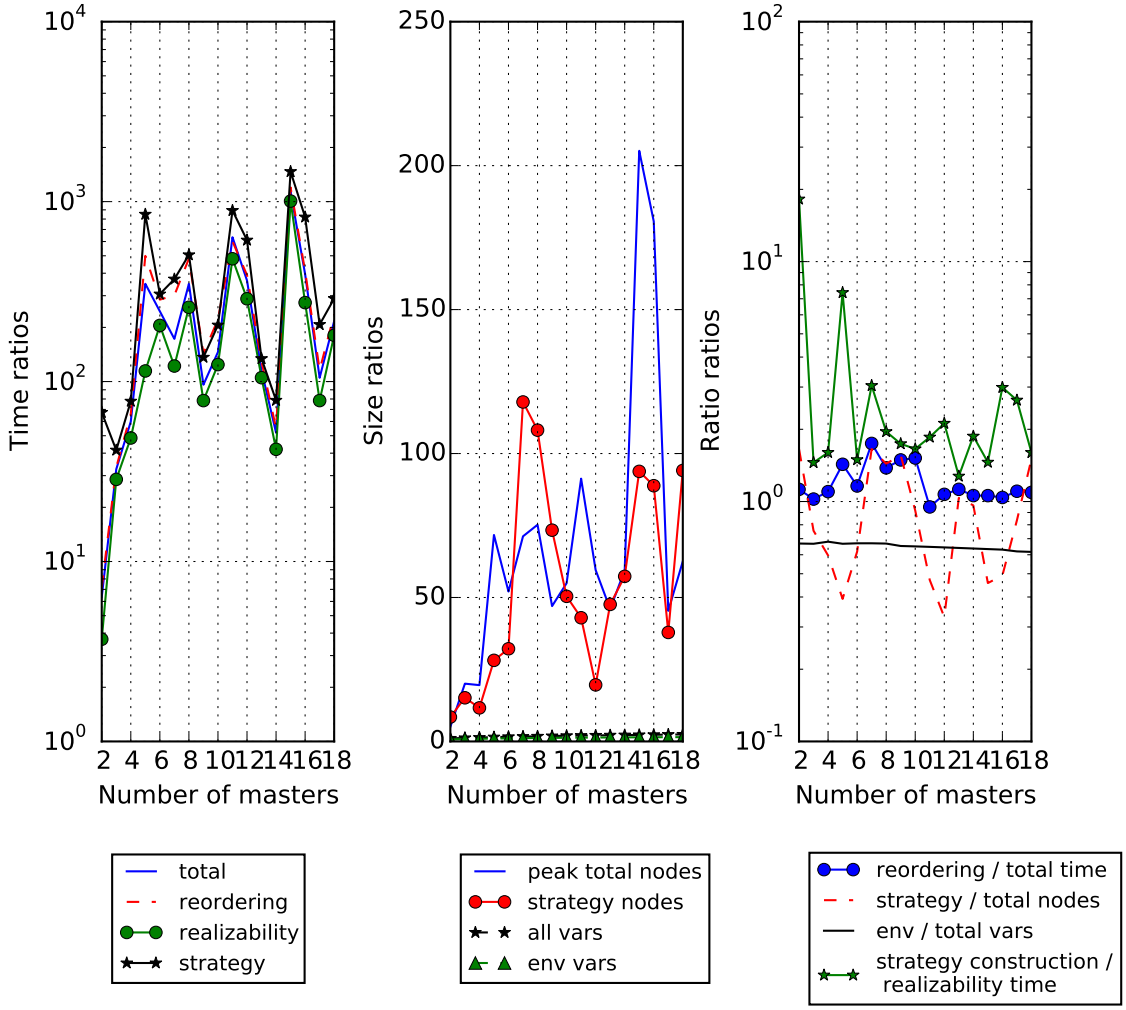


Figure 50: Original with conjunction divided by revised with conjunction (both with reordering).

