Stepwise Development Of Distributed Vertex Coloring Algorithms (Full Report)

Manamiary Bruno Andriamiarina, Dominique Méry

Nancy-Université
Université Henri Poincaré

STEPWISE DEVELOPMENT OF DISTRIBUTED VERTEX COLORING ALGORITHMS

by

Manamiary Bruno ANDRIAMIARINA
Manamiary.Andriamiarina@loria.fr

Dominique MÉRY
Dominique.Mery@loria.fr

Version 1

July 5, 2011

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivations

Software-based systems have a strong impact in the daily life. For instance, systems like televisions, cell phones, credit cards are used for persons, while others systems, like networks, telecommunications, distributed and embedded devices, supercomputers, are used by organisations such as companies, governments, nations... Several countries, especially the advanced ones, rely on systems for the efficiency of domains like economy, health... Since they are needed in daily life, those systems should be reliable, and their specifications and design must be clear, understandable and should follow specific rules and they must avoid faults, failures and if they can not, they should at least be fault-tolerant and fail-safe. Therefore, because of those requirements, *"Formal Verification"* can be usefull to obtain an assurance and guarantee of their correctness with respect to safety and security issues.

1.2 Proof-Based Modelling Of Distributed Algorithms

Formal Verification is provided by techniques and tools like model checking [24] or proof assistants [42] or by combination of abstractions namely *abstract interpretation* [17]; techniques and tools are based on modelling languages which allows one to express the system to model, the property to check and there are environments that integrate these specificities namely Isabelle [42, 16], PVS or Rodin [7]. *System Modelling* is one those techniques, and it is the main technique we have used in our works. *System Modelling* is the action of expressing, analysing and visualising the architecture of a system ([2]). When a model of the system is obtained, properties like safety, liveness can be checked and questions such as *"does the system satisfy its specifications/requirements?..."* can be answered, with the help of methods like *model-checking* ([24]) or the use of *interactive and automatic theorem proving* tools ([8]).

Our works focus on a particular class of systems: systems based on *distributed algorithms*. The verification of those systems is not trivial. A method for checking those systems is the following ([9, 32, 33]): the algorithms used by the systems are redeveloped by targetting a given collection of required properties. During the redevelopment of an algorithm, several models of the system using the algorithm are built. The models are linked together by a relationship called *refinement*: they are produced step-by-step, with the concrete ones adding details and constraints to the abstract ones and preserving their properties and behaviour. Models are developed and proved using the *Event B Method* ([6, 3, 11, 4, 39]) and the tool called *Rodin Platform* ([5]). Other tools like the model-checker *ProB* ([29]) or the visualisation software *Visidia* ([12, 18, 48, 47, 15]) are also used in order to check the behaviour of the models. Consequently, by using this method, we obtain a collection of correct-by-construction algorithms.

We focus here on the probabilistic aspects of distributed algorithms, because they are mainly related to termination, e.g. choice between two delays in the case of communication protocols like IEEE 1394 (FireWire), choice between several colors for vertex coloring algorithms... And since we are interested in adding the study of *probabilistic aspects* of these algorithms, we are planning to use the probabilistic model-checker *PRISM* ([26]) after the last step of the *correct-by-construction* process.

In this report, we present our works about distributed *graph coloring algorithms* (also called *vertex coloring*

algorithms), based on an algorithm developed by Métivier et al ([40]), using the techniques and methods described in this section and the previous ones (see figure 1.1).

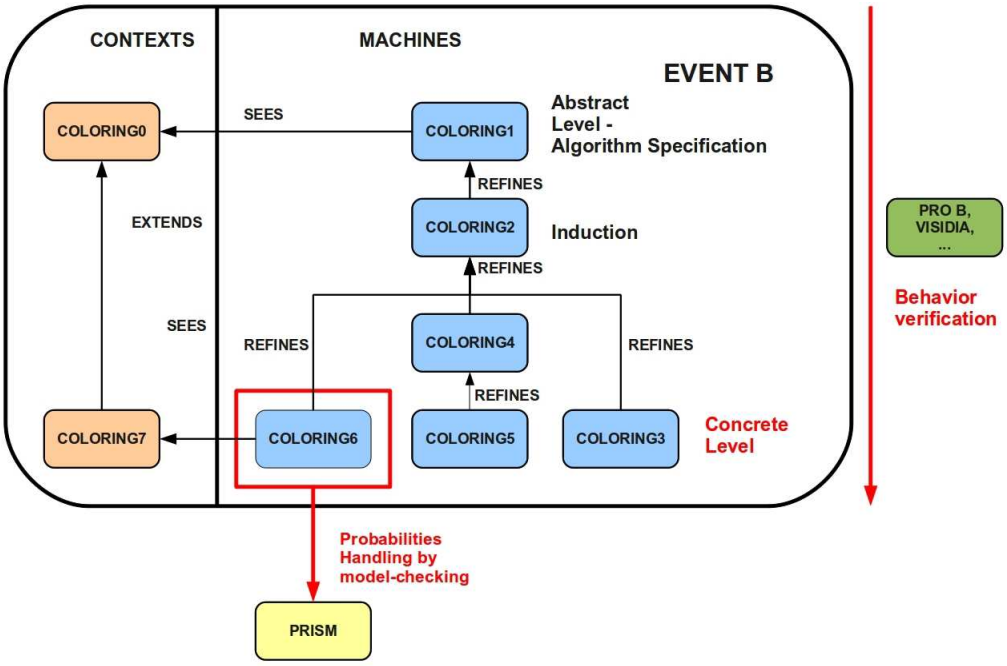


Figure 1.1: Our development methodology

1.3 Related Works

1.3.1 Distributed Systems

a Introduction

A distributed system ([38, 15]) is a system composed of several agents (processes) which apply algorithms written in a programming language and communicate with each other through a communication channel, which can be a network, shared variables, messages...

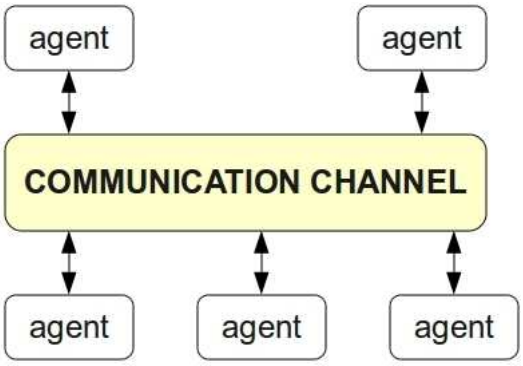


Figure 1.2: A Distributed system

A distributed system is considered as a set of events composed of local events related to the agents and their local computations and global events like the use of the communication channel, messages sending/receiving... and

is often seen as a graph ([48]), where the vertices are the agents/processes and the edges, direct communication links between them.

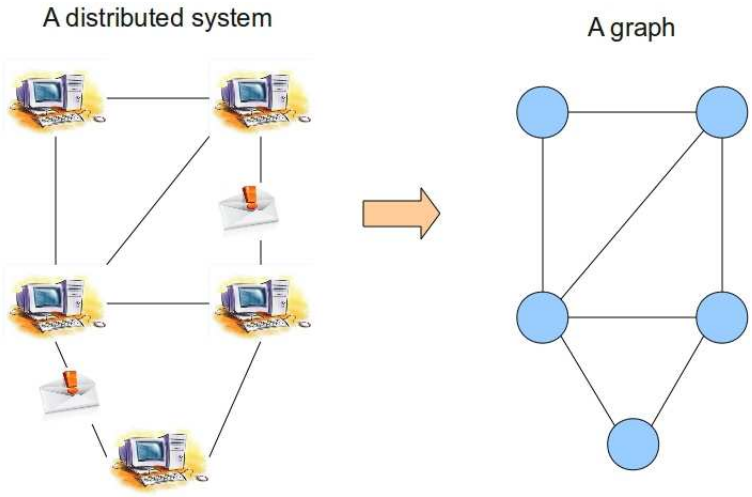


Figure 1.3: Representation of a distributed system by a graph

In a distributed system, the use of resources by agents needs to be coordinated in order to avoid critical situations like deadlocks, starvations, conflicts... Therefore, it is necessary for the events (local and global) to have an ordering. Several researchers, such as Lamport ([28]), have proposed some solutions in order to solve this problem.

b Differences Between Distributed And Centralised Systems

New problems ([38]), which are not present in centralised systems, are encountered in distributed ones, especially:

- Local states vs global states: the system can be blocked (agents blocking each other, deadlocks, starvations...) and its coherence and consistency must always be guaranteed somehow...
- Computations and communications: synchronous or asynchronous?
- Are computations and communications trustworthy?
- Resources sharing: shared resources can lead to competition between agents, and problems such as deadlocks, starvations, conflicts... appear. A policy about how to regulate the use of the resources of the system by its agents must be established (mutual exclusion algorithms...),
- Communication delays: Communications between agents must be reliable and satisfy time constraints,
- Fault management and reliability of the communication channel: communications between agents are often made via untrustworthy devices. Therefore, the system must be fault-tolerant/fail-safe and use several algorithmic solutions in order to guarantee its safe functioning and a good quality of service.

c Examples Of Distributed Systems

Distributed systems often use well-known distributed algorithms ([38]), such as: mutual exclusion, routing, election or distributed calculus... The following systems ([38]) use some of these algorithms:

- Communication protocols,
- IEEE 1394 FireWire: election of a network leader,
- Vertex coloring algorithms,
- Resources sharing: mutual exclusion...

In order to give concrete examples, a few of these systems will be described in details:

Vertex coloring algorithms: The goal of these algorithms ([40, 19, 37, 41, 46, 21, 25, 15]), also called *graph coloring algorithms*, is to assign labels, which are commonly assimilated to colors, to the vertices of a graph. The coloring is done in such a way that no two adjacent vertices of the graph have the same color.

IEEE 1394 FireWire: The IEEE 1394 (FireWire) ([44, 1, 9, 10, 43]) is a protocol created in order to manage communications (data and signals) between interconnected informatic devices and peripherals. The graph/network formed by the interconnected devices must have no cycle. The goal of this protocol is to elect a leader among the device. That leader will be in charge of the management of the network.

Mutual exclusion algorithms: Several agents/processes share and use a common resource. The goal of a mutual exclusion algorithm ([38]) is to avoid the simultaneous use of the common resource. Mutual exclusion algorithms have generally this kind of pattern:

1. Request: a process p asks for the critical section (use of the common resource) and sends a message to all the other processes,
2. Waiting: p waits for all the other processes allowing it to enter the critical section,
3. Critical Section: p enters the critical section and uses the common resource, p will leave it after a finite amount of time,
4. Release: p leaves the critical section and sends a message to all the other processes.

d Distributed Systems And Probabilities

Probabilistic arguments are often involved in termination of some distributed algorithms. A good example showing the importance of probabilities in termination is the communication protocol IEEE 1394 Root Contention Protocol. This example shows us the situation called contention which involves two adjacent nodes asking to each other to become the leader at the same time, therefore introducing a livelock. This situation is solved probabilistically (see figure 1.4): the two nodes have to choose between two delays, a short one and a long one, before asking their neighbour again. If they choose the same, they have to restart the choice again, otherwise the one which has chosen the long delay will become the leader, because he will be asked to become the leader first. We assume that termination is here *almost-certain*: Because of the probabilistic choice, we are sure that one day, the fact that the choices of the two nodes are different will happen.

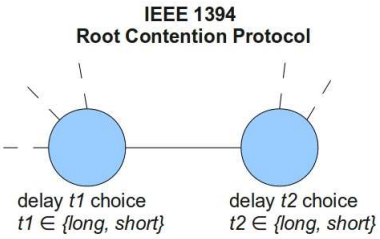


Figure 1.4: Probabilistic choice of a delay before signal emission by a node (IEEE 1394 Root Contention Protocol)

Such kind of situation can also be found during vertex coloring algorithms (see figure 1.5): During the choice of a color by a vertex, the latter can choose the same color as its neighbours or a different one. If it chooses a different one, it means that it has reached its *local* end of the vertex coloring algorithm. Here, the probabilities make us sure that, provided the fact that there are enough colors, so the node and its neighbours can choose different ones, one day, although we do not know the time it will take, the fact that the node and its neighbours choose different colors will happen.

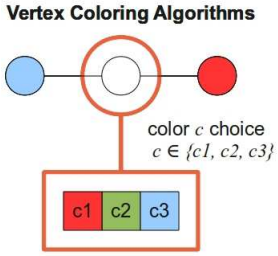


Figure 1.5: Probabilistic choice of a color by a node

1.3.2 Vertex Coloring Algorithms

Symmetry breaking has always been a central problem in distributed systems. Several techniques have been developed in order to achieve it, such as Maximal Independent Set (MIS) algorithms, graph coloring algorithms... In this section, we will focus on graph/vertex coloring algorithms.

a Introduction

A vertex coloring algorithm is a method of graph labelling ([40, 19, 37, 41, 46, 21, 25, 15]): its goal is to assign labels to the vertices of the graph. The labels are often assimilated to colors. Consequently, it is called *graph coloring algorithm*. The coloring/labelling is done in such a manner that no two adjacent vertices of the graph share the same label/color: a proper coloring of a graph $G = \{V, E\}$ (with V the set of the vertices of G and E the set of its edges), using a set of colors $(COLORS \subset \mathbb{N} \mid COLORS = \{1..N\})$, is a function f such as $(f : V \mapsto COLORS \mid f(i) \neq f(j)\ if\ i \leftrightarrow j \in E)$. The minimal N for which f is satisfied is called the *chromatic* number of G and is generally denoted χ or $\chi(G)$.

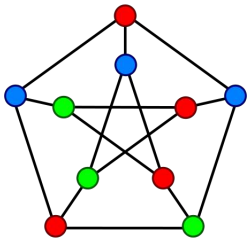


Figure 1.6: A proper vertex coloring of the Petersen graph

Vertex coloring algorithm are generally applied on simple graphs (connected, irreflexive, undirected, unweighted graphs).

b Overview Of Vertex Coloring Algorithms

Several algorithms have been developed in order to color a graph. As described in the work of Duffy, O'Connell and Sapozhnikov ([19]), vertex coloring algorithms can be classified into two categories:

- Algorithms using centralised techniques ([41, 45, 13]): The word *"centralised"* implies that there is at least an *"administrator"* which decides for the graph coloring. It may be a vertex of the graph or an entity which is not a part of it, with complete knowledge of it (structure, edges, number of vertices...). When the *"administrator"* discovers a correct coloring, it sends a message to all the vertices that the algorithm is complete and gives them their colors.
- Algorithms using distributed techniques ([19, 46, 21, 25, 22, 40]): Centralised graph coloring algorithms were developed first. But some problems, which could not be solved using such solutions, have appeared (frequency allocation in IEEE 802.11, in telecommunications [19, 22, 49]). Therefore, new algorithms were

developed. They are still studied nowadays, in order to improve them or to discover new ones. These new algorithms involve all the vertices of the graph which is being colored and those vertices have their own *"intelligence"*: generally, they choose their own colors using probabilities and randomness, know when they have chosen the same color as a neighbour, and when they have a proper color, in which case they remove themselves from the uncolored graph ([19, 40, 46, 21, 25, 22]).

In this work, we focus on developing algorithms using distributed techniques. In fact, there is no or little verification of the safety correctness of the previous algorithms ([19, 46, 21, 25, 22, 40]) which are considering some random numbers for defining the coloring process. The main contribution is yet the analysis of complexity and it can be done later on our models.

c Applications Of Vertex Coloring Algorithms

There are many practical applications of vertex coloring algorithms that include:

- Scheduling ([30]): Graph coloring algorithms can be used to order a set of jobs. Two jobs are considered being adjacent when they can take place at the same time. The goal is to avoid that adjacent jobs happen at the same time.
- Register allocation in compilers ([19, 45]): A program contains a set of variables. Each variable is assigned to a virtual register. The goal is to map virtual registers to physical ones. Two virtual registers are considered being adjacent if there is an *"interference"* between them: They exist at the same time during the execution of the program and they can not be mapped to a same physical register. The purpose of register allocation is to ensure that all the virtual registers are associated with physical ones and that no two adjacent virtual registers share the same physical one.
- Frequency allocation in IEEE 802.11, in telecommunications ([19, 49, 22]): Nowadays, the coexistence of several wireless networks in the same place is common. Interference between these networks must be avoided. Two networks are considered being adjacent if they share the same frequency spectrum. Depending on the standard (IEEE 802.11 a/b/g), there is a certain number of non-overlapping frequencies that can be used in order to avoid interferences. We can see that it is equivalent to a graph coloring problem, with the vertices of the graph being the networks, the colors allowed being the non-overlapping frequencies.

1.4 The Modelling Framework And Tools

1.4.1 Event B

a Introduction

Event B is a formal method for system modelling created by Jean-Raymond Abrial ([6, 11, 4]). It is based on another formal method called *"classical"* B ([3]). An Event B model is charaterized by its invariants and safety properties. The Event B method generates proof obligations in order to show that a model has a correct behaviour, and that its invariants and safety properties are not violated. The proof obligations are generated using the *weakest precondition calculus*. The approach used is generic: the techniques used to model different systems belonging to various domains (such as distributed systems, electrical circuits...) are always the same, with the user only focusing on the proof obligations generated. One of the main aspects of the Event B method is the use of a relation called *"refinement"* between the models: An *"abstract"* model is refined by a *"concrete"* one, if the behaviour of the *"concrete"* model respects the behaviour of the *"abstract"* one. The idea behind *"refinement"* is to begin with a general model containing only the basic behaviour of a studied system, then progressively add more details and constraints to it. Some proof obligations often called *"refinement proofs"* must be demonstrated, in order to prove that, indeed, a concrete model is refining an abstract one. Several tools ([5, 29]) are related to this formal method: we can cite the *"Atelier B"* tool, the *Rodin Platform* and the model-checker *ProB*... During our works, we have mainly used *Rodin Platform* and *ProB*.

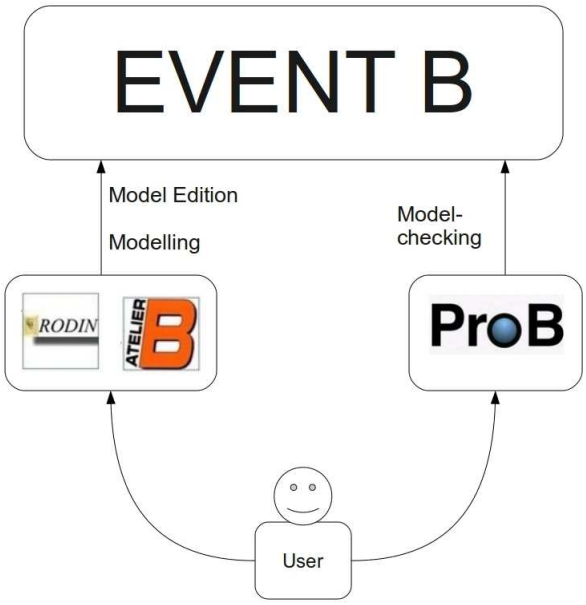


Figure 1.7: Overview of Event B

b Modelling Using The Event B Methodology

Event B models are composed of two elements: a *"context"* and a *"machine"*.

Contexts

The general aspect of a context is the following:

CONTEXT $ctxtN$
EXTENDS $ctxt1,\ ctxt2,\ ...$
SETS $S1,\ S2,\ ...$
CONSTANTS $C1,\ C2,\ ...$
AXIOMS $A1,\ A2,\ ...$
PROPERTIES $P1,\ P2,\ ...$
END

A context contains the static part of a model: it contains the sets (S1, S2, ...), the constants used in the model. It contains also axioms (A1, A2, ...) related to the sets and constants and allowing to prove the properties (P1, P2, ...). A context can extend other contexts (ctxt1, ctxt2, ...) too, and use the sets, constants, axioms, and properties defined in the latters.

Machines

The general aspect of a machine is the following:

MACHINE $machN$
REFINES $machM$
SEES $ctxt1,\ ctxt2,\ ...$
VARIABLES $v1,\ v2,\ ...$
INVARIANTS $I1,\ I2,\ ...$
THEOREMS $T1,\ T2,\ ...$
EVENTS $E1,\ E2,\ ...$
END

A machine contains the dynamic part of a model. It contains the variables (v1, v2, ...), the invariants (I1, I2, ...), the theorems and the events (E1, E2, ...) of the model. The events define the dynamic behaviour of the model:

state changes of the model, variables modifications are done by the events. A machine can refine another one too (machM).

Events

The events of a machine modify its state by changing the values of its variables (denoted by v), which are constrained by its invariants (denoted by $I(v)$). An event is composed of a guard $G(t,\ v)$ and an action $S(t,\ v)$. The symbol t is the set of the local variables of the event. The guard is a condition that must be satisfied, in order to activate the action which will modify the variables.
We have three types of events (called EVT 1, EVT 2, EVT 3):

EVENT EVT 1	EVENT EVT 2	EVENT EVT 3
ANY	WHEN	BEGIN
t	$G(v)$	$S(v)$
WHERE	THEN	END
$G(t, v)$	$S(v)$	
THEN	END	
$S(t, v)$		
END		

$G(t,\ v)$ and $G(v)$ are the possible forms of the guard of an event: $G(t,\ v)$ is used when the event has local variables and $G(v)$ is used when it has none. $S(t,\ v)$ and $S(v)$ are the possible forms of the action of an event: $S(t,\ v)$ is used when the event has local variables and $S(v)$ is used when it has none. The type EVT 2 is used when $t = \varnothing$, i.e. when the event has no local variable. The type EVT 3 is used when the guard is always $true$. An event called INITIALISATION must always be present in a machine: it is the event which defines its initial state by giving the variables v their initial values. The type of this event is EVT 3-like.