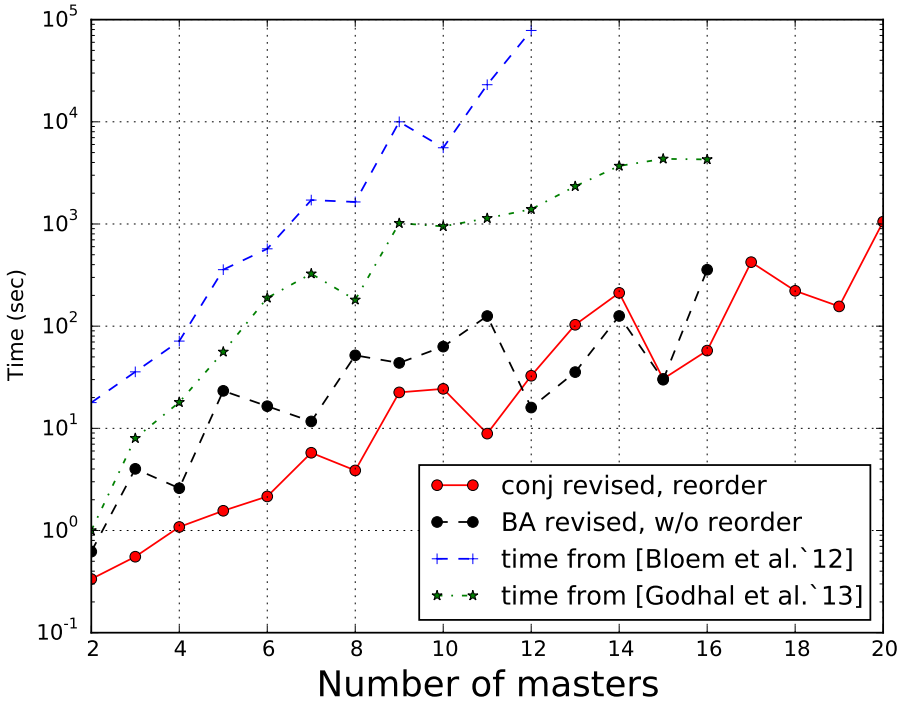


Figure 51: Comparison with [5].

Figure 52: Synthesis time for the revised specification using a BA (w/o strategy reordering), Fig. 9a to Fig. 12c, and conjoining liveness goals (using strategy reordering), Fig. 13a to Fig. 17, compared to results from [5, 7].

4 Relevant work

The specification that we considered was proposed in [3, 1, 5]. The time reported there for 12 masters is in the order of 20 hours, using the synthesizer Anzu [**?**]. Both Anzu (Perl) and Slugs (C++) use Cudd [18] as BDD library.

In [7], a more complete specification was presented for the AHB arbiter, by extending the specification of [5], but without revisions of the form proposed here. In particular, [7] refines the specification, adding more detail, whereas we abstract it, weakening and modifying assumptions. The times reported there are much improved compared to [5], but still scale to above an hour for 16 masters. In contrast, the revised specification with conjunction synthesizes in the order of minutes (with strategy reordering enabled).

In [6], a different algorithm is proposed for constructing the winning strategy from the intermediate sets produced by the fixed point iteration that decides realizability. The resulting strategy is eager to progress in reaching multiple goals, when the opportunity exists, cycling faster through liveness goals. The times reported there are in the order of 6 and 10 hours (for the various strategy construction algorithms) for 15 masters.

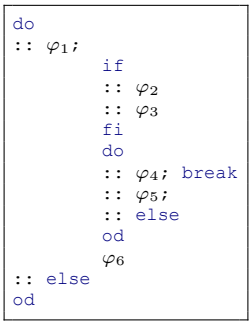


Figure 53: Listing corresponding to formula in text.

A Note on deterministic automata

Note that the (symbolic) automaton is deterministic, because it commits to the choices it makes, i.e., its existential branching is visible to the property (the variables that represent the state are letters in the alphabet). (For automated determinization of LTL properties to either parity or Büchi automata, prior to solving a parity game, see [19]).