The Actions Of An Event And Their Effects

The actions of an event are composed of affectations, which can take one of the following three forms:

AFF 1	AFF 2	AFF 3
$x := E(t,\ v)$	$x :\in E(t,\ v)$	$x : \mid P(t,\ v,\ x')$

x is the variable modified by the affectation, $E(t,\ v)$ is an expression and $P(t,\ v,\ x')$ is a predicate. AFF 1 is a deterministic affectation, it gives x a precise value from a set SET if SET is not empty (SET $\neq\ \varnothing$), while the others are non-deterministic: AFF 2 gives x a random value from SET, if SET is not empty (SET $\neq\ \varnothing$) and AFF 3 gives x a value x' which satisfies the predicate $P(t,\ v,\ x')$. The effects of these affectations can also be described by predicates called *"Before-After"* predicates. These *"Before-After"* predicates define the relations between the state of x before the affectation (denoted by x) and its state after (denoted by x'). In the table below, we list the *"Before-After"* predicates corresponding to each affectation:

Before-After(AFF 1)	*Before-After*(AFF 2)	*Before-After*(AFF 3)
$x' = E(t, v)$	$x' \in E(t, v)$	$P(t, v, x')$

All the affectations contained by an action $S(t,\ v)$ are described by the conjunction of their *Before-after* predicates. In this report, this conjunction will be denoted by a predicate $A(t,\ v,\ x')$. The variables which are not present in the affectations remain unchanged. These unchanged variables will be denoted by y. Therefore, an action can be described by the following *Before-After* predicate:

$$Before - After(S(t,\ v)) \equiv A(t,\ v,\ x') \wedge y' = y$$

In this report, the predicate *Before-After(S(t, v))* will be denoted by *S(t, v, v')*. These predicates are used in the proof obligations of a machine.

Proof Obligations

Proof obligations are generated using the *weakest precondition calculus*. Their main purpose is to demonstrate the correctness of a model: if the model has a correct behaviour, and respects its specifications. In this report, proof obligations are presented under this form: $hypotheses \vdash goals$. Two types of proof obligations are generated: the proof obligations called *Feasibility* and the ones called *Invariant Preservation.*

Feasibility

For each event, *feasibility* must be proved: it must be demonstrated that the *Before-After* predicate $S(t, v, v')$ of an event gives an after-state whenever the guard $G(t,\ v)$ holds. The general aspect of *feasibility* proof obligations is the following:

$$\boxed{I(v), G(v) \vdash (\exists v'.S(t\ , v\ , v'))}$$

By discharging this *feasibility* proof obligation, we show that the guard $G(t,\ v)$ is the enabling condition of the event.

Invariant Preservation

The invariants of a machine must always be verified, even if the state of the machine and/or the values of its variables change. This property can be demonstrated by discharging the following *Invariant Preservation* proof obligation:

$$\boxed{I(v), G(t,\ v), S(t,\ v,\ v') \vdash I(v')}$$

Refinement

One of the main aspects of Event B method is the *refinement*. It is a relationship between two models: if a concrete model refines an abstract one, more details and constraints are added to the abstract model in the concrete one. An event of the abstract machine can be refined by one or more events in the concrete one.
The general form of an event refining another one is the following:

```
EVENT EC
REFINES EVENT EA
  ANY
    I1, I2, ...
  WHERE
    G1, G2, ...
  WITH
    W1, W2, ...
  THEN
    S1, S2, ...
END
```

We can see that the refining event is *EC*, and the refined one is *EA*. Witnesses, which are formulaes assigning values to some abstract parameters of an event and described in the *WITH* part, can also be used.

Formal Definition

A concrete machine can only refine *one* abstract machine. An invariant called *gluing invariant* links the states of the abstract machine to the ones of the concrete machine. This invariant will be denoted $J(v,\ w)$, with v the set of variables of the abstract machine and w the variables of the concrete machine.
When an abstract machine is refined, *each* of its events must be refined by one or several events in the concrete machine.
Let *EA* be an abstract event:

EVENT EA
ANY
t
WHERE
$G(t,\ v)$
THEN
$S(t,\ v)$
END

and *EC* the concrete event refining it:

EVENT EC
ANY
u
WHERE
$G(u,\ w)$
THEN
$T(u,\ w)$
END

EC refines *EA* if the guard of *EC* is stronger than the guard of *EA*, and if the glueing invariant $J(v,\ w)$ establishes a simulation of *EC* by *EA*:

$$I(v), J(v,\ w), H(u\ , w), T(u,\ w,\ w') \vdash (\exists t, v'.G(t,\ v) \wedge S(t,\ v,\ v') \wedge J(v',\ w'))$$

$T(u,\ w,\ w')$ is the *Before-After* predicate of the action $T(u, w)$ of the event *EC*.
New events can be added to the concrete machine: these new events refine a specific event called *SKIP*, which is an event with no action.

c Probabilistic Event-B

Several works were conducted in order to add probabilities to the Event B method especially by Hoang, McIver, Morgan, Hallerstede, Abrial et al ([20, 35, 36, 23, 31, 34]). They propose an extension of Event B which incorporates probabilities, via the introduction of a probabilistic-choice substitution, with its associated semantic and logic based on real numbers, which are necessary to express probabilities. This extension introduces new ways of expressing invariants, events and affectations:

- Invariants: $p < I >$. The interpretation of such an expression is : probability p if I holds, and probability 0 otherwise,
- Events: Different ways to specify probabilistic events are introduced too.
  1. E1 $\equiv SELECT\ p\ THEN\ b\ END$
     E2 $\equiv SELECT\ 1-p\ THEN\ c\ END$

     These expressions show how complementary events can be defined : action b may occur with a probability p and another one c (complementary of b) , can occur with probability $1-p$.
  2. E12a $\equiv$
     $PCHOICE\ p\ THEN\ b$
     $ELSE\ c$
     END
  3. E12b $\equiv BEGIN\ b\ {}_p\bigoplus\ c\ END$
- Affectations: a probabilistic affectation is also introduced.
  - $x \bigoplus |P(t, v, x')$: A new value satisfying the predicate P is assigned to x. This new value is chosen among the set of the values satisfying P, according to a probabilistic distribution over the elements of this set. It means that each values which satisfies P has a probability p to be chosen attached to it, with respect to the fact that the sum of all the probabilities p is 1 and p is neither equal to 0, nor 1, but $0 < p < 1$.

1.4.2 PRISM

One of the main goals of our work is to take probabilities into account during the development of distributed systems. In order to achieve that purpose, we focused our interest on probabilistic formal tools, like the model-checker PRISM ([26]).

a Introduction

PRISM ([26]) is a formal verification tool developed by the University of Birmingham. It is a model-checker whose main purpose is the modelling and the analysis of systems which have features related to random behaviour and/or probabilistic statements, such as distributed systems, biological processes...

b Architecture Of PRISM

PRISM supports three differents probabilistic models:

- DTMC (Discrete Time Markov Chains): This model is a variant of Markov chains. Its characterics are the following: the transitions between the possible states of a system being modelled occur in discrete time-steps, and the probabilities of making a transition are given by discrete probability distribution,
- MDP (Markov Decision Processes): This model is an extension of DTMC. It combines the probabilistic behaviour of DTMC with non-determinism: the choice of the successor of a state of a system being modelled is a non-deterministic choice between several discrete probability distributions over its possible successors,
- CTMC (Continuous Time Markov Chains): In this model, the time is considered being continuous, transitions between the possible states of a system being modelled can occur at any time instant, and its probabilistic behaviour is continuous.

Systems are described using the *PRISM Language*. The descriptions are specified as being DTMC, CTMC or MDP. The resulting models are taken as inputs by the model-checker. The specifications and properties we want to verify are written using two logics derived of the CTL logic (Computation Tree Logic - which is a branching-time logic, meaning that its model of time is a tree-like structure in which a state can have several and different successors):

- PCTL: it is a probabilistic extension of CTL,
- CSL: this logic is based on CTL and PCTL and is used for specifications and properties on models which are specified as being CMTC.

The model-checker computes the set of reachable states, and verify by using BDD (Binary Decision Diagrams) techniques, if there are states which satisfy the specifications and properties written in PCTL or CSL.

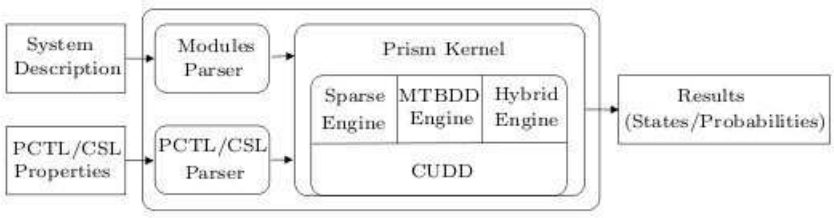


Figure 1.8: Architecture of PRISM

c Modelling With PRISM

In this section, we present a brief description of the *PRISM Language* and the way properties and specifications are written.

Data Types

There are two data types in the *PRISM Language*: *integer* and *boolean*. There are no complex data structures like arrays, lists, records, sets, maps,... in the language.

PRISM Models

PRISM Models can be written that way:

- First, we have to specify the probabilistic behaviour of our model: it can be probabilistic (*dtmc*), non-determistic and probabilistic (*mdp*) or stochastic (*ctmc*)
- Then, we have to specify the model's dynamic behaviour:
  - A PRISM model is composed of several modules, which have the following pattern:

    ```
    module <module name>
            ...
    endmodule
    ```

    A module contain local variables describing its local state of and commands describing its behaviour.
  - The elements of a command are *guards* and *updates*: the *guards* are predicates over the local and global variables of a module, and if they are satisfied, the *updates*, which are the transitions a module can make, occur.
    A command has two forms:
    - a form where there are several updates, and where the choice of the update, which will occur, depends on probabilities:

      ```
      [] guard 1 & guard 2 & ... & guard N
          ->  probability 1 : (update 1) +
              ... +
              probability N : (update N);
      ```

    - a form where there is only one update that does not depend on probabilities:

      ```
      [] guard 1 & guard 2 & ... & guard N -> update;
      ```

      The update will always occur whenever the guard holds.
- Example of a PRISM model:

```
mdp
module M1
    x : [0..2] init 0;
    [] x=0 -> 0.8:(x'=0) + 0.2:(x'=1);
    [] x=1 & y!=2 -> (x'=2);
    [] x=2 -> 0.5:(x'=2) + 0.5:(x'=0);
endmodule

module M2
    y : [0..2] init 0;
    [] y=0 -> 0.8:(y'=0) + 0.2:(y'=1);
    [] y=1 & x!=2 -> (y'=2);
    [] y=2 -> 0.5:(y'=2) + 0.5:(y'=0);
endmodule
```

Verifying Properties With PRISM

We present in this sections some examples of PRISM properties:

1. `P<0.01 [ X y=1 ]`

   This property means: *"The probability that y is equal to 1 in the next state is less than 0.01"*.

2. `P>0.5 [ z<2 U z=2 ]`

   This property means: *"The probability that z is inferior to 2, until z becomes equal to 2, is more than 0.5"*.

3. `P<0.1 [ F z>2 ]`

   This property means: *"The probability that z becomes one day superior to 2 is less than 0.1"*.

4. `P>=0.99 [ G z<10 ]`

   This property means: *"The probability that z is always inferior to 10 is greater or equal to 0.99"*.

5. `P=? [ G z<10 ]`

   This property means: *"We want PRISM to determine the probability that the event 'z is always inferior to 10' occurs"*.

We can notice that the specification of PRISM properties requires the use of temporal logic combined with probabilities.

1.4.3 ViSiDiA

a Introduction

As we have already said in the previous sections, distributed systems are taking more and more place in our society. These systems allow us to improve domains like telecommunications, networks, real-time process control, parallel computation... But their development is a difficult and complicated process: in addition to the classical problems of a system, new problems have appeared such as distribution, communication, competition problems, resources conflicts... and safety properties related to them are most of the time hard to prove... In order to ease the development of distributed systems and help researchers and developers to gain insight into the functioning of those systems, a visualisation and simulation software, dedicated to distributed algorithms, called *ViSiDiA* has been developed by Mohamed Mosbah, Yves Métivier, Pierre Castéran, Akka Zemmari, Cédric Aguerre et al ([14, 12, 18, 48, 47]).

b ViSiDiA Concepts And Architecture

ViSiDiA is a software dedicated to distributed systems. It allows the user to visualise and simulate the behaviour of said systems. Since distributed systems can be represented by graphs, its approach is based on rewriting systems, especially graph relabelling: labels are attached to the vertices and edges of a graph and these labels are modified until no further transformation can be done. It is also based on local computations: the vertices do their computations alone, and sometimes, communicate with neighbours via messages passing. Randomisation is used by ViSiDiA to simulate the non-determinism aspect present in most of distributed systems.
As described in the figure below (figure 1.9), the software can be divided into three layers, which are:

1. a Graphical User Interface (GUI): this component allows the user to draw and edit graphs, to visualise the simulation of a distributed system,

2. a Simulator: this component is the link between the GUI and the algorithms, it simulates the execution of the algorithms by the vertices of the graph representing the studied distributed system,

3. algorithms: these are a set of threads which represent each node. The algorithms are local: they are applied by a vertex/process/agent. They are written in *Java* using some APIs provided by ViSiDiA and can be synchronous or asynchronous ([18]).

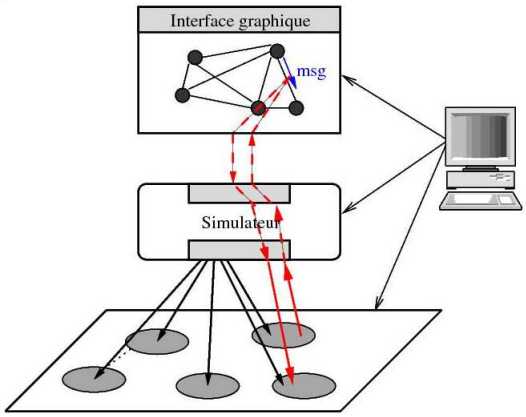


Figure 1.9: Architecture of ViSiDiA

c Related Works

Some works focus on identifying distributed algorithms using local computations, rewriting distributed systems as graph relabelling and studying properties like termination detection... such as the thesis of Affif Selami ([47]), or the report written by Métivier et al ([15]) .

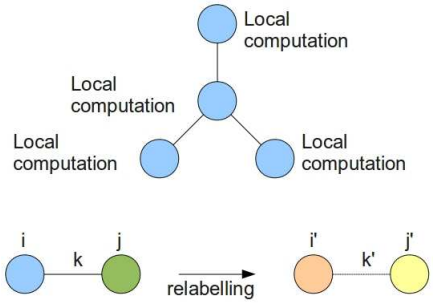


Figure 1.10: Examples of local computations and graph relabelling ([47, 15])

Another topic was also the subject of an in-depth studies: the combination of ViSiDiA with another formal method, such as Event-B. This domain was studied by Mohamed Tounsi, Ahmed Hadj-Kacem, Mohamed Mosbah and Dominique Méry et al. They give in [14] a general model for developing and proving distributed algorithms using Event B and a perspective of their work is the derivation of a ViSiDiA-like model from concrete and local Event-B model ([27]).

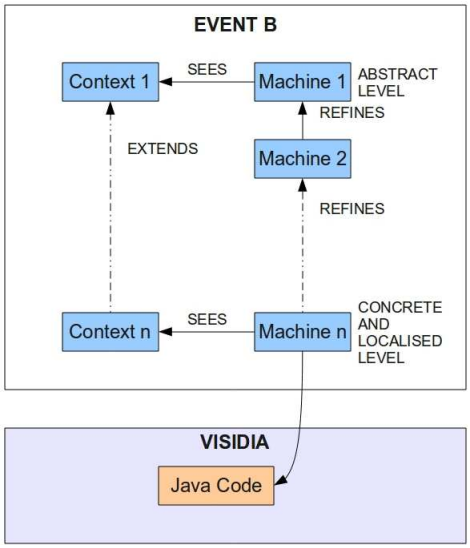


Figure 1.11: Derivation of a ViSiDiA model from an Event B model ([14])

Chapter 2

Stepwise Development Of Distributed Vertex Coloring Algorithms

2.1 Overview

The figure 2.1 shows the methodology we used to develop the vertex coloring algorithms:

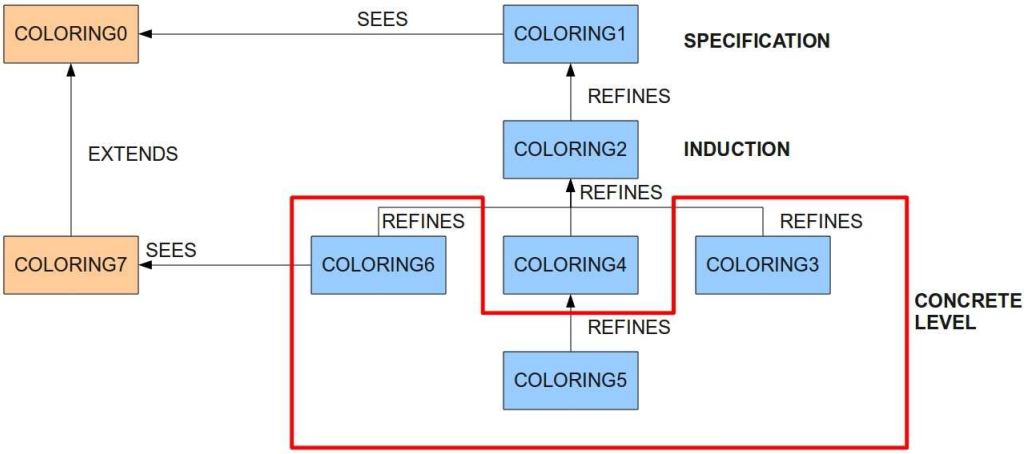


Figure 2.1: Overview of the development in Event B

We begin by an abstract specification of the coloring problem (**COLORING1**), which is refined by a machine defining an inductive coloring fuction (**COLORING2**) and finally, after some refinement steps, we end with three concrete machines, which define three different vertex coloring algorithms (**COLORING3**, **COLORING5**, **COLORING6**). Examples showing the behaviour of these three graph coloring algorithm can be found in the appendices of this report (Appendices A, B and C).

2.2 The Coloring Problem In Event B

We assume that a graph *GRAPH* is given over a set of vertices *NODES*. Then, we define a set of colors *COLORS*, whose components will be the colors chosen by the vertices during the execution of one of the graph coloring algorithms.

SETS
NODES
COLORS

CONSTANTS
$GRAPH$

We specify some properties about these sets and constants:

$axm1 : NODES \neq \varnothing$
$axm2 : GRAPH \in NODES \leftrightarrow NODES$
$axm3 : GRAPH \neq \varnothing$
$axm4 : COLORS \neq \varnothing$

- the axiom *axm1* expresses that the set of vertices is not empty,
- the axiom *axm2* expresses that the graph *GRAPH* is a set containing vertices linked between them by edges,
- the axiom *axm3* expresses that the *GRAPH* is not empty,
- the axiom *axm4* expresses that the set of colors *COLORS* is not empty,

$axm5 : \forall n \cdot n \in NODES \Rightarrow n \in dom(GRAPH)$
Nodes are not isolated
$axm6 : NODES \lhd id \cap GRAPH = \varnothing$
GRAPH is irreflexive
$axm7 : GRAPH = GRAPH^{-1}$
GRAPH is symmetric
$axm8 : \forall s \cdot s \subseteq NODES \wedge s \neq \varnothing \wedge GRAPH[s] \subseteq s \Rightarrow NODES \subseteq s$
GRAPH is connected

- the previous axioms express that the graph *GRAPH* is a simple graph:
  - the axiom *axm5* expresses that all the vertices belong to the graph *GRAPH*, that they are not isolated,
  - the axiom *axm6* expresses that the graph *GRAPH* is irreflexive: the neighbour of a vertex in *GRAPH* must be another vertex, not itself,
  - the axiom *axm7* expresses that the graph *GRAPH* is symmetric,
  - the axiom *axm8* expresses that the graph *GRAPH* is connected,

$$axm9 : \forall g \cdot \left(\begin{array}{l} g \in NODES \leftrightarrow NODES \wedge g \neq \varnothing \\ \Rightarrow \\ \exists colored\_nodes \cdot \left(\begin{array}{l} \forall n1 \cdot \forall n2 \cdot \forall c1 \cdot \forall c2 \cdot \\ \left(\begin{array}{l} n1 \neq n2 \wedge \\ n1 \mapsto n2 \in g \wedge \\ n1 \mapsto c1 \in colored\_nodes \wedge \\ n2 \mapsto c2 \in colored\_nodes \end{array} \right) \\ \Rightarrow \\ c1 \neq c2 \end{array} \right) \end{array} \right)$$

- using classical results of the graph theory, we assume that there exists a vertex coloring function such that no two adjacent vertices of the graph share the same color: the axiom *axm9* expresses this fact. This axiom helps us to prove the *feasibility* of the computation, by stating that there exists a solution and hence one should refine the specification into an algorithmic process leading to the effective computation. Hence, the axiom *axm9* can be defined as a theorem and proved using the proof assistant but we simply reuse the expression of this property and let the proof in another process. It is mandatory for deriving the feasibility of the event expressing the specification of the graph coloring problem.

Now, we specify the graph coloring problem (see figure 2.2):

- the input of the specification is a graph whose vertices are not colored yet,
- the output is the same graph with its vertices colored in such a way that no two vertices share the same color.

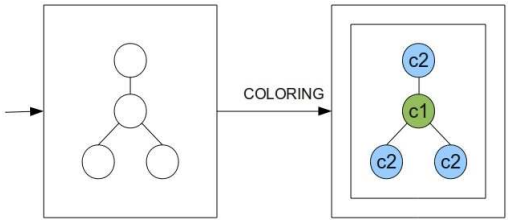


Figure 2.2: Coloring a graph in one shot

- we define a variable *colored_nodes_m0* which contains the assignment of colors to the vertices of *GRAPH* and which satisfies the following invariant *inv1*:

$$inv1 : colored\_nodes\_m0 \in NODES \nrightarrow COLORS$$

- its initial value is the empty set ($\varnothing$):

INITIALISATION
$act1\ colored\_nodes\_m0 := \varnothing$

- the expression of the coloring process is simply stated by the following event COLORING1:

EVENT COLORING1

$$act1 \ : \quad colored\_nodes\_m0 : | \left(\begin{array}{l} (colored\_nodes\_m0' \in NODES \rightarrow COLORS \\ \wedge \\ \left(\begin{array}{l} \forall n1 \cdot \forall n2 \cdot \forall c1 \cdot \forall c2 \cdot \\ \left(\begin{array}{l} n1 \neq n2 \wedge \\ n1 \mapsto n2 \in GRAPH \wedge \\ n1 \mapsto c1 \in colored\_nodes\_m0' \wedge \\ n2 \mapsto c2 \in colored\_nodes\_m0' \end{array} \right) \\ \Rightarrow \\ c1 \neq c2 \end{array} \right) \end{array} \right)$$

The event is feasible, since the axiom *axm9* ensures that there exists one coloring function, such that no two vertices of the graph share the same color.

The following figure (figure 2.3) sums up the behaviour of the machine containing the specification of the graph coloring problem:

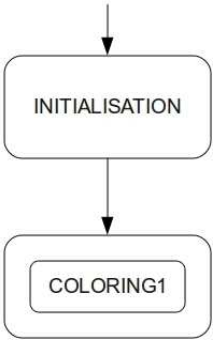


Figure 2.3: First Machine behaviour

Now, the question is to compute the coloring function and we have to find an inductive property simulating the computation of the function.

2.3 Computing The Coloring Function

In the machine **COLORING2**, which refines the previous machine **COLORING1**, we define an inductive coloring function, which assigns the vertices their colors, which are different from the ones given to their neighbours (see figure 2.4).

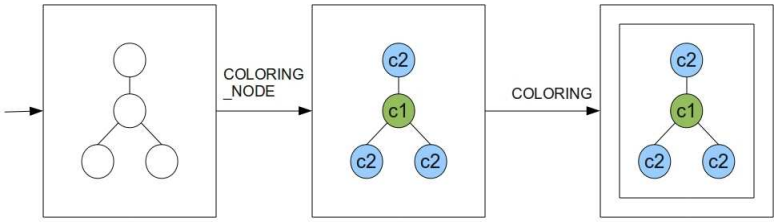


Figure 2.4: The coloring function

We define two new variables: *colored_nodes_m1*, which describes the current state of the graph and contains the vertices which have a color and *has_color_m1*, which is the set of vertices already colored.

$$\begin{array}{l} inv1 : colored\_nodes\_m1 \in NODES \nrightarrow COLORS \\ inv2 : has\_color\_m1 \subseteq NODES \end{array}$$

These variables and the events defined in **COLORING2** have to satisfy the following invariants:

$$\begin{array}{l} inv3 : \left(\begin{array}{l} \forall n1 \cdot \forall n2 \cdot \\ \left(\begin{array}{l} n1 \mapsto n2 \in GRAPH \wedge \\ n1 \in dom(colored\_nodes\_m1) \wedge \\ n2 \in dom(colored\_nodes\_m1) \end{array} \right) \\ \Rightarrow \\ colored\_nodes\_m1(n1) \neq colored\_nodes\_m1(n2) \end{array} \right) \\ inv4 : \forall n \cdot n \notin has\_color\_m1 \Leftrightarrow n \notin dom(colored\_nodes\_m1) \\ inv5 : \forall n \cdot n \in has\_color\_m1 \Leftrightarrow n \in dom(colored\_nodes\_m1) \\ inv6 : \left(\begin{array}{l} dom(GRAPH) = has\_color\_m1 \\ \Rightarrow \\ \left(\begin{array}{l} \forall node \cdot node \in dom(GRAPH) \\ \Rightarrow \\ node \in dom(colored\_nodes\_m1) \end{array} \right) \end{array} \right) \\ inv7 : dom(colored\_nodes\_m1) = has\_color\_m1 \end{array}$$

- the invariant *inv3* expresses that if a vertex $n1$ is colored and another vertex $n2$ is colored and these vertices are neighbours, then the color of $n1$ is different from the color of $n2$,
- the invariant *inv4*, *inv5*, *inv7* express that the domain of *colored_nodes_m1*, which contains the vertices colored, is the same as the set of the vertices (*has_color_m1*) which are already colored,
- the invariant *inv6* expresses that if all the vertices of the graph are colored, then they all belong to the set containing the vertices which have chosen a color (*colored_nodes_m1*).

The initial values of the variables are the empty set ($\varnothing$).

$$\begin{array}{l} \text{INITIALISATION} \\ \quad act1 : colored\_nodes\_m0 := \varnothing \\ \quad act2 : colored\_nodes\_m1 := \varnothing \\ \quad act3 : has\_color\_m1 := \varnothing \end{array}$$

A new event COLORING_NODE2 is introduced: it contains the coloring function. It simulates the local choice of a color by a vertex. When a vertex is not colored yet, a color different from the ones chosen by its neighbours, which have already been colored, is assigned to it.

EVENT COLORING_NODE2
  ANY
    $node$
    $color$
  WHERE
    $grd1$: $node \in NODES$
    $grd2$: $color \in COLORS$
    $grd3$: $node \in dom(GRAPH)$
    $grd4$: $node \notin has\_color\_m1$
    $grd5$: $color \notin colored\_nodes\_m1[GRAPH[\{node\}]]$
  THEN
    $act1$: $colored\_nodes\_m1(node) := color$
    $act2$: $has\_color\_m1 := has\_color\_m1 \cup \{node\}$

The second event controls the completion of the computation.

EVENT COLORING2
REFINES EVENT COLORING1
  WHEN
    $grd1$: $dom(GRAPH) = has\_color\_m1$
  THEN
    $act1$: $colored\_nodes\_m0 := colored\_nodes\_m1$

The following figure (figure 2.5) sums up the behaviour of the first refinement:

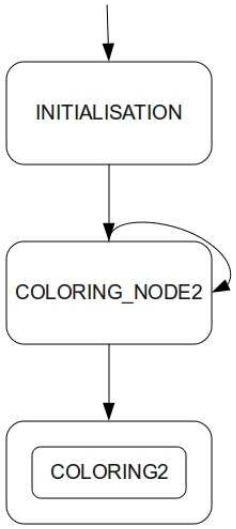


Figure 2.5: First Refinement behaviour

We summarize the current description of our model by stating that a vertex chooses the right color among its neighbours which are supposed to be in a *"ball"* centered on the vertex. This model does not take into account the notion of ill-chosen colors and multiple tentatives for choosing a color. Next section will solve this question.

2.4 Progressing By Errors

The algorithms defined in the machines **COLORING3**, **COLORING5**, **COLORING6** introduce the notion that a vertex might do multiple color choices before finding the right one (see figure 2.6) .

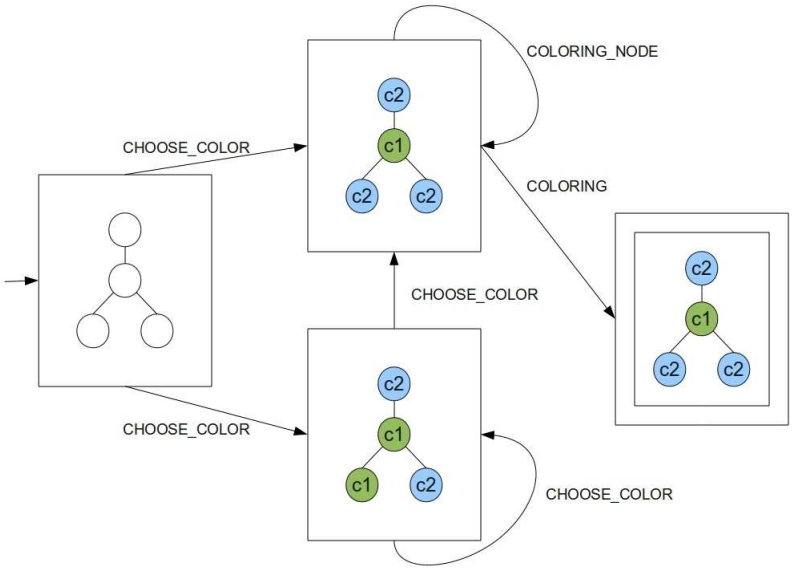


Figure 2.6: Progressing by errors

The principles of theses algorithms are the following: A vertex has to choose a color until it obtains a right one, that is, a color different from the ones its neighbours have chosen. When the vertex has chosen a right color, it stops executing its the algorithms. The end of the algorithm is reached when every vertex in the graph has stopped executing it, that is, when the color chosen by each vertex of the graph is different from the ones assigned to its neighbours.

2.4.1 Synchronous Coloring Algorithm

The concrete machine **COLORING3** describes a first vertex coloring algorithm, which presents the following characteristics:

- each vertex v of a graph G, to which the algorithm is applied, is linked to a list L_v, which contains its active neighbours, i.e. the neighbours not already colored or having the same color as it,
- during the execution of the algorithm, v updates L_v, by removing from it its neighbours which have colors different from the one it has.

This algorithm can be divided into three steps:

1. in a first round, all the vertices, which have no color or are not correctly colored, randomly choose colors which are not already assigned to some of their neighbours, which are not parts of their active neighbours lists anymore,
2. then in a second round, all the vertices which have chosen colors, update the graph by deleting the edges between them and their neighbours which have chosen different colors,
3. and finally the vertices verify if they are still linked to some neighbours; if it is the case, they restart choosing colors otherwise they stop executing the algorithm.

We define five new variables in **COLORING3**:

$$
\begin{array}{l}
inv1 \ : active\_graph\_m2 \subseteq GRAPH \\
inv2 \ : colored\_nodes\_m2 \in NODES \nrightarrow COLORS \\
inv3 \ : has\_chosen\_color\_m2 \subseteq NODES \\
inv4 \ : relations\_m2 \subseteq GRAPH \\
inv5 \ : active\_nodes\_m2 \subseteq NODES
\end{array}
$$

- *active_graph_m2* is a set containing the edges between the vertices of the graph $GRAPH$ which are not already colored or share the same colors (i.e a vertex a which is adjacent to a vertex b and which has the same color as b is an active neighbour of b),
- *colored_nodes_m2* is contains the vertices of the graph as keys and the colors assigned to these vertices as values,
- *has_chosen_color_m2* contains the vertices which have chosen a color during the execution of the algorithm,
- *relations_m2* contains the edges of the graph which have to be observed during the execution of the algorithm,
- *active_nodes_m2* is a set which contains the vertices of the graph with active neighbours.

These variables and the events defined in **COLORING3** have to satisfy the following invariants:

$$
\begin{array}{l}
inv6 \ : \forall n1 \cdot \forall n2 \cdot \\
\quad \left(\begin{array}{l} n1 \mapsto n2 \in GRAPH \land n2 \mapsto n1 \in GRAPH \land \\ n1 \mapsto n2 \notin active\_graph\_m2 \land n2 \mapsto n1 \notin active\_graph\_m2 \land \\ n1 \in dom(colored\_nodes\_m2) \land \\ n2 \in dom(colored\_nodes\_m2) \end{array} \right) \\
\quad \Rightarrow \\
\quad colored\_nodes\_m2(n1) \neq colored\_nodes\_m2(n2) \\
\\
inv7 \ : active\_graph\_m2 = \varnothing \\
\quad \Rightarrow \\
\quad \left(\begin{array}{l} \forall node \cdot \forall color \cdot node \mapsto color \in colored\_nodes\_m2 \\ \Rightarrow \\ \neg color \in ran(ran(\{node\} \lhd GRAPH) \lhd colored\_nodes\_m1)) \end{array} \right) \\
\\
inv8 \ : active\_graph\_m2 \neq \varnothing \quad \Rightarrow \\
\quad (\forall n \cdot n \in NODES \Rightarrow n \notin dom(colored\_nodes\_m1)) \\
\\
inv9 \ : colored\_nodes\_m1 \subseteq colored\_nodes\_m2
\end{array}
$$

- the invariant *inv6* expresses that if two vertices of *GRAPH* are adjacent in *GRAPH* but not linked in *active_graph_m2* and have chosen colors, then the color chosen by the first vertex is different from the one chosen by the second,
- the invariant *inv7* expresses that if the vertices in *GRAPH* have no active neighbours anymore, then the color of a vertex in *colored_nodes_m2* is different from the ones its neighbours have definitely chosen in *colored_nodes_m1*,
- the invariant *inv8* expresses that if there are still vertices with active neighbours, then no vertex has a definitive color in *colored_nodes_m1*,
- the invariant *inv9* expresses that the process in **COLORING3** is simulating the previous process in **COLORING2**; we indicate a possible error in the choice of a color and we state that the new variable *colored_nodes_m2* is correct with respect to *colored_nodes_m1*: *colored_nodes_m1* $\subseteq$ *colored_nodes_m2*.

The initial values of the variables are the empty set ($\varnothing$), except for *active_graph_m2*, *relations_m2* which is initialised with *GRAPH* and *active_nodes_m2* which is initialised with *dom(GRAPH)*, because all the vertices have active neighbours in the initial state.

$$
\begin{array}{l}
\text{INITIALISATION} \\
\quad act1 \ : colored\_nodes\_m0 := \varnothing \\
\quad act2 \ : colored\_nodes\_m1 := \varnothing \\
\quad act3 \ : has\_color\_m1 := \varnothing \\
\quad act4 \ : active\_graph\_m2 := GRAPH \\
\quad act5 \ : colored\_nodes\_m2 := \varnothing \\
\quad act6 \ : has\_chosen\_color\_m2 := \varnothing \\
\quad act7 \ : relations\_m2 := GRAPH \\
\quad act8 \ : active\_nodes\_m2 := dom(GRAPH)
\end{array}
$$

The event CHOOSE_COLOR describes the choice of a color by a node: the node randomly chooses a color not already chosen by its non-active neighbours. A non-active neighbour is a neighbour of the vertex which has already chosen a color different of the vertex of the node in a previous step of the algorithm. However, the vertex may choose the same color than the others active vertices. When all the vertices, which have to chose a color, have completed the phase modelled by this event, the event GRAPH_REDUCTION is enabled.

EVENT CHOOSE_COLOR
ANY
$node$
$color$
WHERE
$grd1$: $node \in NODES$
$grd2$: $color \in COLORS$
$grd3$: $node \in dom(active\_graph\_m2)$
$grd4$: $node \notin has\_chosen\_color\_m2$
$grd5$: $color \notin ran(ran(\{node\} \lhd (GRAPH \setminus active\_graph\_m2)) \lhd colored\_nodes\_m2)$
$grd6$: $has\_chosen\_color\_m2 \subset active\_nodes\_m2$
THEN
$act1$: $colored\_nodes\_m2(node) := color$
$act2$: $has\_chosen\_color\_m2 := has\_chosen\_color\_m2 \cup \{node\}$

The event GRAPH_REDUCTION removes from the list of active neighbours of a vertex the neighbours which have chosen a different color from the one it has chosen.

EVENT GRAPH_REDUCTION
ANY
$node$
WHERE
$grd1$: $node \in NODES$
$grd2$: $node \in dom(relations\_m2)$
$grd3$: $node \in has\_chosen\_color\_m2$
$grd4$: $node \in dom(colored\_nodes\_m2)$
$grd5$: $has\_chosen\_color\_m2 = active\_nodes\_m2$
THEN
$act1$: $active\_graph\_m2 := \left(active\_graph\_m2 \setminus \left(\begin{array}{l} (\{node\} \lhd active\_graph\_m2) \\ \rhd dom(colored\_nodes\_m2 \rhd \{colored\_nodes\_m2(node)\}) \end{array} \right) \right)$
$act2$: $relations\_m2 := relations\_m2 \setminus (\{node\} \lhd active\_graph\_m2)$

When there are still vertices with active neighbours, it means that the coloring process is not complete: there are vertices which have chosen a color similar to the ones their neighbours have chosen. It means that they have to restart a new choice. The event RETURN_TO_CHOICE allows this new choice to happen by reinitializing the variables *relations_m2*, *active_nodes_m2* and *has_chosen_color_m2*.

EVENT RETURN_TO_CHOICE
WHEN
$grd1$: $active\_graph\_m2 \neq \varnothing$
$grd2$: $relations\_m2 = \varnothing$
$grd3$: $has\_chosen\_color\_m2 = active\_nodes\_m2$
THEN
$act1$: $relations\_m2 := active\_graph\_m2$
$act2$: $active\_nodes\_m2 := dom(active\_graph\_m2)$
$act3$: $has\_chosen\_color\_m2 := \varnothing$

The event COLORING_NODE3 finally colors the vertices in the variable *colored_nodes_m1* from the variable *colored_nodes_m2*, which is a temporary coloring variable.

EVENT COLORING_NODE3
REFINES EVENT COLORING_NODE2
ANY
$node$
$color$
WHERE
$grd1$: $node \in NODES$
$grd2$: $color \in COLORS$
$grd3$: $node \in dom(GRAPH)$
$grd4$: $node \notin has\_color\_m1$
$grd5$: $node \mapsto color \in colored\_nodes\_m2$
$grd6$: $active\_graph\_m2 = \varnothing$
THEN
$act1$: $colored\_nodes\_m1(node) := color$
$act2$: $has\_color\_m1 := has\_color\_m1 \cup \{node\}$

The event COLORING_NODE3 refines the event COLORING_NODE2 of the previous model; the guard means that each node has a color and that the global state corresponds to the fact that each node is no more active. A condition should be that for each vertex of the graph, the neighbours of the vertex have a color different from the one it has chosen. This property is simple to check and is localizable. It means that each vertex stops when all his neighbours are no more active. Moreover, the property to prove is that when a vertex has a proper color and when his neighbours have proper ones, then the algorithm will be in a global state leading to a global stability by fairness assumption.

The event COLORING3 controls the completion of the computation.

EVENT COLORING3
REFINES EVENT COLORING2
WHEN
$grd1$: $dom(GRAPH) = has\_color\_m1$
THEN
$act1$: $colored\_nodes\_m0 := colored\_nodes\_m1$

The following figure (figure 2.7) sums up the behaviour of the synchronous algorithm:

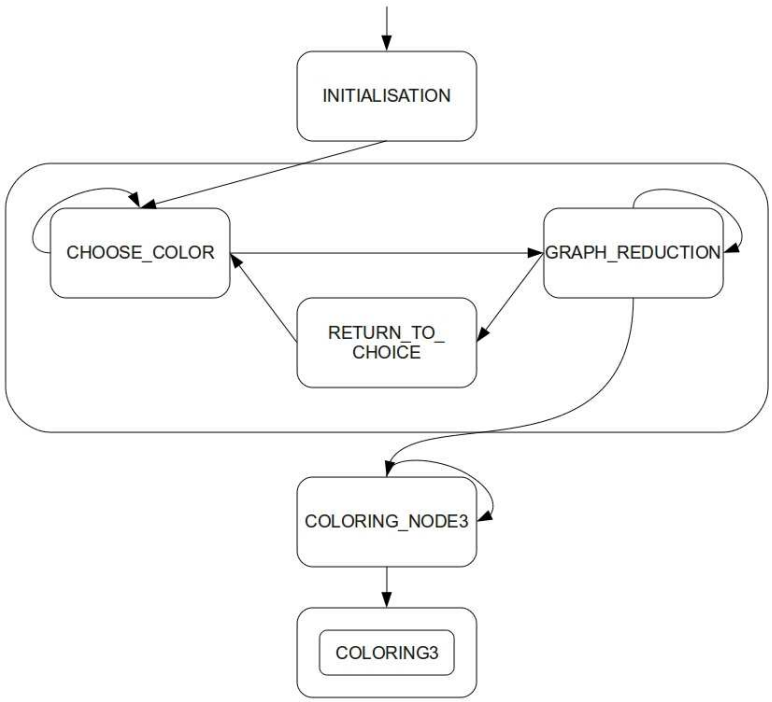


Figure 2.7: Synchronous algorithm behaviour

The current model is stating a solution close to the algorithm described by Métivier et al ([40]). Assumptions on the model are that the underlying computation model has a global clock and all the processors start the algorithm at the same time. The event CHOOSE_COLOR integrates a probabilistic argument when one wants to express the choice of a good set of colors for the remaining vertices. Moreover, our model can be either transformed into a synchronous algorithm in a very direct way or it can be refined into another model to provide an asynchronous algorithm.