For example, an automaton with four nodes can be represented by introducing an auxiliary integer variable $u \in \{0, 1, 2, 3\}$. Suppose that it has the transitions $((u = 0) \rightarrow ((\varphi_1 \wedge \bigcirc(u = 1)) \vee (\neg\varphi_1 \wedge \bigcirc(u = 0)))) \wedge ((u = 1) \rightarrow ((\varphi_2 \wedge \bigcirc(u = 2)) \vee (\varphi_3 \wedge \bigcirc(u = 2)))) \wedge ((u = 2) \rightarrow ((\varphi_4 \wedge \bigcirc(u = 3)) \vee (\varphi_5 \wedge \bigcirc(u = 2)) \vee (\neg\varphi_4 \wedge \neg\varphi_5 \wedge \bigcirc(u = 2)))) \wedge ((u = 3) \rightarrow (\varphi_6 \wedge \bigcirc(u = 0))$. It can be argued that this formula is much less readable and editable than the listing of Fig. 53.

Note that the above automaton can have more than a single enabled transition. It is still a deterministic Buchi automaton, because the program counter u is visible to the environment. This forces the system to commit to its future strategy during GR(1) synthesis, when it selects a valuation of its variables as reaction. In other words, the non-determinism is visible, because u is part of the alphabet and so of the moves that the synthesis algorithm takes in the game. For the above symbolic automaton to be non-deterministic, the variable u must not be part of the alphabet, so not a visible output. This is not the case in GR(1) synthesis.

B Revised AMBA AHB specification

Listing 1: The revised AMBA AHB specification.

```
#define N 2  /* N + 1 masters */
#define SINGLE 0
#define BURST4 1
#define INCR 2


/* variables of masters and slaves */
/* A4: initial condition */
free env bool ready = false;
```

```
10 free env int(0, 2) burst;
11 free env bool request[N + 1] = false;
12 free env bool grantee_lockreq = false;
13 free env bool master_lockreq = false;
14
15 /* arbiter variables */
16 /* G11: sys initial condition */
17 free bool start = true;
18 free bool decide = true;
19 free bool lock = false;
20 free bool lockmemo;
21 free int(0, N) master = 0;
22 free int(0, N) grant;
23
24 /* A2: slaves must progress with receiving data */
25 assume ltl { []<> ready }
26
27 /* A3: dropped, weakening the assumptions */
28
29 /* A1: if current master is granted locked access,
30  * then it must progress by withdrawing the lock request.
31  */
32 assume env proctype withdraw_lock(){
33         progress:
34         do
35         :: lock;
36                 do
37                 :: ! master_lockreq; break
38                 :: true /* wait */
39                 od
40         :: else
41         od
42 }
43
44
45 assert ltl {
46         [](
47                 /* G1: new access starts only when slave is ready
                        */
48                 (start' -> ready)
49                 /* G4,5: current master and lock updated
50                  * only when communicating slave signals
51                  * that it completed receiving data.
52                  */
53                 && (ready -> ((master' == grant) && (lock' <->
                        lockmemo)))
```

```
54                  /* G6: current master and locking may change only
55                   * when an access starts, and remain invariant
                        otherwise
56                   */
57                  && (! start' -> (
58                          (master' == master) &&
59                          (lock' <-> lock)))
60                  /* G7: when deciding, remember if the requestor
61                   * requested also locking.
62                   * when implementing the circuit, route:
63                   * grantee_lockreq = lockreq[grant']
64                   */
65                  && (decide -> (lockmemo' <-> grantee_lockreq))
66                  /* G8: current grantee and locking memo
67                   * remain invariant while not deciding.
68                   */
69                  && (! decide -> (
70                          (grant' == grant) &&
71                          (lockmemo' <-> lockmemo)))
72                  /* G10: only a requestor can become grantee */
73                  && ((grant' == grant) || (grant' == 0) || request[
                        grant'])
74          )
75  }
76
77  /* all properties must hold synchronously */
78  sync{
79
80  /* G9: weak fairness */
81  assert sys proctype fairness(){
82          int(0, N) count;
83          do
84          :: (! request[count] || (master == count));
85                  if
86                  :: (count < N) && (count' == count + 1)
87                  :: (count == N) && (count' == 0);
88                          progress: skip
89                  fi
90          :: else
91          od
92  }
93
94  /* G2: if locked access of unspecified length starts,
95   * then locking shall be withdrawn before starting
96   * another access.
97   */
```

```
98  assert sys proctype maintain_lock(){
99          do
100         :: (lock && start && (burst == INCR));
101                 do
102                 :: (! start && ! master_lockreq); break
103                 :: ! start
104                 od
105         :: else
106         od
107 }
108
109 /* G3: for a BURST4 access,
110  * count the "ready" time steps.
111  */
112 assert sys proctype count_burst(){
113         int(0, 3) count;
114         do
115         :: (start && lock &&
116                 (burst == BURST4) &&
117                 (!ready || (count' == 1)) &&
118                 (ready || (count' == 0)) );
119                 do
120                 :: (! start && ! ready)
121                 :: (! start && ready && (count < 3) &&
122                         (count' == count + 1))
123                 :: (! start && ready && (count >= 3)); break
124                 od
125         :: else
126         od
127 }
128
129 }
```

C Original AMBA AHB specification

The following specification was created by a translator based on the original one from the Anzu website.

Listing 2: The original AMBA AHB specification from [5], for the case of 2 masters.

```
1 free env bit hready, hburst0, hburst1, hbusreq0, hlock0, hbusreq1,
      hlock1;
2 free sys bit hmaster0, hmastlock, start, locked, decide, hgrant0,
    busreq, stateA1_0, stateA1_1, stateG2, stateG3_0, stateG3_1,
    stateG3_2, hgrant1, stateG10_1;
3 assume ltl {
```

```
4 hready == 0 &&
5 hbusreq0 == 0 &&
6 hlock0 == 0 &&
7 hbusreq1 == 0 &&
8 hlock1 == 0 &&
9 hburst0 == 0 &&
10 hburst1 == 0 &&
11
12 [](hlock0 == 1 -> hbusreq0 == 1) &&
13 [](hlock1 == 1 -> hbusreq1 == 1) &&
14
15 [](<>(stateA1_1 == 0)) &&
16 [](<>(hready == 1))
17
18 }
19 assert ltl {
20 hmaster0 == 0 &&
21 hmastlock == 0&&
22 start == 1&&
23 decide == 1&&
24 locked == 0&&
25 hgrant0 == 1&&
26 hgrant1 == 0 &&
27 busreq==0 &&
28 stateA1_0 == 0 &&
29 stateA1_1 == 0 &&
30 stateG2 == 0 &&
31 stateG3_0 == 0 &&
32 stateG3_1 == 0 &&
33 stateG3_2 == 0 &&
34 stateG10_1 == 0 &&
35
36 [](((hmaster0 == 0) -> (hbusreq0 == 0 <-> busreq==0)) &&
37 [](((hmaster0 == 1) -> (hbusreq1 == 0 <-> busreq==0)) &&
38 [](((stateA1_1 == 0) && (stateA1_0 == 0) && ((hmastlock == 0) || (
      hburst0 == 1) || (hburst1 == 1))) ->
39  X((stateA1_1 == 0) && (stateA1_0 == 0))) &&
40 [](((stateA1_1 == 0) && (stateA1_0 == 0) &&  (hmastlock == 1) && (
      hburst0 == 0) && (hburst1 == 0)) ->
41  X((stateA1_1 == 1) && (stateA1_0 == 0))) &&
42 [](((stateA1_1 == 1) && (stateA1_0 == 0) && (busreq == 1)) ->
43  X((stateA1_1 == 1) && (stateA1_0 == 0))) &&
44 [](((stateA1_1 == 1) && (stateA1_0 == 0) && (busreq == 0) && ((
      hmastlock == 0) || (hburst0 == 1) || (hburst1 == 1))) ->
45  X((stateA1_1 == 0) && (stateA1_0 == 0))) &&
46 [](((stateA1_1 == 1) && (stateA1_0 == 0) && (busreq == 0) &&  (
```

```
    hmastlock == 1) && (hburst0 == 0) && (hburst1 == 0)) ->
47  X((stateA1_1 == 0) && (stateA1_0 == 1))) &&
48 [](((stateA1_1 == 0) && (stateA1_0 == 1) && (busreq == 1)) ->
49  X((stateA1_1 == 1) && (stateA1_0 == 0))) &&
50 [](((stateA1_1 == 0) && (stateA1_0 == 1) &&  (hmastlock == 1) && (
    hburst0 == 0) && (hburst1 == 0)) ->
51  X((stateA1_1 == 1) && (stateA1_0 == 0))) &&
52 [](((stateA1_1 == 0) && (stateA1_0 == 1) && (busreq == 0) && ((
    hmastlock == 0) || (hburst0 == 1) || (hburst1 == 1))) ->