2.4.2 Asynchronous Coloring Algorithms

The models defined in the machines **COLORING4**, **COLORING5** and **COLORING6** present asynchronous algorithms. The behaviour of these algorithms is the same as the one defined in **COLORING3**, except the fact that the vertices do not have to wait for all the others to choose their colors, update the links between them and their neighbours or restart choices.

a A First Asynchronous Coloring Algorithm (COLORING4 And COLORING5)

The asynchronous vertex coloring algorithm described in the machines **COLORING4** and **COLORING5** can be divided into steps:

1. First, a vertex, which still have active neighbours, that is a vertex which have neighbours sharing the same color with it, blocks itself and these neighbours, if none of them are already blocked or are blocking other nodes (event BLOCK_NODES),
2. Then, the blocked vertices choose colors, with the condition that the colors were not already chosen by their non-active neighbours (i.e. the neighbours which have chosen different colors from the ones the vertices have chosen in previous rounds of the algorithm) (event CHOOSE_COLOR),
3. The vertices update the connections between them: they erase the edges between them and their neighbours which have chose colors different from theirs, and the blocked vertices are released (event GRAPH_REDUCTION),
4. Finally, when none of the vertice have active neighbours anymore, the algorithm ends, because it means that all the vertices have chosen colors different from the ones their neighbours have chosen (event COLORING_NODE4).

The following figure (figure 2.8) describes this behaviour:

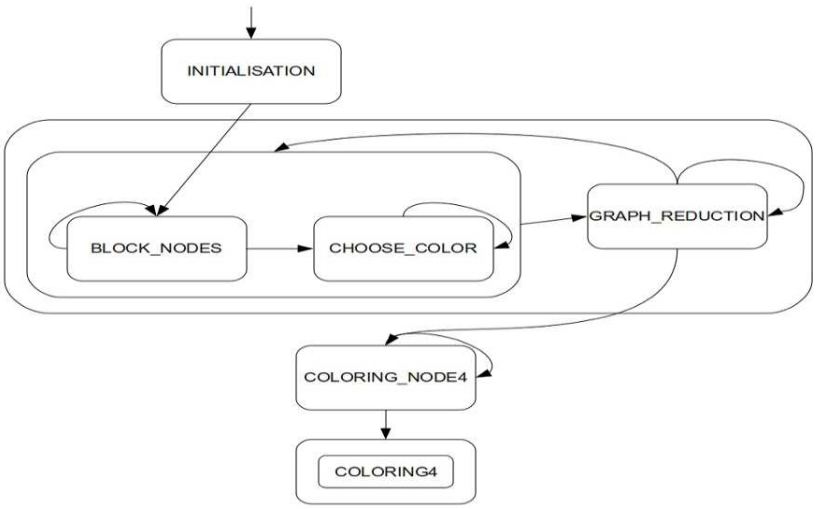


Figure 2.8: First asynchronous algorithm behaviour

COLORING4

Five new variables are defined:

$$
\begin{array}{l}
inv1 \ : active\_graph\_m2 \subseteq GRAPH \\
inv2 \ : colored\_nodes\_m2 \in NODES \nrightarrow COLORS \\
inv3 \ : has\_chosen\_color\_m2 \subseteq NODES \\
inv4 \ : blocked\_nodes\_m2 \subseteq GRAPH \\
inv5 \ : blocker\_m2 \subseteq NODES
\end{array}
$$

- *active_graph_m2* is a set containing the edges between the vertices of the graph *GRAPH* which are not already colored or share the same colors (i.e a vertex *a* which is adjacent to a vertex *b* and which has the same color as *b* is an active neighbour of *b*),
- *colored_nodes_m2* contains the vertices of the graph as keys and the colors assigned to these vertices as values,
- *has_chosen_color_m2* contains the vertices which have chosen a color during the execution of the algorithm,
- *blocked_nodes_m2* contains the edges blocked by a vertex during the execution of the algorithm,
- *blocker_m2* contains the vertices blocking edges during the execution of the algorithm.

These variables and the events defined in **COLORING4** have to satisfy the following invariants:

$$
\begin{array}{l}
inv6 \ : active\_graph\_m2 = active\_graph\_m2^{-1} \\
\\
inv7 \ : \forall n1 \cdot \forall n2 \cdot \left(\begin{array}{l} \left(\begin{array}{l} n1 \mapsto n2 \in GRAPH \wedge n2 \mapsto n1 \in GRAPH \wedge \\ n1 \mapsto n2 \notin active\_graph\_m2 \wedge n2 \mapsto n1 \notin active\_graph\_m2 \wedge \\ n1 \in dom(colored\_nodes\_m2) \wedge \\ n2 \in dom(colored\_nodes\_m2) \end{array} \right) \\ \Rightarrow \\ colored\_nodes\_m2(n1) \neq colored\_nodes\_m2(n2) \end{array} \right) \\
\\
inv8 \ : active\_graph\_m2 = \varnothing \\
\qquad \Rightarrow \\
\qquad \left(\begin{array}{l} \forall node \cdot \forall color \cdot \\ \left(\begin{array}{l} node \mapsto color \in colored\_nodes\_m2 \\ \Rightarrow \\ \neg color \in ran(ran(\{node\} \lhd GRAPH) \lhd colored\_nodes\_m1)) \end{array} \right) \end{array} \right) \\
\\
inv9 \ : active\_graph\_m2 \neq \varnothing \\
\qquad \Rightarrow \\
\qquad (\forall n \cdot n \in NODES \Rightarrow n \notin dom(colored\_nodes\_m1)) \\
\\
inv10 \ : colored\_nodes\_m1 \subseteq colored\_nodes\_m2
\end{array}
$$

- the invariant *inv6* expresses that the graph reduced is still symmetric,
- the invariant *inv7* expresses that if two vertices of *GRAPH* are adjacent in *GRAPH* but not linked in *active_graph_m2* and have chosen colors, then the color chosen by the first vertex is different from the one chosen by the second,
- the invariant *inv8* expresses that if the vertices in *GRAPH* have no active neighbours anymore, then the color of a vertex in *colored_nodes_m2* is different from the ones its neighbours have definitely chosen in *colored_nodes_m1*,
- the invariant *inv9* expresses that if there are still vertices with active neighbours, then no vertex has a definitive color in *colored_nodes_m1*,
- the invariant *inv10* expresses that the process in **COLORING4** is simulating the previous process in **COLORING2**; we indicate a possible error in the choice of a color and we state that the new variable *colored_nodes_m2* is correct with respect to *colored_nodes_m1*: $colored\_nodes\_m1 \subseteq colored\_nodes\_m2$.

The initial values of the variables are the empty set ($\varnothing$) except for *active_graph_m2*, which is initialised with the edges of *GRAPH*.

INITIALISATION
$act1$: $colored\_nodes\_m0 := \varnothing$
$act2$: $colored\_nodes\_m1 := \varnothing$
$act3$: $has\_color\_m1 := \varnothing$
$act4$: $active\_graph\_m2 := GRAPH$
$act5$: $colored\_nodes\_m2 := \varnothing$
$act6$: $has\_chosen\_color\_m2 := \varnothing$
$act7$: $blocked\_nodes\_m2 := \varnothing$
$act8$: $blocker\_m2 := \varnothing$

The model introduces the notion of a vertex blocking its neighbours: a mechanism which manages the creation of entities called *"blocked balls"* with blocking vertices as their centers and blocked neighbours at their peripheries, is specified by the event BLOCK_NODES.

EVENT BLOCK_NODES
ANY
$node$
WHERE
$grd1$: $node \in NODES$
$grd2$: $node \notin dom(blocked\_nodes\_m2) \cup ran(blocked\_nodes\_m2)$
$grd3$: $((\{node\} \lhd active\_graph\_m2) \cup (active\_graph\_m2 \rhd \{node\})) \cap blocked\_nodes\_m2 = \varnothing$
$grd4$: $ran(\{node\} \lhd active\_graph\_m2) \cap (dom(blocked\_nodes\_m2) \cup ran(blocked\_nodes\_m2)) = \varnothing$
$grd5$: $node \notin blocker\_m2$
$grd6$: $node \in dom(active\_graph\_m2)$
THEN
$act1$: $blocked\_nodes\_m2 := blocked\_nodes\_m2 \cup ((\{node\} \lhd active\_graph\_m2) \cup (active\_graph\_m2 \rhd \{node\}))$
$act2$: $blocker\_m2 := blocker\_m2 \cup \{node\}$

A vertex *node* is not in a "ball" (center of peripheral - blocker or blocked). It means that it can block itself and its neighbours. The effect of the event BLOCK_NODES is to create a new blocked ball and related blocked nodes. The event is not triggered, if one node member of the ball is already blocking or being blocked. The figure (see figure 2.9) provides an example and some counter-examples.

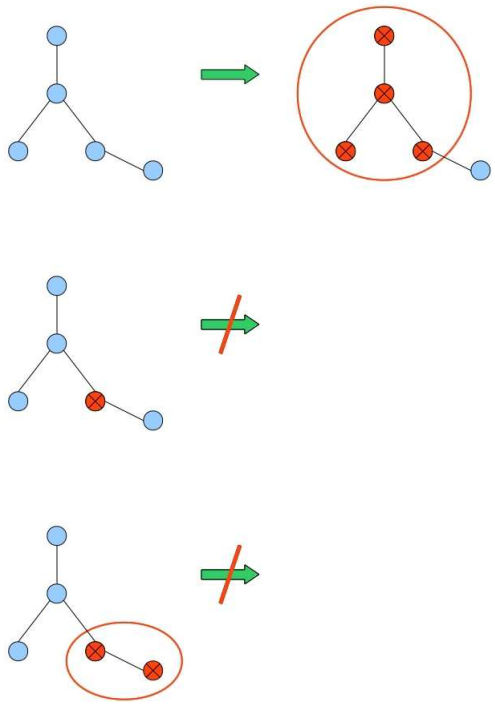


Figure 2.9: Examples and counter-examples for BLOCK_NODES

The next event (CHOOSE_COLOR) colors the set of vertices belonging to a *"blocked ball"*.

EVENT CHOOSE_COLOR
ANY
$node$
$color$
WHERE
$grd1$: $node \in NODES$
$grd2$: $color \in COLORS$
$grd3$: $node \in dom(active\_graph\_m2)$
$grd4$: $node \notin has\_chosen\_color\_m2$
$grd5$: $color \notin ran(ran(\{node\} \lhd (GRAPH \setminus active\_graph\_m2)) \lhd colored\_nodes\_m2)$
$grd6$: $node \in dom(blocked\_nodes\_m2) \cup ran(blocked\_nodes\_m2)$
THEN
$act1$: $colored\_nodes\_m2(node) := color$
$act2$: $has\_chosen\_color\_m2 := has\_chosen\_color\_m2 \cup \{node\}$

The event is triggered when a vertex, which is the origin of an edge in *active_graph_m2*, has not already chosen a color and is a member of a *"blocked ball"* (center or neighbour). The vertex chooses a color which have not already been chosen by his non-active neighbours.

The next event (GRAPH_REDUCTION) updates the graph, when the vertices have chosen their colors.

EVENT GRAPH_REDUCTION
ANY
$node$
WHERE
$grd1$: $node \in NODES$
$grd2$: $node \in dom(active\_graph\_m2)$
$grd3$: $node \in has\_chosen\_color\_m2$
$grd4$: $node \in dom(colored\_nodes\_m2)$
$grd5$: $(\{node\} \cup dom(active\_graph\_m2 \rhd \{node\})) \subseteq has\_chosen\_color\_m2$
$grd6$: $((\{node\} \lhd active\_graph\_m2) \cup (active\_graph\_m2 \rhd \{node\})) \subseteq blocked\_nodes\_m2$
$grd7$: $node \in dom(blocked\_nodes\_m2) \cup ran(blocked\_nodes\_m2)$
$grd8$: $node \in blocker\_m2$
THEN
$act1$: $has\_chosen\_color\_m2 := has\_chosen\_color\_m2 \setminus (\{node\} \cup dom(active\_graph\_m2 \rhd \{node\}))$
$act2$: $blocked\_nodes\_m2 := blocked\_nodes\_m2 \setminus ((\{node\} \lhd active\_graph\_m2) \cup (active\_graph\_m2 \rhd \{node\}))$
$act3$: $active\_graph\_m2 := \left(active\_graph\_m2 \setminus \left(\left(\begin{array}{l} (\{node\} \lhd active\_graph\_m2) \\ \rhd dom(colored\_nodes\_m2 \rhd \{colored\_nodes\_m2(node)\}) \end{array} \right) \cup \left(\begin{array}{l} (\{node\} \lhd active\_graph\_m2) \\ \rhd dom(colored\_nodes\_m2 \rhd \{colored\_nodes\_m2(node)\}) \end{array} \right)^{-1} \right) \right)$
$act4$: $blocker\_m2 := blocker\_m2 \setminus \{node\}$

The event GRAPH_REDUCTION is triggered when all the members of "blocked ball" have chosen their colors. Then, the edges between blocked neighbours which have chosen different colors are removed, and blocked nodes become unblocked and are removed from the set of nodes which have chosen a color.

The two next events, COLORING_NODE4 and COLORING4, occur when the nodes have correctly chosen their colors: these events color the graph using the set of colors obtained during the previous steps of the algorithms.

EVENT COLORING_NODE4
REFINES EVENT COLORING_NODE2
ANY
$node$
$color$
WHERE
$grd1$: $node \in NODES$
$grd2$: $color \in COLORS$
$grd3$: $node \in dom(GRAPH)$
$grd4$: $node \notin has\_color\_m1$
$grd5$: $node \mapsto color \in colored\_nodes\_m2$
THEN
$act1$: $colored\_nodes\_m1(node) := color$
$act2$: $has\_color\_m1 := has\_color\_m1 \cup \{node\}$

EVENT COLORING4
REFINES EVENT COLORING2
  WHEN
    $grd1 : dom(GRAPH) = has\_color\_m1$
  THEN
    $act1 : colored\_nodes\_m0 := colored\_nodes\_m1$

COLORING5 : The Refinement Of **COLORING4**

This refinement has been made in order to remove the global knowledge of the structure of the graph *GRAPH* from the event **COLORING4**. Some invariants have been defined in order to achieve that goal:

$inv1 : dom(colored\_nodes\_m1) = has\_color\_m1$

$inv2 : \forall node \cdot node \in has\_chosen\_color\_m2 \Rightarrow node \in dom(colored\_nodes\_m2)$

$inv3 : \forall n \cdot n \in dom(GRAPH \setminus active\_graph\_m2) \Rightarrow n \in dom(colored\_nodes\_m2)$

$inv4 : active\_graph\_m2 = \varnothing \Rightarrow dom(colored\_nodes\_m2) = dom(GRAPH)$

$inv5 : active\_graph\_m2 = \varnothing \wedge colored\_nodes\_m1 \neq \varnothing \wedge colored\_nodes\_m2 = colored\_nodes\_m1$
$\Rightarrow$
$(\forall node \cdot node \in dom(GRAPH) \Rightarrow node \in dom(colored\_nodes\_m1))$

$inv6 : has\_chosen\_color\_m2 \subseteq dom(blocked\_nodes\_m2) \cup ran(blocked\_nodes\_m2)$

- *inv1* expresses that the vertices with a color attached to them in *colored_nodes_m1* have chosen their definitive colors (they are members of *has_color_m1*),
- *inv2* expresses that the vertices with a temporary colors are members of the set containig the vertices which have chosen temporary colors,
- *inv3* expresses that the vertices which have no active neighbours anymore have already chosen temporary colors,
- *inv4* expresses that if all the vertices of *GRAPH* have no active neighbours anymore, it implies that they have already chosen a color,
- *inv5* expresses that if all the vertices of *GRAPH* have no active neighbours anymore and the set of temporary colors is equal to the set of final colors, then all the vertices have chosen their final colors,
- *inv6* expresses that the set of nodes which have chosen a color is a subset of the set of blocked nodes.

These invariants allow us to remove the global knowledge of the graph *GRAPH* from the event COLORING4, leading to the event COLORING5, where it is replaced by the knowledge of the sets of active neighbours of the vertices.

EVENT COLORING5
REFINES $COLORING4$
  WHEN
    $grd1 : active\_graph\_m2 = \varnothing$
    $grd2 : colored\_nodes\_m1 \neq \varnothing$
    $grd3 : colored\_nodes\_m2 = colored\_nodes\_m1$
  THEN
    $act1 : colored\_nodes\_m0 := colored\_nodes\_m1$

We summarize the current description of our models by stating that these models are still abstract and can be refined into more concrete ones later.

b A Second Asynchronous Coloring Algorithm (**COLORING6**)

The machine **COLORING6** describes another asynchronous coloring algorithm different from the previous one, described in **COLORING4** and **COLORING5**: the algorithm described is local, i.e. focused on the point of view of a node.

This new algorithm can be divided into several steps:

- a round where a vertex with active neighbours generates randomly a color and send it to his neighbours,
- another round where this vertex receive the colors chosen by his neighbours, one at a time,
- and finally, a round where it updates its active neighbours.

New definitions of the graph *GRAPH*, and several new constants and axioms are needed for the modelling. Therefore, we extend the previous context **COLORING0** with a new one ,**COLORING7**, which is used by the machine **COLORING6**.

$$
\begin{array}{l}
axm1 : G \in NODES \to \mathbb{P}(NODES) \\
axm2 : \forall x \cdot (x \in NODES \Rightarrow G(x) = GRAPH[\{x\}]) \\
axm3 : INIT\_FORBIDDEN\_COLORS \in NODES \to \mathbb{P}(COLORS) \\
axm4 : \forall n \cdot n \in NODES \Rightarrow n \mapsto \varnothing \in INIT\_FORBIDDEN\_COLORS \\
axm5 : INIT\_COLORS\_NAT \in NODES \to \mathbb{P}(COLORS) \\
axm6 : \forall n \cdot n \in NODES \Rightarrow n \mapsto \varnothing \in INIT\_COLORS\_NAT \\
axm7 : INIT\_HAS\_FINISHED \in NODES \to BOOL \\
axm8 : \forall n \cdot n \in NODES \Rightarrow n \mapsto FALSE \in INIT\_HAS\_FINISHED \\
axm9 : INIT\_RCV\_FROM \in NODES \to \mathbb{P}(NODES) \\
axm10 : \forall n \cdot n \in NODES \Rightarrow n \mapsto \varnothing \in INIT\_RCV\_FROM \\
axm11 : INIT\_RCV\_FROM\_NODES \in NODES \to (NODES \nrightarrow \mathbb{P}(COLORS)) \\
axm12 : \forall n \cdot n \in NODES \Rightarrow n \mapsto \varnothing \in INIT\_RCV\_FROM\_NODES
\end{array}
$$

In this new context, we have a redefinition of the graph *GRAPH* and some definitions of new constants:

- *G* is a representation of the graph. The graph is defined in such a way that a vertex is associated with its neighbours,
- *INIT_FORBIDDEN_COLORS* is a constant which associates a vertex with a set containing its forbidden colors. As it is used to define the initial state of the current model and the vertices have no forbidden colors during the initialisation, it associates every node of *GRAPH* with the empty set ($\varnothing$),
- *INIT_COLORS_NAT* is a constant which associates a vertex with a the color it as chosen. As it is used to define the initial state of the current model and the vertices have no color during the initialisation, it associates every node of *GRAPH* with an empty color ($\varnothing$),
- *INIT_HAS_FINISHED* is a constant which associates a vertex with a state: *TRUE* if it has finished executing the algorithm, otherwise *FALSE*. As it is used to define the initial state of the current model, it associates every node of *GRAPH* with *FALSE*,
- *INIT_RCV_FROM* is a constant which associates a vertex with the neighbours from which it has received a color. As it is used to define the initial state of the current model and the vertices have sent nothing yet, it associates every node of *GRAPH* with the empty set ($\varnothing$),
- *INIT_RCV_FROM_NODES* is a constant which associates a vertex with the neighbours and the colors they have sent. As it is used to define the initial state of the current model and the vertices have sent nothing yet, it associates every node of *GRAPH* with the empty set ($\varnothing$).

In the machine **COLORING6**, we define new variables:

$$
\begin{array}{l}
inv1 : graph \in NODES \rightarrow \mathbb{P}(NODES) \\
inv2 : forbiddenColors \in NODES \rightarrow \mathbb{P}(COLORS) \\
inv3 : colors \in NODES \rightarrow \mathbb{P}(COLORS) \\
inv4 : control \in NODES \rightarrow \mathbb{P}(NODES) \\
inv5 : hasFinished \in NODES \rightarrow BOOL \\
inv6 : send \in NODES \nrightarrow \mathbb{P}(COLORS) \\
inv7 : receiveFrom \in NODES \rightarrow \mathbb{P}(NODES) \\
inv8 : receiveFromNode \in NODES \rightarrow (NODES \nrightarrow \mathbb{P}(COLORS)) \\
inv9 : block \in NODES \rightarrow \mathbb{P}(NODES)
\end{array}
$$

- *graph* contains the vertices of the graph associated with the neighbours which have the same color as them,
- *forbiddenColors* contains the vertices of the graph associated with the colors they can not choose,
- *colors* contains the vertices of the graph associated with the colors they have chosen,
- *control* contains the vertices of the graph associated with the neighbours to which they have not yet compared their colors,
- *hasFinished* contains the state vertices of the graph: *TRUE* if they have finished executing the algorithm, otherwise *FALSE*,
- *send* contains the vertices of the graph associated with the colors they have sent to their neighbours,
- *receiveFrom* contains the vertices of the graph associated with the neighbours from which they have received a color,
- *receiveFromNode* contains the vertices of the graph associated with the neighbours from which they have received a color and said color,
- *block* contains the vertices of the graph associated with the neighbours which are blocking them.