53  X((stateA1_1 == 0) && (stateA1_0 == 0))) &&
54 [](((hready == 0) -> X(start == 0)) &&
55 [](((stateG2 == 0) && ((hmastlock == 0) || (start == 0) || (
    hburst0 == 1) || (hburst1 == 1))) -> X(stateG2 == 0)) &&
56 [](((stateG2 == 0) &&  (hmastlock == 1) && (start == 1) && (
    hburst0 == 0) && (hburst1 == 0))  -> X(stateG2 == 1)) &&
57 [](((stateG2 == 1) && (start == 0) && (busreq == 1)) -> X(stateG2
    == 1)) &&
58 [](((stateG2 == 1) && (start == 1)) -> false) &&
59 [](((stateG2 == 1) && (start == 0) && (busreq == 0)) -> X(stateG2
    == 0)) &&
60 [](((stateG3_0 == 0) && (stateG3_1 == 0) && (stateG3_2 == 0) &&
61   ((hmastlock == 0) || (start == 0) || ((hburst0 == 1) || (hburst1
        == 0)))) ->
62   (X(stateG3_0 == 0) && X(stateG3_1 == 0) && X(stateG3_2 == 0)))
       &&
63 [](((stateG3_0 == 0) && (stateG3_1 == 0) && (stateG3_2 == 0) &&
64   ((hmastlock == 1) && (start == 1) && ((hburst0 == 0) && (hburst1
        == 1)) && (hready == 0))) ->
65    (X(stateG3_0 == 1) && X(stateG3_1 == 0) && X(stateG3_2 == 0)))
        &&
66 [](((stateG3_0 == 0) && (stateG3_1 == 0) && (stateG3_2 == 0) &&
67   ((hmastlock == 1) && (start == 1) && ((hburst0 == 0) && (hburst1
        == 1)) && (hready == 1))) ->
68    (X(stateG3_0 == 0) && X(stateG3_1 == 1) && X(stateG3_2 == 0)))
        &&
69
70 [](((stateG3_0 == 1) && (stateG3_1 == 0) && (stateG3_2 == 0) && ((
    start == 0) && (hready == 0))) ->
71    (X(stateG3_0 == 1) && X(stateG3_1 == 0) && X(stateG3_2 == 0)))
        &&
72 [](((stateG3_0 == 1) && (stateG3_1 == 0) && (stateG3_2 == 0) && ((
    start == 0) && (hready == 1))) ->
73    (X(stateG3_0 == 0) && X(stateG3_1 == 1) && X(stateG3_2 == 0)))
        &&
74
75 [](((stateG3_0 == 1) && (stateG3_1 == 0) && (stateG3_2 == 0) && ((
```

```
    start == 1))) -> false) &&
76
77
78 [](((stateG3_0 == 0) && (stateG3_1 == 1) && (stateG3_2 == 0) && ((
    start == 0) && (hready == 0))) ->
79    (X(stateG3_0 == 0) && X(stateG3_1 == 1) && X(stateG3_2 == 0)))
       &&
80 [](((stateG3_0 == 0) && (stateG3_1 == 1) && (stateG3_2 == 0) && ((
    start == 0) && (hready == 1))) ->
81    (X(stateG3_0 == 1) && X(stateG3_1 == 1) && X(stateG3_2 == 0)))
       &&
82 [](((stateG3_0 == 0) && (stateG3_1 == 1) && (stateG3_2 == 0) && ((
    start == 1))) -> false) &&
83
84 [](((stateG3_0 == 1) && (stateG3_1 == 1) && (stateG3_2 == 0) && ((
    start == 0) && (hready == 0))) ->
85    (X(stateG3_0 == 1) && X(stateG3_1 == 1) && X(stateG3_2 == 0)))
       &&
86 [](((stateG3_0 == 1) && (stateG3_1 == 1) && (stateG3_2 == 0) && ((
    start == 0) && (hready == 1))) ->
87    (X(stateG3_0 == 0) && X(stateG3_1 == 0) && X(stateG3_2 == 1)))
       &&
88 [](((stateG3_0 == 1) && (stateG3_1 == 1) && (stateG3_2 == 0) && ((
    start == 1))) -> false) &&
89
90 [](((stateG3_0 == 0) && (stateG3_1 == 0) && (stateG3_2 == 1) && ((
    start == 0) && (hready == 0))) ->
91    (X(stateG3_0 == 0) && X(stateG3_1 == 0) && X(stateG3_2 == 1)))
       &&
92 [](((stateG3_0 == 0) && (stateG3_1 == 0) && (stateG3_2 == 1) && ((
    start == 0) && (hready == 1))) ->
93    (X(stateG3_0 == 0) && X(stateG3_1 == 0) && X(stateG3_2 == 0)))
       &&
94
95 [](((stateG3_0 == 0) && (stateG3_1 == 0) && (stateG3_2 == 1) && ((
    start == 1))) -> false) &&
96 []((hready == 1) -> ((hgrant0 == 1) <-> (X(hmaster0 == 0)))) &&
97 []((hready == 1) -> ((hgrant1 == 1) <-> (X(hmaster0 == 1)))) &&
98 []((hready == 1) -> (locked == 0 <-> X(hmastlock == 0))) &&
99 [](X(start == 0) -> (((hmaster0 == 0)) <-> (X(hmaster0 == 0)))) &&
100 [](X(start == 0) -> (((hmaster0 == 1)) <-> (X(hmaster0 == 1)))) &&
101 [](((X(start == 0))) -> ((hmastlock == 1) <-> X(hmastlock == 1)))
       &&
102 []((decide == 1 && hlock0 == 1 && X(hgrant0 == 1))->X(locked == 1)
    ) &&
103 []((decide == 1 && hlock0 == 0 && X(hgrant0 == 1))->X(locked == 0)
```

```
    ) &&
[]((decide == 1 && hlock1 == 1 && X(hgrant1 == 1))->X(locked == 1)
    ) &&
[]((decide == 1 && hlock1 == 0 && X(hgrant1 == 1))->X(locked == 0)
    ) &&
[]((decide == 0) -> (((hgrant0 == 0)<-> X(hgrant0 == 0)))) &&
[]((decide == 0) -> (((hgrant1 == 0)<-> X(hgrant1 == 0)))) &&
[]((decide == 0)->(locked == 0 <-> X(locked == 0))) &&
[](((stateG10_1 == 0) && (((hgrant1 == 1) ||(hbusreq1 == 1)))) ->
    X(stateG10_1 == 0)) &&
[](((stateG10_1 == 0) && ((hgrant1 == 0) && (hbusreq1 == 0))) -> X
    (stateG10_1 == 1)) &&
[](((stateG10_1 == 1) && ((hgrant1 == 0) && (hbusreq1 == 0)))-> X(
    stateG10_1 == 1)) &&
[](((stateG10_1 == 1) && (((hgrant1 == 1)) && (hbusreq1 == 0))) ->
     false) &&
[](((stateG10_1 == 1) && (hbusreq1 == 1)) -> X(stateG10_1 == 0))
    &&
[]((decide==1  && hbusreq0 == 0 && hbusreq1 == 0) -> X(hgrant0==1)
    ) &&

[](<>(stateG2 == 0)) &&
[](<>((stateG3_0 == 0)  &&  (stateG3_1 == 0)  &&  (stateG3_2 == 0)
    ))
 && [](<>(((hmaster0 == 0))  ||  hbusreq0 == 0))&&
[](<>(((hmaster0 == 1))  ||  hbusreq1 == 0))
}
```