These variables and the events of **COLORING6** have to respect the following invariants:

$$
\begin{array}{l}
inv10 : NODES = dom(graph) \\
inv11 : NODES = dom(colors) \\
inv12 : NODES = dom(forbiddenColors) \\
inv13 : NODES = dom(control) \\
inv14 : NODES = dom(hasFinished) \\
inv15 : NODES = dom(send) \\
inv16 : NODES = dom(receiveFrom) \\
inv17 : NODES = dom(block) \\
\\
inv18 : \forall n \cdot (receiveFrom(n) = dom(receiveFromNode(n))) \\
inv19 : \forall node \cdot hasFinished(node) = FALSE \\
\qquad\qquad \Rightarrow node \notin dom(colored\_nodes\_m1) \\
inv20 : \forall n1 \cdot n1 \in dom(colored\_nodes\_m1) \\
\qquad\quad \Rightarrow \{colored\_nodes\_m1(n1)\} = colors(n1) \\
\\
inv21 : \forall n1 \cdot \forall n2 \cdot \left(\begin{array}{l} \left(\begin{array}{l} n1 \in dom(colored\_nodes\_m1) \wedge \\ n2 \in dom(colored\_nodes\_m1) \wedge \\ n1 \in G(n2) \wedge \\ n2 \in G(n1) \end{array} \right) \\ \Rightarrow \\ colored\_nodes\_m1(n1) \neq \\ colored\_nodes\_m1(n2) \end{array} \right) \\
\\
inv22 : \forall n1 \cdot n1 \in dom(colored\_nodes\_m1) \\
\qquad \Rightarrow graph(n1) = \varnothing \\
inv23 : \forall n \cdot hasFinished(n) = TRUE \Rightarrow graph(n) = \varnothing \\
\\
inv24 : \forall n1 \cdot \forall n2 \cdot \left(\begin{array}{l} \left(\begin{array}{l} n1 \in dom(colored\_nodes\_m1) \wedge \\ n2 \in dom(colored\_nodes\_m1) \wedge \\ n1 \in G(n2) \wedge \\ n2 \in G(n1) \end{array} \right) \\ \Rightarrow \\ n1 \notin graph(n2) \wedge n2 \notin graph(n1) \end{array} \right)
\end{array}
$$

- the invariants from *inv10* to *inv17* state that the domains of *graph*, *forbiddenColors*, *colors*, *control*, *hasFinished*, *send*, *receiveFrom*, *block* are equal to *NODES*,
- *inv18* states that if a node has received something from a neighbour n, then n has sent it a color,
- *inv19* states that if a node has not stopped executing the algorithm, then it does not have a definitive color,
- *inv20* states that if a node has its definitive color, then this color is the one it has chosen during the execution of the algorithm,
- *inv21* expresses that if two nodes which are neighbours in the original graph have their definitive colors, then these colors are different,
- *inv22* expresses that if a vertex has its definitive color, then its list of active neighbours is empty,
- *inv23* expresses that if a vertex has stopped executing the algorithm, then its list of active neighbours is empty,
- *inv24* expresses that if two nodes which are neighbours in the original graph have their definitive colors, then they are not in each other's list of active neighbours.

The initialisation of *colored_nodes_m0*, *colored_nodes_m1*, *has_color_m1* with the empty set ($\varnothing$) expresses the fact that at the initial state of the algorithm, no vertex has a definitive color.

INITIALISATION
$act1 : colored\_nodes\_m0 := \varnothing$
$act2 : colored\_nodes\_m1 := \varnothing$
$act3 : has\_color\_m1 := \varnothing$
$act4 : graph := G$
$act5 : forbiddenColors := INIT\_FORBIDDEN\_COLORS$
$act6 : colors := INIT\_COLORS\_NAT$
$act7 : control := G$
$act8 : hasFinished := INIT\_HAS\_FINISHED$
$act9 : send := INIT\_COLORS\_NAT$
$act10 : receiveFrom := INIT\_RCV\_FROM$
$act11 : receiveFromNode := INIT\_RCV\_FROM\_NODES$
$act12 : block := INIT\_RCV\_FROM$

The other variables are initialised with respect to the initial state of the machine:

- *graph* is initialised with *G*, which contains the vertices of *GRAPH* associated with their neighbours,
- *forbiddenColors* is initialised in such a way that at the beginning of the algorithm, the vertices have no forbidden colors,
- *colors* is initialised in such a way that at the beginning of the algorithm, the vertices have no colors,
- *control* is initialised with *G*, which contains the vertices of *GRAPH* associated with their neighbours,
- *hasFinished* is initialised in such a way that no vertex of *GRAPH* has finished executing the algorithm,
- *send* is initialised in such a way that no vertex of *GRAPH* has sent a color,
- *receiveFrom* is initialised in such a way that no vertex of *GRAPH* has received a message from its neighbours,
- *receiveFromNode* is initialised in such a way that no vertex of *GRAPH* has received a color from its neighbours,
- *block* is initialised in such a way that is no neighbour blocking a vertex of *GRAPH*.

The event GENERATE_AND_SEND_COLOR models the generation and the sending of a color by a node. It is triggered when the vertex has active neighbours, has generated and sent no color, has not yet compared its color to the colors of its neighbours, has received nothing, is not blocked and has not already finished executing the algorithm. If these conditions hold, it generates a random color, not already taken by its non-active neighbours (not in its list of forbidden colors), and sends it to its neighbours. We can notice that when a node has no active neighbours anymore, it does not generate and send a new color, because the last color sent by it, is stored in the variable *send* and is not modified.

EVENT GENERATE_AND_SEND_COLOR
ANY
$node$
$color$
WHERE
$grd1$: $node \in NODES$
$grd2$: $graph(node) \neq \varnothing$
$grd3$: $color \in COLORS$
$grd4$: $color \notin forbiddenColors(node)$
$grd5$: $control(node) = G(node)$
$grd6$: $send(node) = \varnothing$
$grd7$: $receiveFrom(node) = \varnothing$
$grd8$: $block(node) = \varnothing$
$grd9$: $hasFinished(node) = FALSE$
THEN
$act1$: $colors(node) := \{color\}$
$act2$: $send(node) := \{color\}$

The event RECEIVE_FROM_NEIGHBOUR models the receiving of the colors sent by its neighbours by a vertex. This event is triggered when the vertex has active neighbours and has sent something to them. Then, the node receive the colors sent by its neighbours and blocks them.

EVENT RECEIVE_FROM_NEIGHBOUR
ANY
$node$
$neighbour$
WHERE
$grd1$: $node \in NODES$
$grd2$: $neighbour \in G(node)$
$grd3$: $send(node) \neq \varnothing$
$grd4$: $neighbour \in NODES$
$grd5$: $send(neighbour) \neq \varnothing$
$grd6$: $neighbour \notin receiveFrom(node)$
$grd7$: $receiveFrom(node) \neq G(node)$
$grd8$: $node \notin block(neighbour)$
$grd9$: $graph(node) \neq \varnothing$
THEN
$act1$: $receiveFrom(node) := receiveFrom(node) \cup \{neighbour\}$
$act2$: $receiveFromNode(node) := receiveFromNode(node) \cup \{neighbour \mapsto send(neighbour)\}$
$act3$: $block(neighbour) := block(neighbour) \cup \{node\}$

When a vertex receives colors from its neighbours, it performs some treatments and operations based on comparisons between its color and the colors it received. The treatments can be classified into three types:

1. First Treatment: The vertex receives a color from an active neighbour and which is different from its color. Then, the neighbour is marked as observed, is removed from the list of active neighbours of the vertex and the color chosen by the neighbour is stored in its list of forbidden colors.

EVENT FOR_RECEIVE_COLOR1
ANY
$node$
$neighbour$
WHERE
$grd1$: $node \in NODES$
$grd2$: $graph(node) \neq \varnothing$
$grd3$: $receiveFrom(node) = G(node)$
$grd4$: $control(node) \neq \varnothing$
$grd5$: $neighbour \in G(node)$
$grd6$: $neighbour \in receiveFrom(node)$
$grd7$: $neighbour \in control(node)$
$grd8$: $neighbour \in graph(node)$
$grd9$: $(receiveFromNode(node))(neighbour) \neq colors(node)$
$grd10$: $node \in block(neighbour)$
$grd11$: $neighbour \in block(node)$
THEN
$act1$: $forbiddenColors(node) := forbiddenColors(node) \cup (receiveFromNode(node))(neighbour)$
$act2$: $graph(node) := graph(node) \setminus \{neighbour\}$
$act3$: $control(node) := control(node) \setminus \{neighbour\}$

2. Second Treatment: The vertex receives a color from an active neighbour and which is the same as its color. Because the colors are the same, nothing is done. The neighbour is just marked as observed.

EVENT FOR_RECEIVE_COLOR2
  ANY
    $node$
    $neighbour$
  WHERE
    $grd1$: $node \in NODES$
    $grd2$: $graph(node) \neq \varnothing$
    $grd3$: $receiveFrom(node) = G(node)$
    $grd4$: $control(node) \neq \varnothing$
    $grd5$: $neighbour \in G(node)$
    $grd6$: $neighbour \in receiveFrom(node)$
    $grd7$: $neighbour \in control(node)$
    $grd8$: $neighbour \in graph(node)$
    $grd9$: $(receiveFromNode(node))(neighbour) = colors(node)$
    $grd10$: $node \in block(neighbour)$
    $grd11$: $neighbour \in block(node)$
  THEN
    $act1$: $control(node) := control(node) \setminus \{neighbour\}$

3. Third Treatment: The vertex receives a color from a non-active neighbour. The neighbour is marked as observed and the color received from it is stored in the list of forbidden colors of the vertex.

EVENT FOR_RECEIVE_COLOR3
  ANY
    $node$
    $neighbour$
  WHERE
    $grd1$: $node \in NODES$
    $grd2$: $graph(node) \neq \varnothing$
    $grd3$: $receiveFrom(node) = G(node)$
    $grd4$: $control(node) \neq \varnothing$
    $grd5$: $neighbour \in G(node)$
    $grd6$: $neighbour \in receiveFrom(node)$
    $grd7$: $neighbour \in control(node)$
    $grd8$: $neighbour \notin graph(node)$
    $grd9$: $node \in block(neighbour)$
  THEN
    $act1$: $forbiddenColors(node) := forbiddenColors(node) \cup (receiveFromNode(node))(neighbour)$
    $act2$: $control(node) := control(node) \setminus \{neighbour\}$

The event END_FOR_RECEIVE is triggered when a node, which still has active neighbours, has performed all the previous treatments based on the colors comparisons. The fact that it still has active neighbours means that these neighbours have chosen the same colors as the one it has chosen, so the variables representing its receiving of colors from its neighbours (*receiveFrom* and *receiveFromNode*), are set to the empty set ($\varnothing$). Therefore, the node, after being released by the neighbours which has blocked it, can restart a new choice of color.

EVENT END_FOR_RECEIVE
  ANY
    $node$
  WHERE
    $grd1$: $node \in NODES$
    $grd2$: $graph(node) \neq \varnothing$
    $grd3$: $control(node) = \varnothing$
  THEN
    $act1$: $send(node) := \varnothing$
    $act2$: $control(node) := G(node)$
    $act3$: $receiveFrom(node) := \varnothing$
    $act4$: $receiveFromNode(node) := \varnothing$

The event RELEASE is triggered when a node is blocking a neighbour, has reinitialised the values of the variables representing its receiving of colors from its neighbours (*receiveFrom* and *receiveFromNode*), and when its neighbour has done the same thing. Then, the node releases the neighbour.

EVENT RELEASE
ANY
$node$
$neighbour$
WHERE
$grd1$: $node \in NODES$
$grd2$: $neighbour \in NODES$
$grd3$: $receiveFrom(node) = \varnothing$
$grd4$: $receiveFrom(neighbour) = \varnothing$
$grd5$: $node \in block(neighbour)$
THEN
$act1$: $block(neighbour) := block(neighbour) \setminus \{node\}$

The event FINISH is triggered when a node has no active neighbours anymore and is in the state *FALSE* (i.e. the node has not finished executing the algorithm). If these conditions hold, the state of the vertex becomes *TRUE*, meaning it has finished executing the algorithm and the variables representing its receiving of colors from its neighbours (*receiveFrom* and *receiveFromNode*), are set to the empty set ($\varnothing$).

EVENT FINISH
ANY
$node$
WHERE
$grd1$: $node \in NODES$
$grd2$: $graph(node) = \varnothing$
$grd3$: $hasFinished(node) = FALSE$
THEN
$act1$: $hasFinished(node) := TRUE$
$act2$: $receiveFrom(node) := \varnothing$
$act3$: $receiveFromNode(node) := \varnothing$

The event COLORING_NODE6 assigns its definitive color to a vertex. It is triggered when a vertex has finished executing the algorithm (i.e. its state is *TRUE*), that is, when the color it has chosen is different from the ones its neighbours have chosen, and if it is not blocked anymore. Then, the color chosen by the vertex becomes its definitive color.

EVENT COLORING_NODE6
REFINES EVENT COLORING_NODE2
ANY
$node$
$color$
WHERE
$grd1$: $node \in NODES$
$grd2$: $color \in COLORS$
$grd4$: $node \notin has\_color\_m1$
$grd6$: $colors(node) = \{color\}$
$grd7$: $hasFinished(node) = TRUE$
$grd8$: $color \in colored\_nodes\_m1[G(node)]$
$grd9$: $block(node) = \varnothing$
THEN
$act1$: $colored\_nodes\_m1 := colored\_nodes\_m1 \cup \{node \mapsto color\}$
$act2$: $has\_color\_m1 := has\_color\_m1 \cup \{node\}$

The following figure (see figure 2.10) illustrates the behaviour of the algorithm when it is localised to a node:

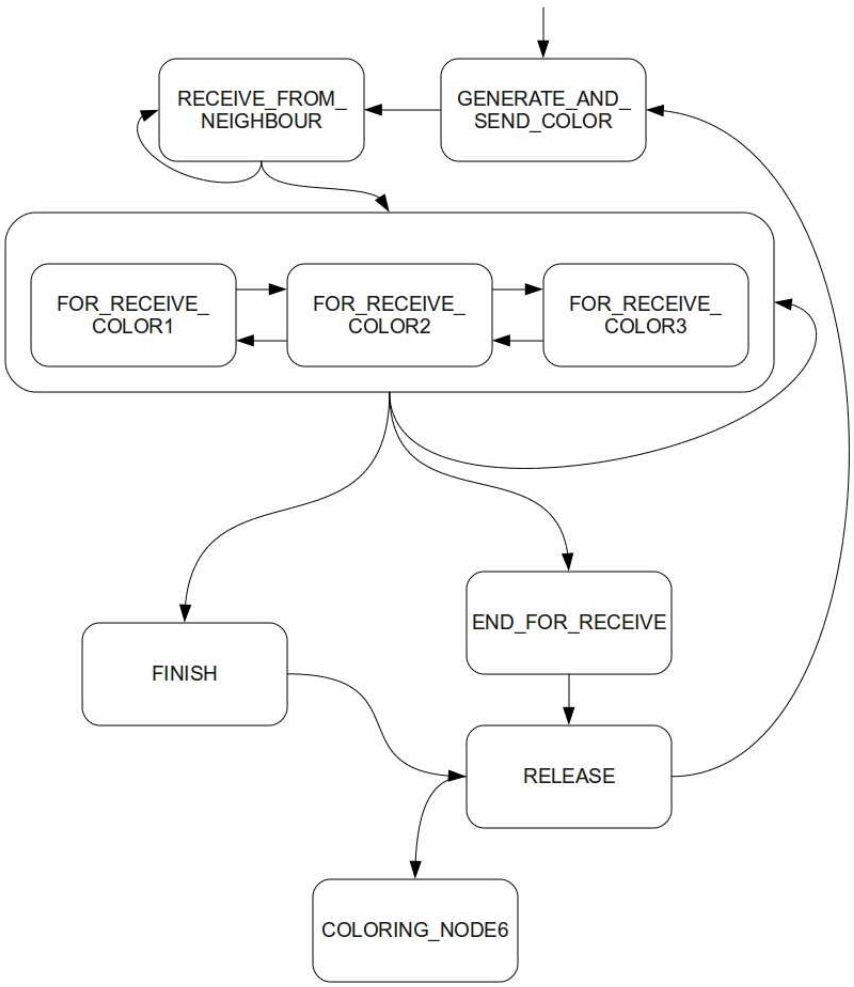


Figure 2.10: A vertex's behaviour for the local and asynchronous algorithm

The event COLORING6 occurs when the nodes have correctly chosen their final colors: this event colors the graph using the set of colors obtained during the previous steps of the algorithm.

EVENT COLORING6
REFINES EVENT COLORING2
  WHEN
    $grd1 : NODES = has\_color\_m1$
  THEN
    $act1 : colored\_nodes\_m0 := colored\_nodes\_m1$

The general behaviour of this algorithm is summed up by the following figure (see figure 2.11) :

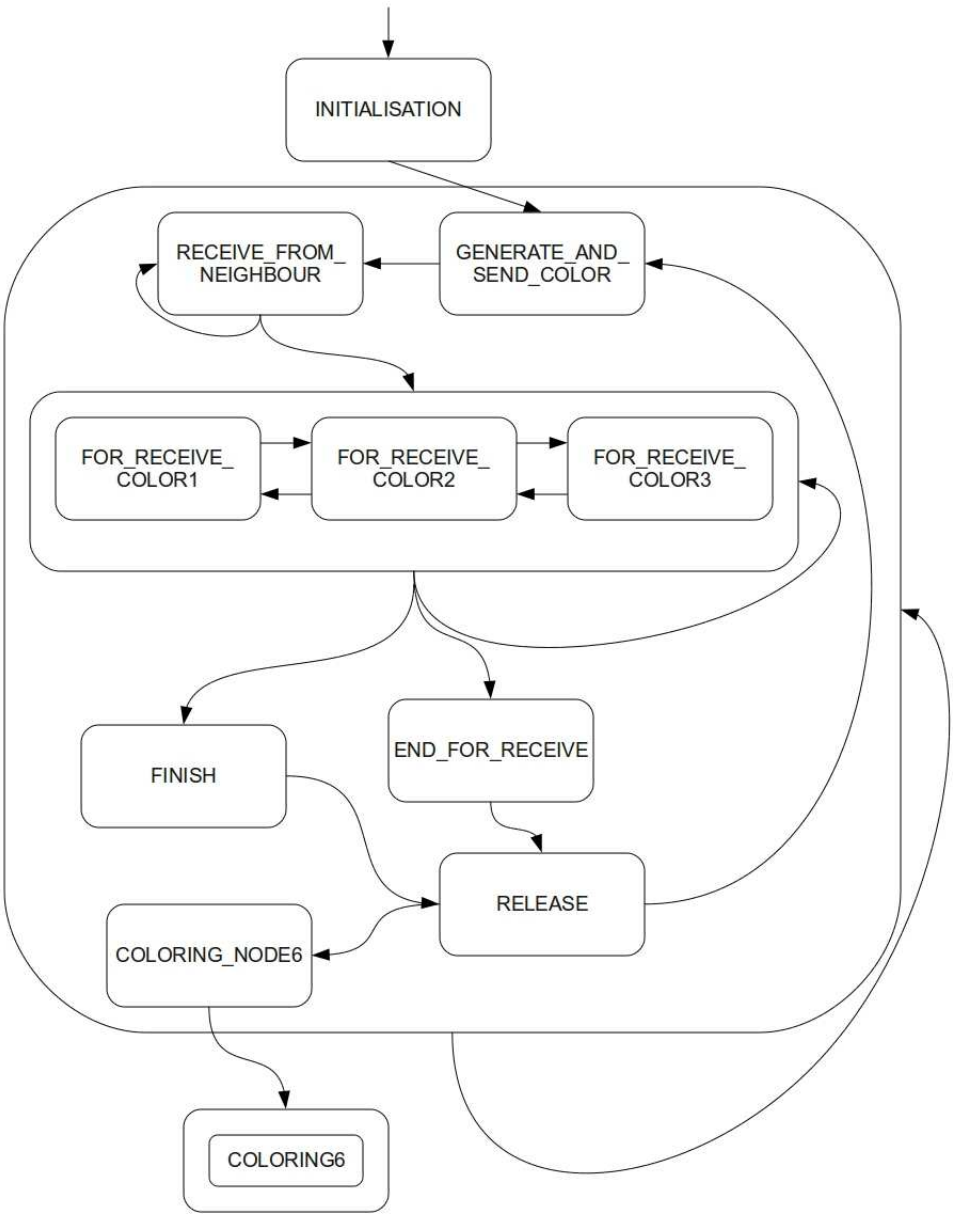


Figure 2.11: Asynchronous algorithm behaviour (local algorithm)

We summarize the current description of our models by stating that this model is the one really close to an algorithmic form (i.e. simple and local guards, actions, representations of the graph...) and is concrete and local enough to be translated into a programming language, like the one (*Java*) used by Visidia.

Chapter 3

Verification

3.1 Verifying The Behaviour Of The Algorithms

The behaviour of the algorithms, we presented in the previous sections of this report, must be verified, in order to ensure that this behaviour repect the specifications and that the algorithms do what we expect them to: coloring the vertices of a graph in such a way that no two adjacent vertices share the same color. Two formal tools are used for this verification: the model-checker ProB ([29]), which can animate an Event B model and a software dedicated to the simulation and visualisation of distributed algorithms, ViSiDiA ([12, 18, 48, 47, 14]).

3.1.1 Using ProB

ProB is a model-checker ([29]) for the B method and its derivatives (Event B...). Some interesting features are that it is quite practical to detect errors in Event B specifications and it also provides an *animator*, which we can use to visualise the behaviour of our Event B models (variables changes, states evolution...).
The features allow us to verify that our models indeed respect our specifications and do the coloration of the vertices of a given graph in a correct way (no two adjacent vertices share the same color).
ProB is a useful tool for verifying of the behaviour of non-enough concrete models, as it uses the Event B language. However, due to to the fact that the model-checker verify all the reachable states in the system, problems, like state explosion ([24]), occur and there are sometimes restrictions on the size of the sets we can use and on variables ranges (e.g. integers). There are limitations in using ProB: we can only verify the behaviour of complex graph coloring algorithms with a few number of nodes (up to 3 or 4), the number of colors are limited (up to 3 or 4)...
Therefore, we combine the use of ProB with the use another software, called ViSiDiA, which goal is the simulation and visualisation of distributed algorithms.

3.1.2 Using ViSiDiA

a From Event B To ViSiDiA

Introduction

ViSiDiA ([12, 18, 48, 47, 14]) is a formal verification software dedicated to the simulation and visualisation of distributed algorithms. Our purpose here is to find a suitable way to obtain Event B models which can be transformed into ViSiDiA models, in order to see if the behaviour of the algorithms defined in these models is correct. As seen in the works of Mohamed Tounsi, Ahmed Hadj-Kacem, Mohamed Mosbah and Dominique Méry et al. ([14]), the goal is to have a last concrete model, which can be translated with no or little efforts into the language used by ViSiDiA (*Java*) (see figure 3.1).

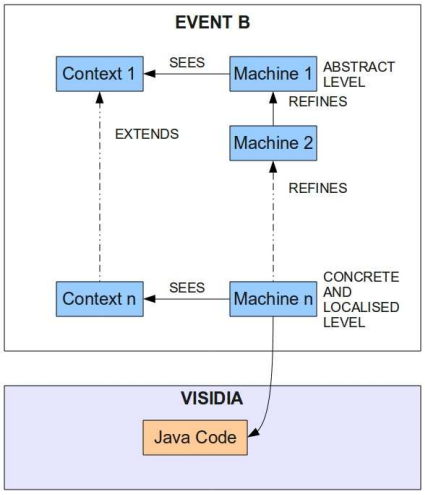


Figure 3.1: From Event B to ViSiDiA ([14])

Required Characteristics Of Event B Models

There are characteristics that Event B models must have, in order to be translatable into the ViSiDiA programming language. First of all, it should be noticed that the underlying model behind ViSiDiA is based on graph theory, and the algorithms described using ViSiDiA are local: they are not described at the level of the whole graph, but at the level of a vertex, that is, the behaviour are the same for every vertex in the graph. We can see that the vertices have no global knowledge of the state of the graph or its number of nodes... The only knowledge they can get is the one that their neighbours give them through messages passing. Therefore, the system described by an Event B model must be one that can be described with a graph and use messages passing mechanisms and one of the main characteristics the model must have is localisation: the events in the model should only focus on a node, and not use elements and/or variables leading to a global knowledge of the graph. Using functions are one of the best ways to localise an Event B model. And since the programming language used in ViSiDiA is *Java*, the events should be concrete: for example, the use of relations, set theory must be restricted in the guards and the actions; only uses that can be translated into simple boolean conditions (guards) or simple transformations of variables (actions) are allowed.
Our goal here is like the one described in the works of Mohamed Tounsi, Ahmed Hadj-Kacem, Mohamed Mosbah and Dominique Méry et al. ([14, 27]). We will apply the results of their works but also propose ideas:

1. First, we choose our concrete model **COLORING6** for the translation from Event B to ViSiDiA, because it is local,
2. then, we try to find rules for rewriting Events from our Event B model into algorithmic steps,
3. the steps are integrated into a skeleton of a classic ViSiDiA model, which is basically a Java Class,
4. we use the axioms, properties, theorems... defined in the contexts to draw an instatiation of the graph defined in the Event B model in the ViSiDiA GUI,
5. we animate and simulate the resulting algorithm using ViSiDiA.

Our ideas are summed up by the following figure (see figure 3.2):

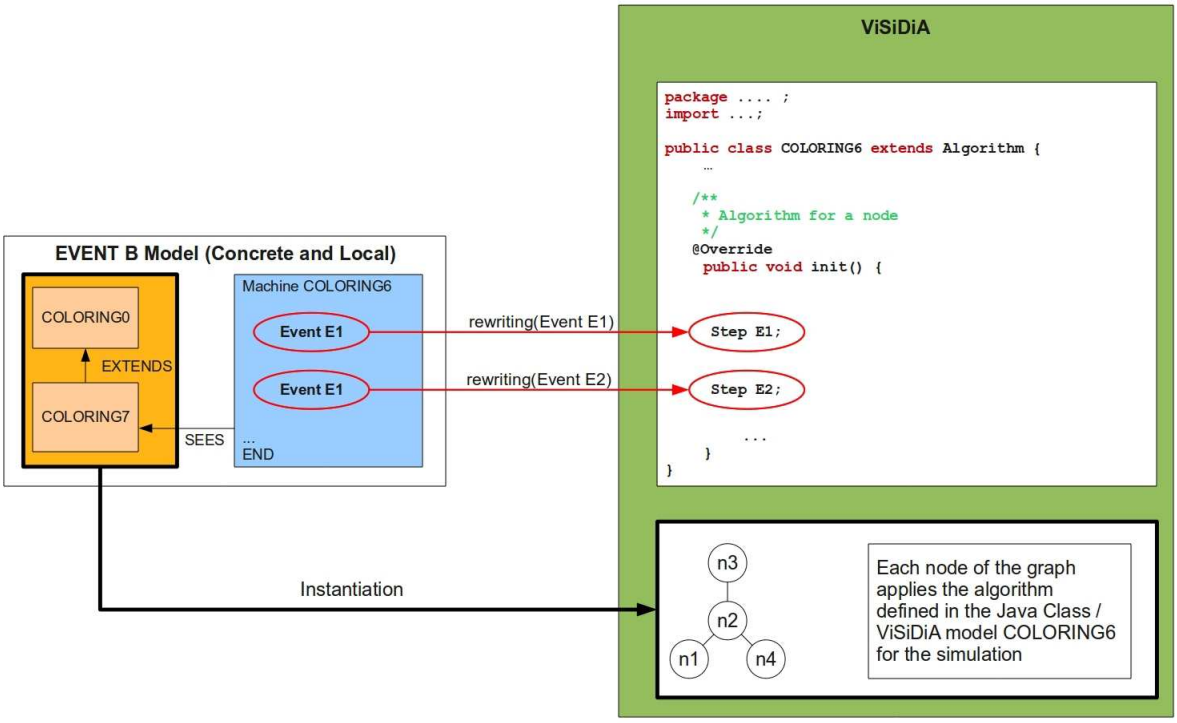


Figure 3.2: From an Event B model to a ViSiDiA model ([14])

3.2 Probabilities

One aspect of our work is to study how probabilities can appear into distributed algorithms and to take them into account during the process of refinement. It appears that several works related to this domain have already been done: we cite the semlinal works of McIver and Morgan [36, 34] and the approach of Hoang, Hallerstede, McIver and Morgan et al [20, 35, 23, 31] who are proposing an approach integrated to the Event B methodology.
Our goal here is to integrate probabilities into the graph coloring problem. Our analysis shows that probabilities in the graph coloring algorithm can be studied at three levels: first, at the level of each vertex, during the choice of a color and then, at the level of a ball when the vertices which composed it have to choose their colors.

3.2.1 Probabilities At The Level Of A Vertex

a Theory

The choice of a color c_n among a set C of colors by a vertex can be seen as a probabilistic event: in fact, the vertex is picking randomly a particular color c_n among the set CA of available colors, which is a subset of C. Each available color has the same chance to be picked by the vertex: consequently, we can say the choice of an available color c_n by a vertex follows the probability theory called " equiprobability ".
It should be noticed that during the execution of the algorithm, each vertex n of the graph is maintaining and updating a list L_n containing its active neighbours, i.e the neighbours which still have the same color as it. When an active neighbour takes a different color, the vertex n erases it from L_n and the color chosen by the neighbour is no more available for n: if n has to run its part of the algorithm once again, it will be forbidden for it to choose one of the colors picked by his neighbours which are not in its list L_n .
Therefore, we can say that the choice of a color by a vertex follows these probabilistic rules:

$$P(choice\ of\ an\ available\ color\ c_a\ by\ a\ vertex\ n) = \frac{1}{card(C_A)}$$
$$= \frac{1}{Total\ number\ of\ colors\ -\ Number\ of\ forbidden\ colors\ for\ n}$$
$$= \frac{1}{card(C)\ -\ card(CF|CF\ the\ set\ containing\ the\ colors\ chosen\ by\ the\ neighbours\ of\ n \notin L_n)}$$
$$P(choice\ of\ a\ forbidden\ color\ c_f\ by\ a\ vertex\ n) = 0$$

b Examples

We illustrate our theory through two examples, with a graph with three vertices labelled A, B, C and a set C of three colors R,G and B:

Example 1: In this case (figure 3.3), all the colors are available for each vertex, there is no restriction on the choices.

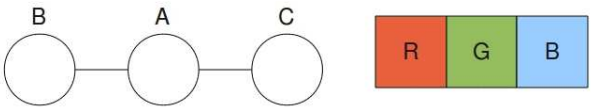


Figure 3.3: First case: no restriction on colors choices

Therefore, we have these probabilities for the choices of colors by the three vertices:

- Probabilities for the choice of a color by A:
  - $P(choice\ of\ R\ by\ A) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$
  - $P(choice\ of\ G\ by\ A) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$
  - $P(choice\ of\ B\ by\ A) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$

- Probabilities for the choice of a color by B:
  - $P(choice\ of\ R\ by\ B) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$
  - $P(choice\ of\ G\ by\ B) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$
  - $P(choice\ of\ B\ by\ B) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$

- Probabilities for the choice of a color by C:
  - $P(choice\ of\ R\ by\ C) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$
  - $P(choice\ of\ G\ by\ C) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$
  - $P(choice\ of\ B\ by\ C) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$

Example 2: In this case (figure 3.4), there is a restriction on the choices: B is no more an active neighbour of A, consequently A can not choose the color taken by B.

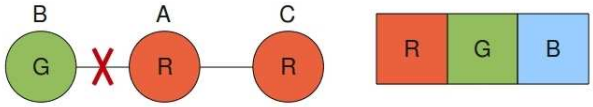


Figure 3.4: Second case: Existence of a forbidden color for a vertex

Therefore, we have these probabilities for the choices of colors by A and C:

- Probabilities for the choice of a color by A:
  - $P(choice\ of\ R\ by\ A) = \frac{1}{card(C) - card(CF)} = \frac{1}{card(\{R,G,B\}) - card(\{G\})} = \frac{1}{2}$
  - $P(choice\ of\ G\ by\ A) = 0$
  - $P(choice\ of\ B\ by\ A) = \frac{1}{card(C) - card(CF)} = \frac{1}{card(\{R,G,B\}) - card(\{G\})} = \frac{1}{2}$

- Probabilities for the choice of a color by C:
  - $P(choice\ of\ R\ by\ C) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$
  - $P(choice\ of\ G\ by\ C) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$
  - $P(choice\ of\ B\ by\ C) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$

3.2.2 Probabilities At The Level Of A *"Ball"*

a Theory

Our main interest here is to determine the probabilities that the vertices members of a ball are correctly colored in one-shot, i.e. i.e. that the vertices which are neighbours in the ball choose different colors. We can see that determining these probabilities is equivalent to the process of computing those that the central vertex n of the ball chooses a color different from the colors its $m+1$ neighbours nb_i, with $i = \{0..m\}$, may have chosen.
Therefore, we can say that the correct choice of colors by the members of a ball follows this probabilistic rule, which is very close to the conditional probability ones:

$$P(ball\ correctly\ colored\ in\ one\ shot) = \sum \left(P(n\ chooses\ a\ color\ c_n) \times \prod_{i=0}^{m} P\left(nb_i\ chooses\ a\ color\ different\ from\ c_n\right) \right)$$

As we have already said, computing the probabilities that a ball with a central vertex n is correctly colored in one-shot is equivalent to computing the probabilities that n chooses a color different from the colors its neighbours may have chosen. Therefore, we can say that if the event *"choice of a color by n"* is decomposed into two sub-events, an event *"choice of a right color by n"* (i.e. n chooses a color different from the ones its neighbours may have chosen), and another event *"choice of a wrong color by n"* (i.e. n chooses the same color as one/several of his neighbours may have chosen), the probabilities the ball is correctly colored in one-shot is equivalent to the probabilities that n activates the sub-event *"good choice of a color by n"*. Consequently, we can say the events *"good choice of a color by n"*, and *"bad choice of a color by n"* follow those probabilistic rules:

$$P(choice\ of\ a\ right\ color\ by\ a\ vertex\ n) = P(ball\ correctly\ colored\ in\ one\ shot)\ with\ n\ center\ of\ the\ ball$$

$$\begin{aligned} P(choice\ of\ a\ wrong\ color\ by\ a\ vertex\ n) &= P(\neg(choice\ of\ a\ right\ color\ by\ a\ vertex\ n)) \\ &= 1 - P(choice\ of\ a\ right\ color\ by\ a\ vertex\ n) \end{aligned}$$

b Examples

We illustrate our theory with the two previous examples:

Example 1: In this case, the ball is composed of the vertices A, B, C and the center of the ball is A.

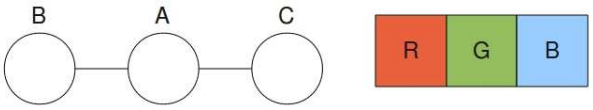


Figure 3.5: First example: no restriction on colors choices

As we have already said, the probabilities for choices of colors, in the case described by the figure 3.5, are the following:

- Probabilities for the choice of a color by A:
  - $P(choice\ of\ R\ by\ A) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$
  - $P(choice\ of\ G\ by\ A) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$
  - $P(choice\ of\ B\ by\ A) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$

- Probabilities for the choice of a color by B:
  - $P(choice\ of\ R\ by\ B) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$
  - $P(choice\ of\ G\ by\ B) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$
  - $P(choice\ of\ B\ by\ B) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$

- Probabilities for the choice of a color by C:
  - $P(choice\ of\ R\ by\ C) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$
  - $P(choice\ of\ G\ by\ C) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$
  - $P(choice\ of\ B\ by\ C) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$

The following tree (figure 3.6) shows us the combinations of the differents choices of colors the vertices can make and the probabilities related to them:

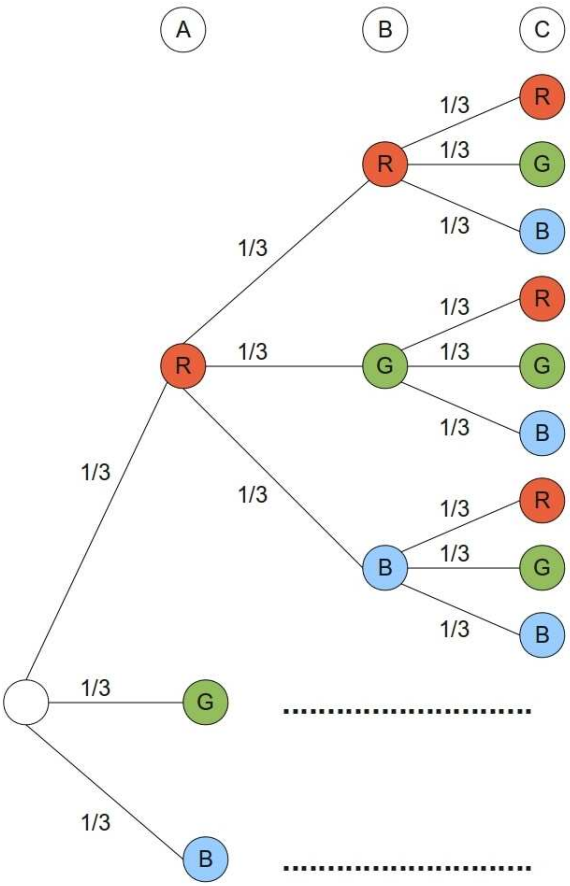


Figure 3.6: Probabilities for the first case

Among all the possibilities (there are *27* possibilities), only *12* of them are correct combinations. Therefore, we can say that the probability for the ball to be correctly colored in one-shot is:

$$P(ball\ correctly\ colored\ in\ one-shot) =$$
$$\sum_{i=1}^{12} \left(\begin{array}{l} P(A\ chooses\ a\ color) \times \\ P(B\ chooses\ a\ color\ different\ from\ the\ color\ of\ A) \times \\ P(C\ chooses\ a\ color\ different\ from\ the\ color\ of\ A) \end{array} \right) = \sum_{i=1}^{12} (\frac{1}{3} \times \frac{1}{3} \times \frac{1}{3}) = 12 \times \frac{1}{27} = \frac{12}{27}$$

$$\boxed{P(ball\ correctly\ colored\ in\ one-shot) = \frac{12}{27} = \frac{4}{9}}$$

We can see that the probabilities that the vertex A activates the events *"choice of a right color by A"* and *"choice of a wrong color by A"* are the following:

$$\boxed{P(choice\ of\ a\ right\ color\ by\ A) = P(ball\ correctly\ colored\ in\ one-shot) = \frac{12}{27} = \frac{4}{9}}$$

$$\boxed{P(choice\ of\ a\ wrong\ color\ by\ A) = 1 - P(ball\ correctly\ colored\ in\ one-shot) = 1 - \frac{12}{27} = 1 - \frac{4}{9} = \frac{5}{9}}$$

Example 2: In this case, the ball is composed of the vertices A, C and the center of the ball is A.

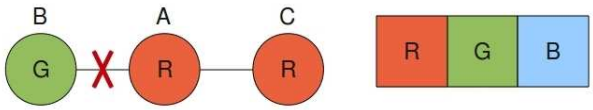


Figure 3.7: Second example: Existence of a forbidden color for a vertex

As we have already said, the probabilities for choices of colors, in the case described by the figure 3.7, are the following:

- Probabilities for the choice of a color by A:
  - $P(choice\ of\ R\ by\ A) = \frac{1}{card(C) - card(CF)} = \frac{1}{card(\{R,G,B\}) - card(\{G\})} = \frac{1}{2}$
  - $P(choice\ of\ G\ by\ A) = 0$
  - $P(choice\ of\ B\ by\ A) = \frac{1}{card(C) - card(CF)} = \frac{1}{card(\{R,G,B\}) - card(\{G\})} = \frac{1}{2}$

- Probabilities for the choice of a color by C:
  - $P(choice\ of\ R\ by\ C) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$
  - $P(choice\ of\ G\ by\ C) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$
  - $P(choice\ of\ B\ by\ C) = \frac{1}{card(C)} = \frac{1}{card(\{R,G,B\})} = \frac{1}{3}$

The following tree (figure 3.8) shows us the combinations of the differents choices of colors the vertices can make and the probabilities related to them:

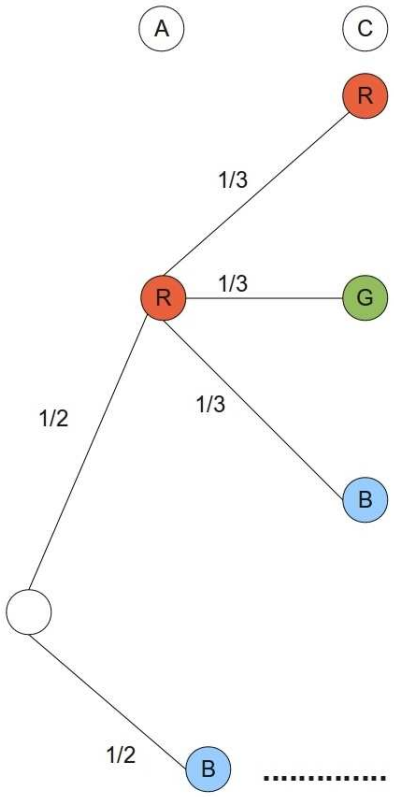


Figure 3.8: Probabilities for the second case

Among all the possibilities (there are *6* possibilities), *4* of them are correct combinations. Therefore, we can say that the probability for the ball to be correctly colored in one-shot is:

$$P(ball\ correctly\ colored\ in\ one-shot) =$$
$$\sum_{i=1}^{12} \left(\begin{array}{l} P(A\ chooses\ a\ color) * \\ P(C\ chooses\ a\ color\ different\ from\ the\ color\ of\ A) \end{array} \right) = \sum_{i=1}^{4} (\frac{1}{2} \times \frac{1}{3}) = 4 \times \frac{1}{6} = \frac{4}{6}$$

$$\boxed{P(ball\ correctly\ colored\ in\ one-shot) = \frac{4}{6} = \frac{2}{3}}$$

We can see that the probabilities that the vertex A activates the events *"choice of a right color by A"* and *"choice of a wrong color by A"* are the following:

$$\boxed{P(choice\ of\ a\ right\ color\ by\ A) = P(ball\ correctly\ colored\ in\ one-shot) = \frac{4}{6} = \frac{2}{3}}$$

$$\boxed{P(choice\ of\ a\ wrong\ color\ by\ A) = 1 - P(ball\ correctly\ colored\ in\ one-shot) = 1 - \frac{4}{6} = 1 - \frac{2}{3} = \frac{1}{3}}$$

3.2.3 Perspectives: Use Of Probabilistic Formal Verification Tools

We can observe that the probabilities in graph coloring algorithms lie within the choices of colors by vertices. Two ways of handling probabilities can be studied:

1. The first one is focused on colors available for a vertex: The probabilities are distributed over the set of available colors a vertex can choose.
2. The second one is focused on the event of the choice of a color by a vertex: we can say that the vertex has a certain probability of triggering an event called *"wrong choice"*, in which it chooses a color which is the same as one of its neighbours, and another probability of triggering an event called *"good choice"* , in which it chooses a color different from the colors its neighbours has chosen.