References

[1] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer, "Interactive presentation: Automatic hardware synthesis from specifications: A case study," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '07. San Jose, CA, USA: EDA Consortium, 2007, pp. 1188–1193. [Online]. Available: http://dl.acm.org/citation.cfm?id=1266366.1266622

[2] *AMBA™ Specification*, Rev 2.0 ed., ARM Ltd., 1999.

[3] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer, "Specify, compile, run: Hardware from PSL," *Electronic Notes in Theoretical Computer Science*, vol. 190, no. 4, pp. 3 – 16, 2007, proceedings of the Workshop on Compiler Optimization meets Compiler Verification (COCV 2007). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S157106610700583X

[4] A. Morgenstern, "Symbolic controller synthesis for LTL specifications," Ph.D. dissertation, Computer Science, February 2010. [Online]. Available: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:hbz:386-kluedo-25721

[5] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of Reactive(1) designs," *Journal of Computer and System Sciences*, vol. 78, no. 3, pp. 911 – 938, 2012, in Commemoration of Amir Pnueli. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022000011000869

[6] M. Schlaipfer, G. Hofferek, and R. Bloem, "Generalized reactivity (1) synthesis without a monolithic strategy," in *Hardware and Software: Verification and Testing*. Springer, 2012, pp. 20–34.

[7] Y. Godhal, K. Chatterjee, and T. A. Henzinger, "Synthesis of AMBA AHB from formal specification: a case study," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5-6, pp. 585–601, 2013.

[8] R. Bloem, S. Jacobs, and A. Khalimov, "Parameterized synthesis case study: AMBA AHB (extended version)," 06 2014. [Online]. Available: http://arxiv.org/abs/1406.7608

[9] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of Reactive(1) designs," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Charleston, SC, USA: Springer, January 2006, pp. 364–380.

[10] I. Filippidis, R. M. Murray, and G. J. Holzmann, "Reactive synthesis from PROMELA," California Institute of Technology, Tech. Rep., Feb 2015.

[11] R. Ehlers and V. Raman, "Low-effort specification debugging and analysis," *EPTCS*, vol. 157, pp. 117–133, 07 2014.

[12] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society Press, 1993, pp. 42–47.

[13] S. Panda and F. Somenzi, "Who are the variables in your neighbourhood," in *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on.* IEEE, 1995, pp. 74–77.

[14] M. Byrod, B. Lennartson, A. Vahidi, and K. Akesson, "Efficient reachability analysis on modular discrete-event systems using binary decision diagrams," in *Discrete Event Systems, 2006 8th International Workshop on.* IEEE, 2006, pp. 288–293.

[15] J. Geldenhuys and A. Valmari, "Techniques for smaller intermediary bdds," in *CONCUR 2001—Concurrency Theory.* Springer, 2001, pp. 233–247.

[16] W. Thomas, "Solution of church's problem: A tutorial," *New Perspectives on Games and interaction*, vol. 5, 2008.

[17] A. Browne, E. M. Clarke, S. Jha, D. E. Long, and W. Marrero, "An improved algorithm for the evaluation of fixpoint expressions," *Theoretical Computer Science*, vol. 178, no. 1, pp. 237–255, 1997.

[18] F. Somenzi, "CUDD: CU Decision Diagram package - release 2.5.0," *University of Colorado at Boulder*, 2012. [Online]. Available: http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html

[19] S. Sohail, F. Somenzi, and K. Ravi, "A hybrid algorithm for ltl games," in *Verification, Model Checking, and Abstract Interpretation.* Springer, 2008, pp. 309–323.