The next step is to transform the local and concrete model **COLORING6**: the non-deterministic choice of a color by a node are replaced using these two ways as illustrated by the following figures (see figure 3.10 for the probabilistic choice between two events *"wrong choice"* and *"good choice"* and figure 3.9 for the probabilistic choice of a color among a set of available ones):

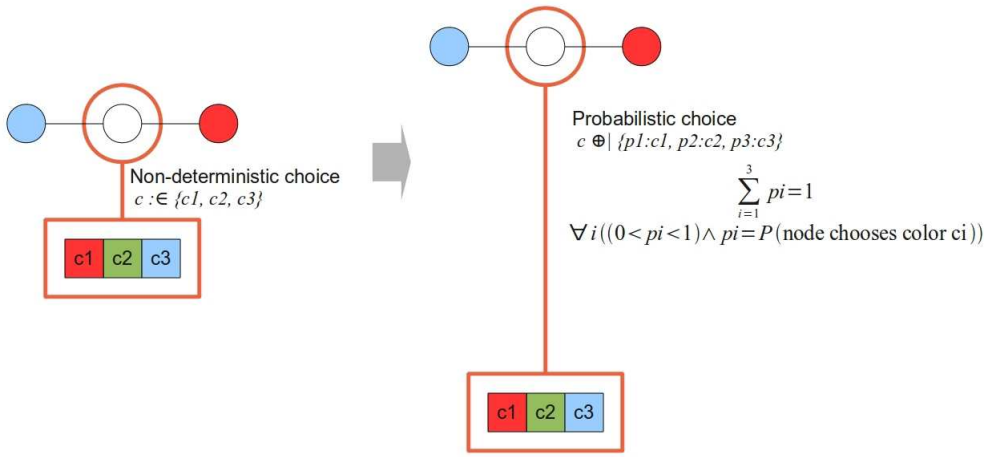


Figure 3.9: Probabilistic choice at the level of a node

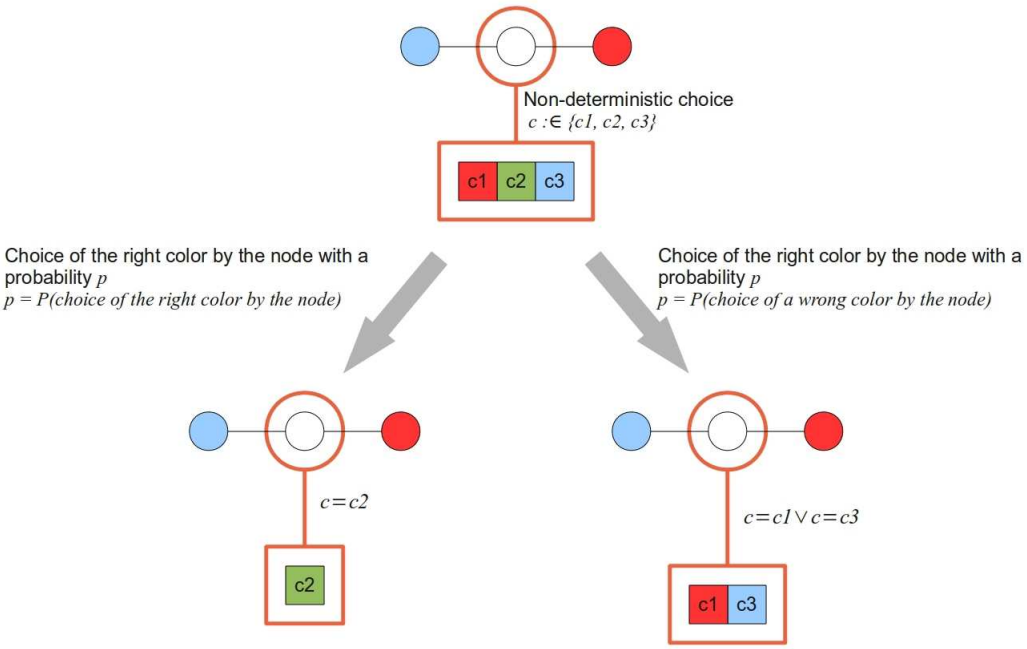


Figure 3.10: Probabilistic choice at the level of an event

We choose here to apply the transformation described by the figure 3.9 because we have a local algorithm defined in **COLORING6** and we wish only to focus on the choice of a node only, not on its choice related to the choices of its neighbours. The next step is then to analyse the resulting models (see figure 3.11) after this transformation using probabilistic tools like for instance PRISM ([26]) that was already used in the master thesis of the author for dealing with the tree identification protocol IEEE 1394 ([44, 43, 10]).

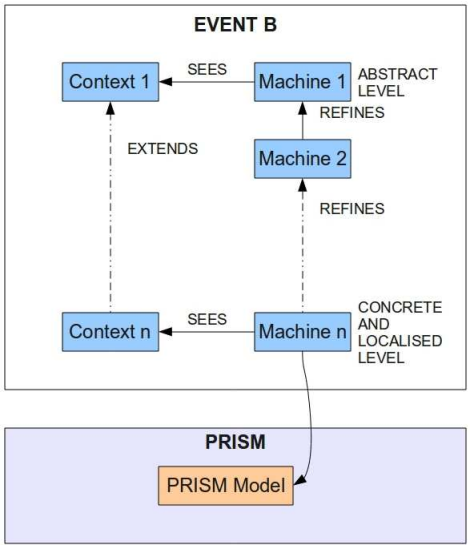


Figure 3.11: A way to handle probabilistic arguments

We propose a way to translate Event B models into PRISM ones:

1. the local and concrete machine **COLORING6** is used for the translation,

2. We choose the behaviour of our PRISM model: it is the one defined by MDP (Markov Decision Processes) which combines discrete probabilistic behaviour and non-determinism,

3. the graph and the set of colors are instantiated,

4. since the algorithm defined by **COLORING6** is local (centered on a node), each module of the PRISM model corresponds to a node,

5. the events in **COLORING6** is translated into PRISM commands, with a version for each node, with respect to their environment (neighbours, colors available for choice...). The events with non-deterministical choices (GENERATE_AND_SEND_COLOR) are translated into commands with probabilistic choices using the method described in the figure 3.9,

6. then, the model-checker PRISM is used to analyse and check properties of the model, like probabilistic termination, safety properties, ...

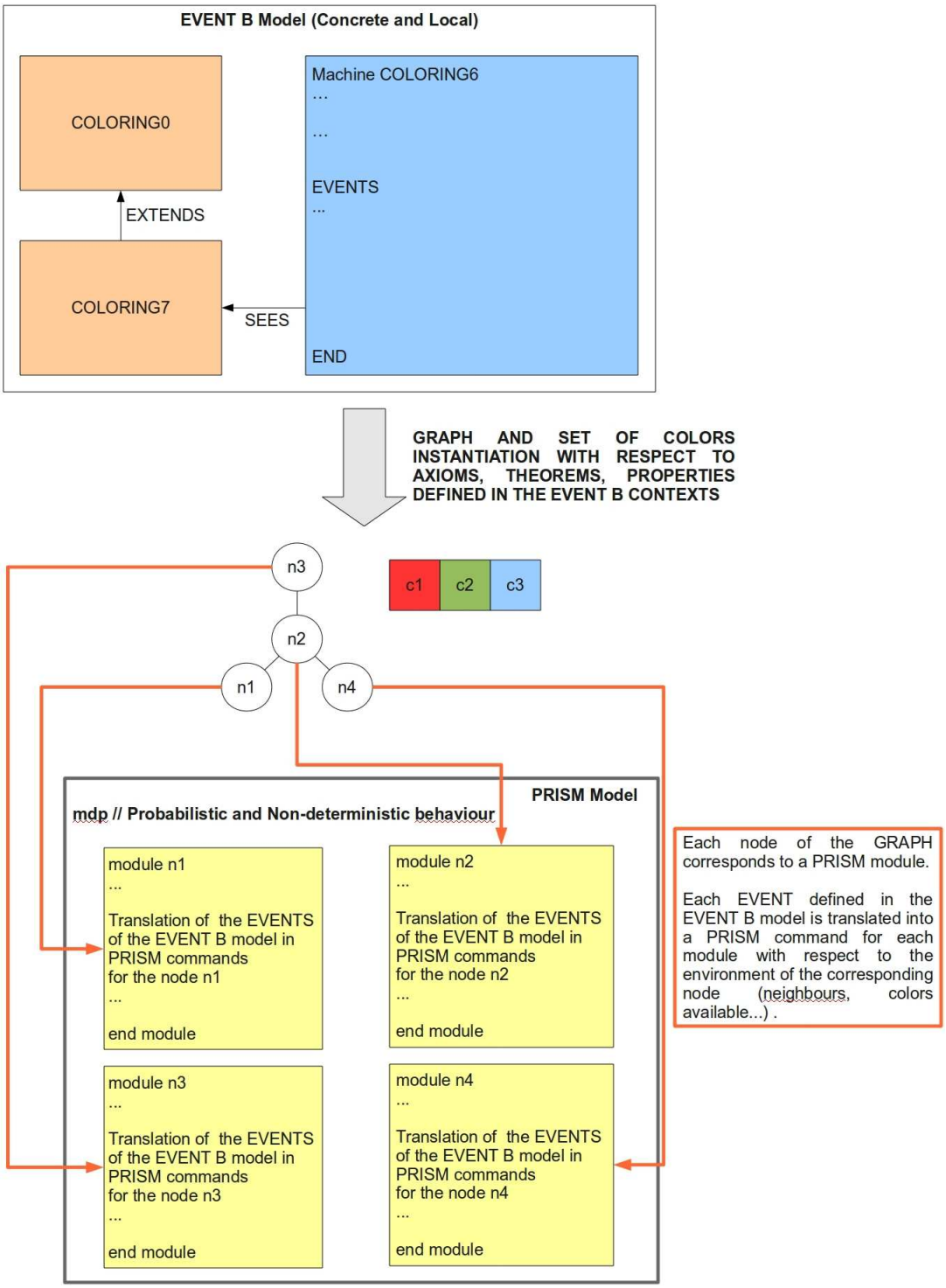


Figure 3.12: Translation from Event B to PRISM

Chapter 4

Conclusion and Future Work

The Event-B method generates proof obligations. The summary of these proof obligations discharged either automatically or interactively is a measure of the complexity of the development itself:

Model	Total	Auto		Interactive	
COLORING0	0	0	0.00%	0	0.00%
COLORING7	2	2	100.00%	0	0.00%
COLORING1	3	2	66.67%	1	33.33%
COLORING2	13	10	76.92%	3	23.08%
COLORING3	23	16	69.57%	7	30.43%
COLORING4	25	15	60.00%	10	40.00%
COLORING5	20	16	80.00%	4	20.00%
COLORING6	150	142	94.66%	8	5.33%

Table 4.1: The proof obligations for the vertex coloring algorithms

The current stepwise development focuses on vertex coloring algorithms wich require probabilistic arguments to achieve termination. They have in common the integration of possible errors during the choice of colors by vertices and this is the point where probabilistic arguments can be handled. As we have mentioned in the abstract, this paper presents preliminary elements of a global methodology for handling the correct-by-construction refinement-based approach applied to distributed algorithms and further work is needed to adapt technique and tools. It is also a first step towards our goal to obtaining a development framework, integrating probabilistic arguments and refinement, for distributed algorithms. Our work here focuses on the combination (see figure 4.1) of the refinement provided by the Event B framework and the probabilities handling provided by another formal tool, the model-checker PRISM ([26]).

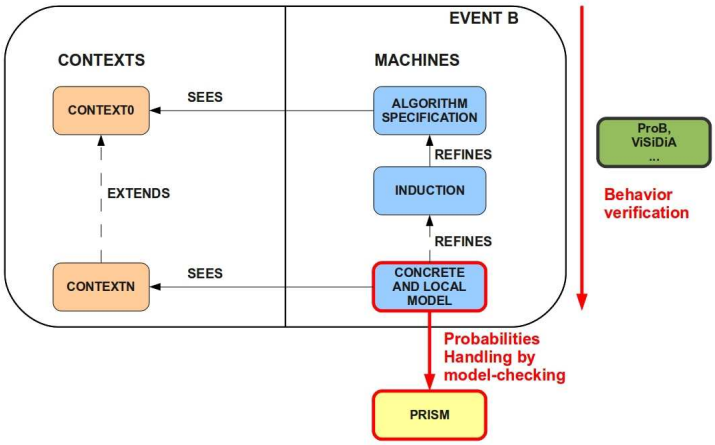


Figure 4.1: Our current framework for the development of distributed algorithms

Handling the probabilities through model-checking is interesting, but what we want to do, as a future work, is to integrate probabilities handling into our Event B modelling, that is extending the Event B Framework ([20, 35, 36, 23, 31, 34]) in such a way, that it can deal with probabilistic arguments (see figure 4.2) through probabilistic refinement.

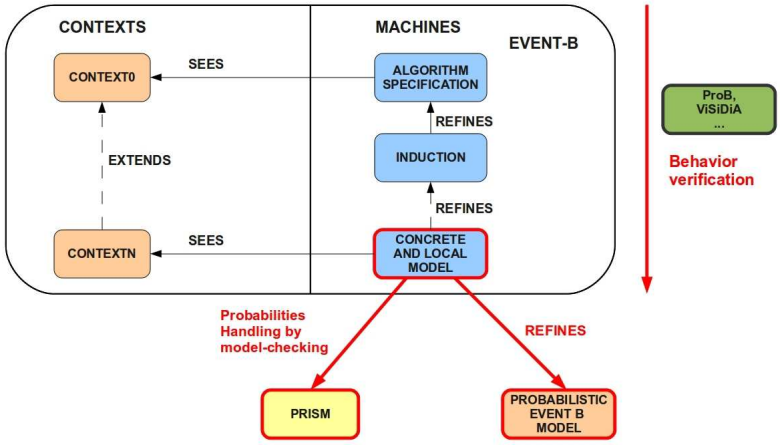


Figure 4.2: Extension of our current framework for the development of distributed algorithms

Bibliography

[1] IEEE Standard for a High-Performance Serial Bus. *IEEE Std 1394-1995*, August 1996.

[2] Technology Briefing Report - System Modelling. `http://www.comp.lancs.ac.uk/projects/renaissance/RenaissanceWeb/project/Acrobat/System_Modelling.pdf`, 1996.

[3] J. R. Abrial. *The B-Book. Assigning programs to meaning*. Cambridge University Press, 1996.

[4] J.-R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias, editor, *Proc. 1st Conf. on the B Method*, pages 169–190. IRIN Institut de recherche en informatique de Nantes, 1996.

[5] J.-R. Abrial. A system development process with event-b and the rodin platform. In *Proceedings of the formal engineering methods 9th international conference on Formal methods and software engineering*, ICFEM'07, pages 1–3, Berlin, Heidelberg, 2007. Springer-Verlag.

[6] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, first edition, June 2010.

[7] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *STTT*, 12(6):447–466, 2010.

[8] Jean-Raymond Abrial and Dominique Cansell. Click'n'Prove: Interactive Proofs Within Set Theory. In David Basin et Burkhart Wolff, editor, *16th International Conference on Theorem Proving in Higher Order Logics - TPHOLs'2003*, volume 2758 of *Lecture notes in Computer Science*, pages 1–24, Rome, Italy, 2003. Springer. Colloque avec actes et comité de lecture. internationale.

[9] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of ieee 1394 tree identify protocol. *Formal Asp. Comput.*, 14(3):215–227, 2003.

[10] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A new ieee 1394 leader election protocol. *Rigorous Methods for Software Construction and Analysis Seminar N06161,07.05.-12.05.06. Schloss Dagstuhl, U.Glaser and J. Abrial*, 2006.

[11] Jean-Raymond Abrial and Louis Mussat. Introducing dynamic constraints in b. In *B*, pages 83–128, 1998.

[12] Cédric Aguerre. ViSiDiA - user manual. `http://visidia.labri.fr/doc/user_manual/user_manual.pdf`, 2010.

[13] Noga Alon and Nabil Kahale. A spectral technique for coloring random 3-colorable graphs. *SIAM J. Comput.*, 26:1733–1748, December 1997.

[14] Pierre Castéran, Vincent Filou, and Mohamed Mosbah. Formal Proofs of Local Computation Systems. Technical Report 0, LaBRI - University of Bordeaux, 2009.

[15] Jérémie Chalopin, Emmanuel Godard, Yves Métivier, and Akka Zemmari. Autour des algorithmes distribués. LaBri - Laboratoire Bordelais de Recherche en Informatique, 2011.

[16] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. The tla$^+$ proof system: Building a heterogeneous verification platform. In Ana Cavalcanti, David Déharbe, Marie-Claude Gaudel, and Jim Woodcock, editors, *ICTAC*, volume 6255 of *Lecture Notes in Computer Science*, page 44. Springer, 2010.

[17] Patrick Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.

[18] Bilel Derbel. A brief introduction to visidia. `http://www2.lifl.fr/~derbel/resources_visidia/visidia_tutorial.pdf`, 2007.

[19] K. R. Duffy, N. O'Connell, and A. Sapozhnikov. Complexity analysis of a decentralised graph colouring algorithm. *Inf. Process. Lett.*, 107:60–63, July 2008.

[20] Stefan Hallerstede and Thai Son Hoang. Qualitative probabilistic modelling in event-b. In *IFM*, pages 293–312, 2007.

[21] Jennie C. Hansen, Marek Kubale, Lukasz Kuszner, and Adam Nadolski. Distributed largest-first algorithm for graph coloring. In *Euro-Par*, pages 804–811, 2004.

[22] Lehning Hervé. Téléphonie cellulaire - coloriages pour allocation de fréquences. *La Recherche - N.390*, pages 98–99, 2005.

[23] Thai Son Hoang, Zhendong Jin, Ken Robinson, Annabelle McIver, and Carroll Morgan. Development via Refinement in Probabilistic B — Foundation and Case Study. In Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors, *ZB2005: Formal Specification and Development in Z and B, Proceedings of the 4th International Conference of B and Z Users*, volume 3455 of *Lecture Notes in Computer Science*, pages 355–373, Guildford, United Kingdom, April 2005. Springer Verlag.

[24] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[25] Marek Kubale and Lukasz Kuszner. A better practical algorithm for distributed graph coloring. In *PARELEC*, pages 72–75, 2002.

[26] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, volume 2324 of *LNCS*, pages 200–204. Springer, 2002.

[27] Laboratoire LABRI. B2visidia. `http://www.labri.fr/projet/visidia`, 2010.

[28] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[29] Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[30] Mohammad Malkawi, Mohammad Al-Haj Hassan, and Osama Al-Haj Hassan. A new exam scheduling algorithm using graph coloring. *Int. Arab J. Inf. Technol.*, 5(1):80–86, 2008.

[31] Annabelle McIver, Carroll Morgan, and Thai Son Hoang. Probabilistic termination in b. In Didier Bert, Jonathan P. Bowen, Steve King, and Marina A. Waldén, editors, *ZB*, volume 2651 of *Lecture Notes in Computer Science*, pages 216–239. Springer, 2003.

[32] Dominique Méry, Mohamed Mosbah, and Mohamed Tounsi. Refinement-based verification of local synchronization algorithms. In Michael Butler and Wolfram Schulte, editors, *FM*, volume 6664 of *Lecture Notes in Computer Science*, pages 338–352. Springer, 2011.

[33] Dominique Méry, Mohammed Mosbah, and Mohammed Tounsi. Proving Distributed Algorithms by Combining Refinement and Local Computations. In Jens Bendisposto, Michael Leuschel, and Markus Roggenbach, editors, *AVOCS 2010 10th International Workshop on Automated Verification of Critical Systems*, Dusseldorf, Germany, September 2010.

[34] Carroll Morgan. The generalised substitution language extended to probabilistic programs. In Didier Bert, editor, *B*, volume 1393 of *Lecture Notes in Computer Science*, pages 9–25. Springer, 1998.

[35] Carroll Morgan, Thai Son Hoang, and Jean-Raymond Abrial. The challenge of probabilistic *event b* - extended abstract. In *ZB*, pages 162–171, 2005.

[36] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Trans. Program. Lang. Syst.*, 18(3):325–353, 1996.

[37] J. Mycielski. Sur le coloriage des graphes. *Colloq. Math. 3*, pages 161–162, 1955.

[38] Dominique Méry. Systèmes répartis, algorithmes répartis, programmation distribuée. `www.loria.fr/~mery/aspd/lncs_aspd1.pdf`, 2005.

[39] Dominique Méry. Refinement-based guidelines for constructing algorithms. In Jean-Raymond Abrial, Michael Butler, Rajev Joshi, Elena Troubitsyna, and Jim C. P. Woodcock, editors, *Refinement Based Methods for the Construction of Dependable Systems*, number 09381 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[40] Y. Métivier, J.M. Robson, N. Saheb-Djahromi, and A. Zemmari. An analysis of an optimal bit complexity randomised distributed vertex colouring algorithm (extended abstract). *OPODIS*, pages 359–364, 2009.

[41] Brian R. Nickerson. Graph coloring register allocation for processors with multi-register operands. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 40–52, New York, NY, USA, 1990. ACM.

[42] Lawrence C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[43] Joris Rehm. *Gestion Du Temps Par Le Raffinement*. PhD thesis, Université Henri Poincaré - Nancy 1, 2009.

[44] Joris Rehm and Dominique Cansell. Proved development of the real-time properties of the ieee 1394 root contention protocol with the event b method. In *ISoLA*, pages 179–190, 2007.

[45] Benoît Robillart. *Vérification formelle et optimisation de l'allocation de registres*. PhD thesis, Conservatoire National des Arts et Métiers, 2010.

[46] Johannes Schneider and Roger Wattenhofer. A new technique for distributed symmetry breaking. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 257–266, New York, NY, USA, 2010. ACM.

[47] Afif Sellami. *Des calculs locaux aux algorithmes distribués*. PhD thesis, Université Bordeaux I, Talence, France, 2004.

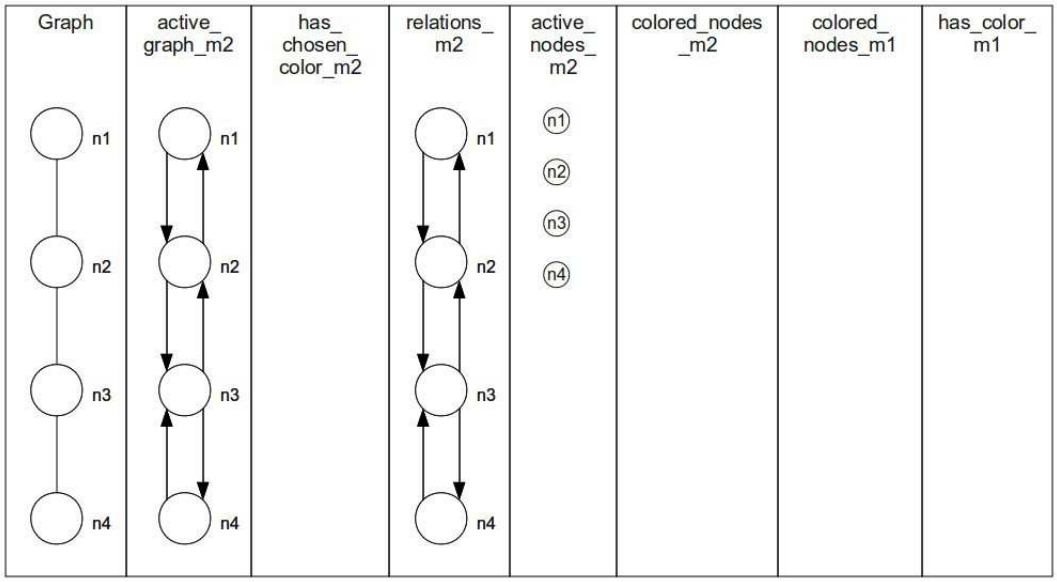
[48] Mohamed Tounsi, Ahmed Hadj-Kacem, Mohamed Mosbah, and Dominique Méry. A Refinement Approach for Proving Distributed Algorithms : Examples of Spanning Tree Problems. In *Integration of Model based Formal Methods and Tools(IMFMT 2009)*, Düsseldorf Allemagne, 2009.

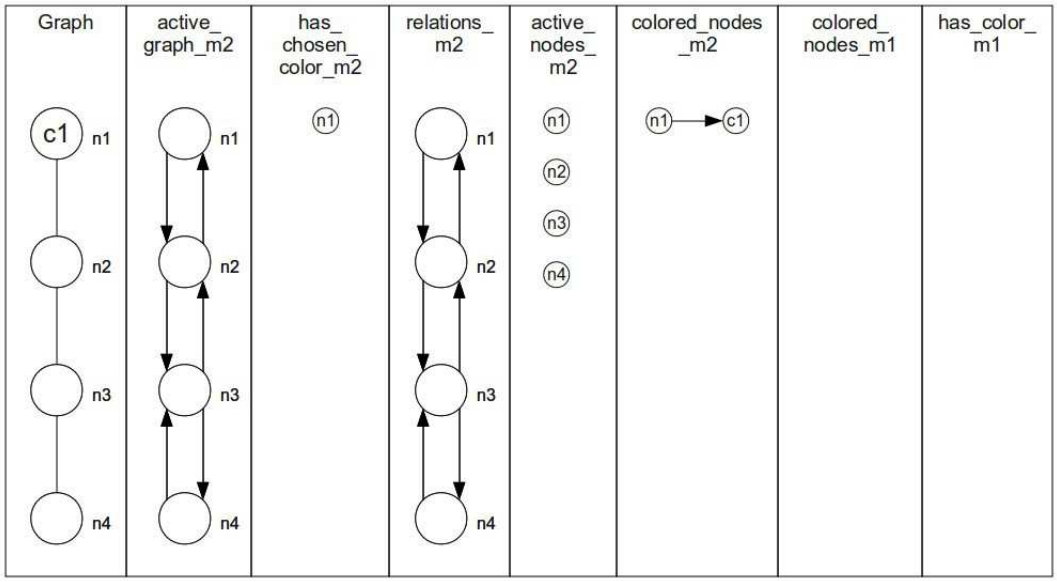
[49] E.-G. Villegas, E. Lopez-Aguilera, R. Vidal, and Paradells J. Effect of adjacent-channel interference in IEEE 802.11 WLANs. *Cognitive Radio Oriented Wireless Networks and Communications, 2007. CrownCom 2007. 2nd International Conference*, pages 118 – 125, August 2007.

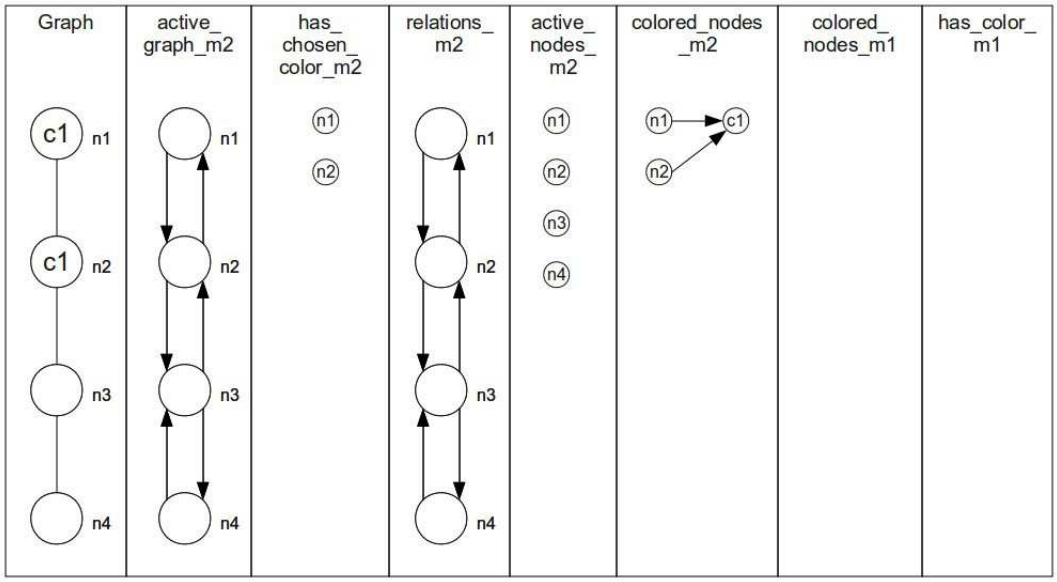
Appendices

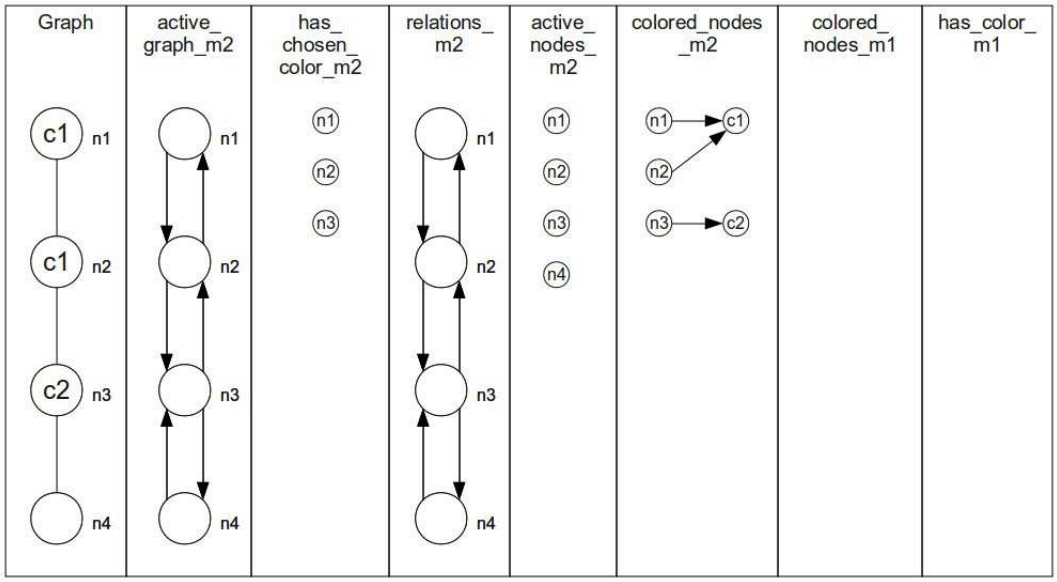
Appendix A

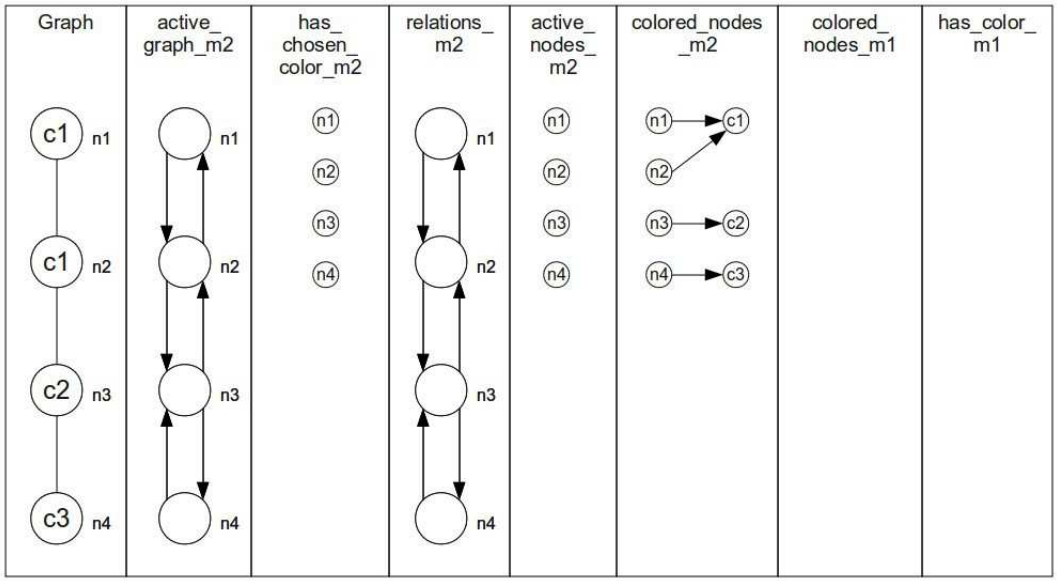
Behaviour Of The Synchronous Algorithm

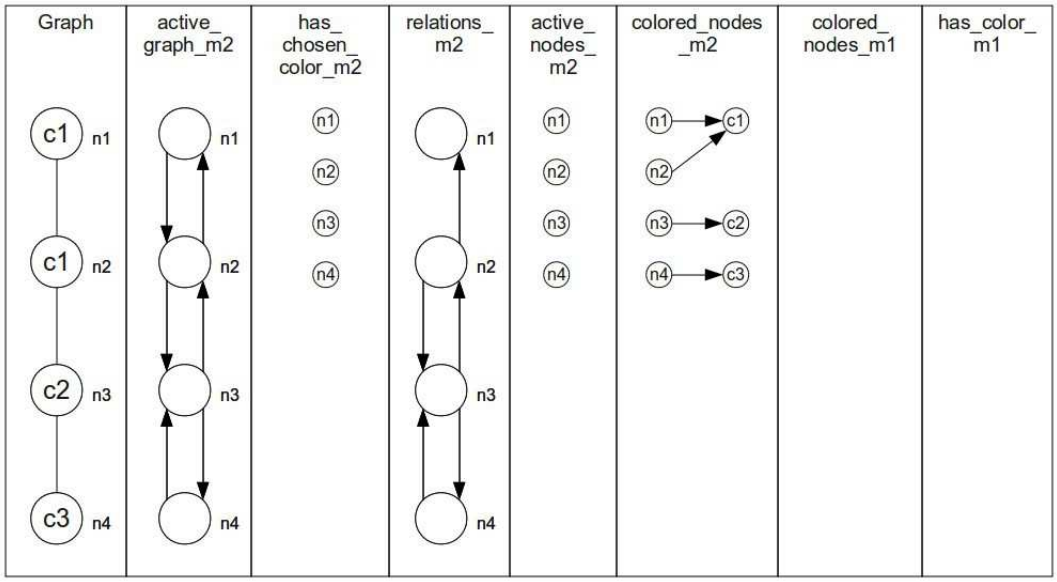

Graph
active_ graph_m2
has_ chosen_ color_m2
relations_ m2
active_ nodes_ m2
colored_nodes _m2
colored_ nodes_m1
has_color_ m1
n1
n2
n3
n4

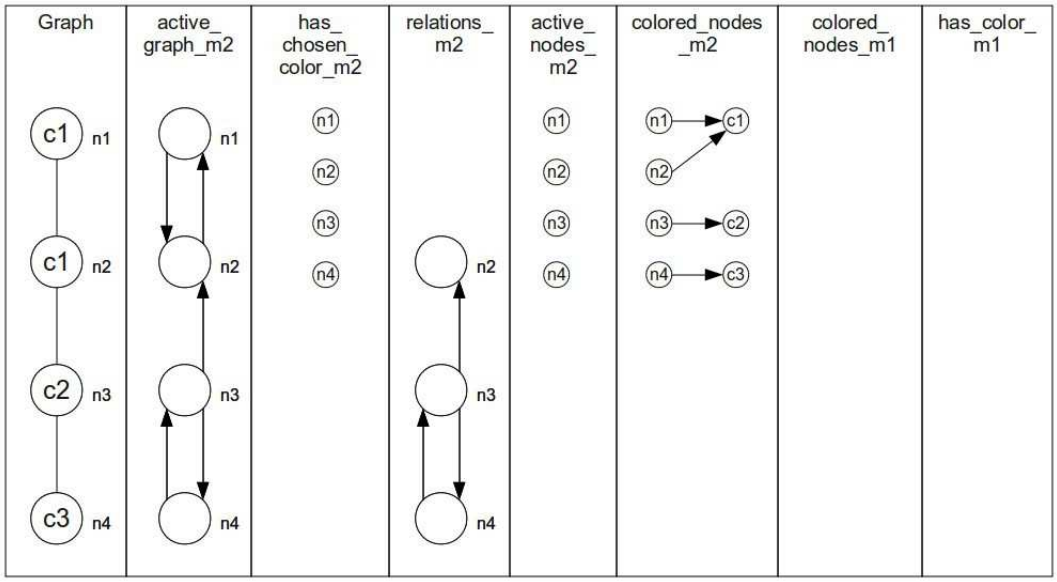

Graph
active_ graph_m2
has_ chosen_ color_m2
relations_ m2
active_ nodes_ m2
colored_nodes _m2
colored_ nodes_m1
has_color_ m1
c1
n1
n2
n3
n4

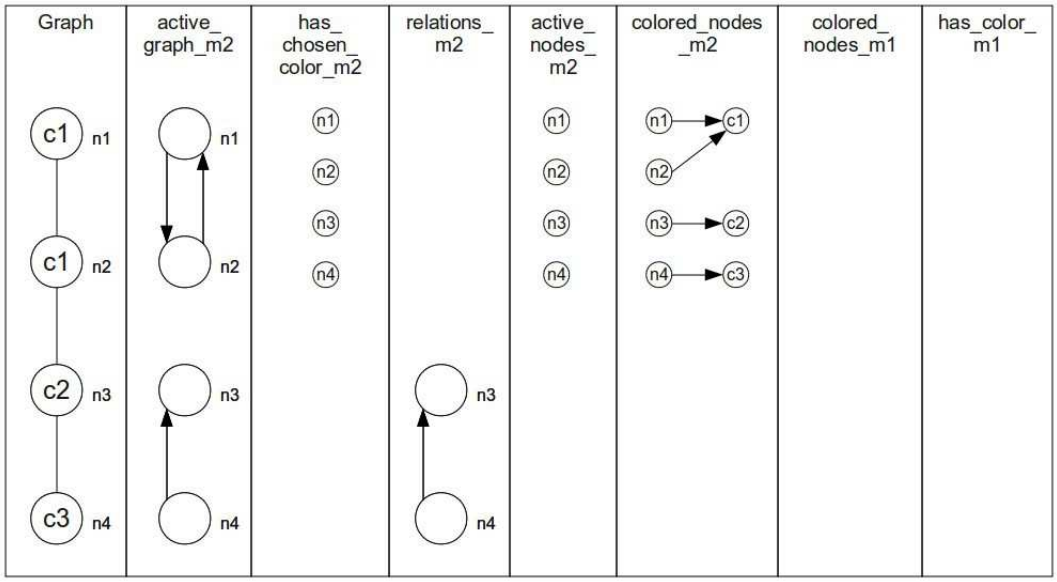

Graph
active_ graph_m2
has_ chosen_ color_m2
relations_ m2
active_ nodes_ m2
colored_nodes _m2
colored_ nodes_m1
has_color_ m1
c1
n1
c1
n2
n3
n4
n1
n2
n3
n4
n1
n2
n1
n2
n3
n4
n1
n2
n3
n4
n1
c1
n2

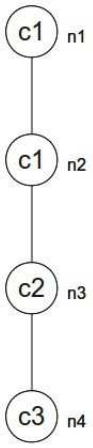
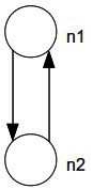
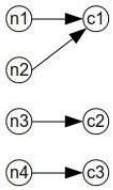

Graph
active_ graph_m2
has_ chosen_ color_m2
relations_ m2
active_ nodes_ m2
colored_nodes _m2
colored_ nodes_m1
has_color_ m1
c1
n1
c1
n2
c2
n3
n4
n1
n2
n3
n4
n1
n2
n3
n1
n2
n3
n4
n1
c1
n2
n3
c2

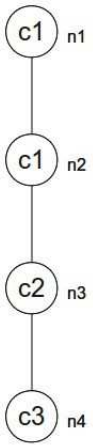
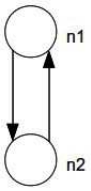
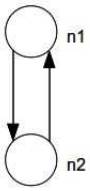
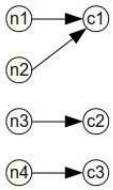

Graph
active_ graph_m2
has_ chosen_ color_m2
relations_ m2
active_ nodes_ m2
colored_nodes _m2
colored_ nodes_m1
has_color_ m1
c1
n1
c1
n2
c2
n3
c3
n4
n1
n2
n3
n4
n1
n2
n3
n4
n1
n2
n3
n4
n1
n2
n3
n4
n1
c1
n2
c1
n3
c2
n4
c3

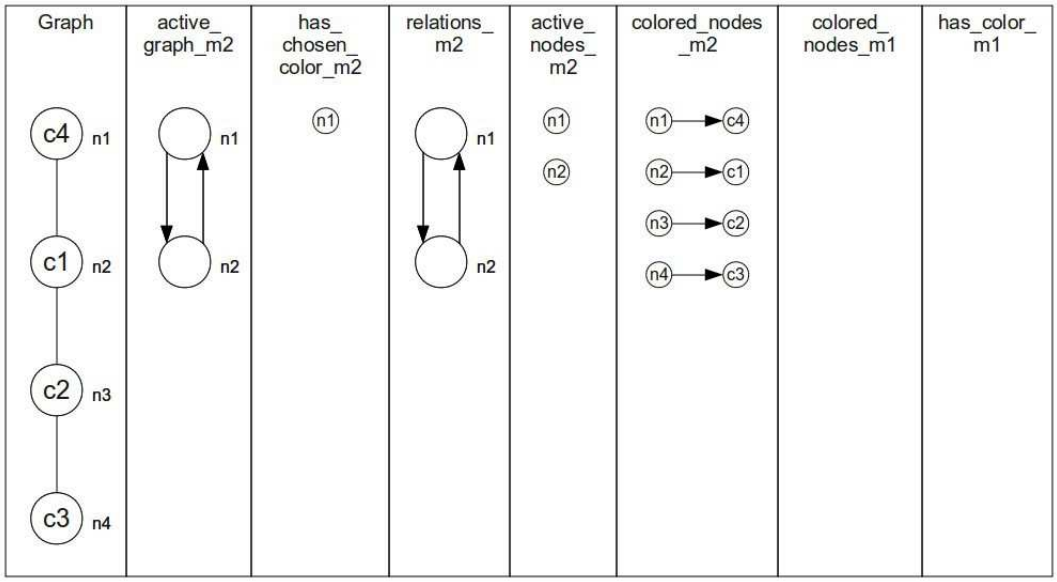

Graph
active_ graph_m2
has_ chosen_ color_m2
relations_ m2
active_ nodes_ m2
colored_nodes _m2
colored_ nodes_m1
has_color_ m1
c1
n1
c1
n2
c2
n3
c3
n4
n1
n2
n3
n4
n1
n2
n3
n4
n1
n2
n3
n4
n1
c1
n2
c1
n3
c2
n4
c3

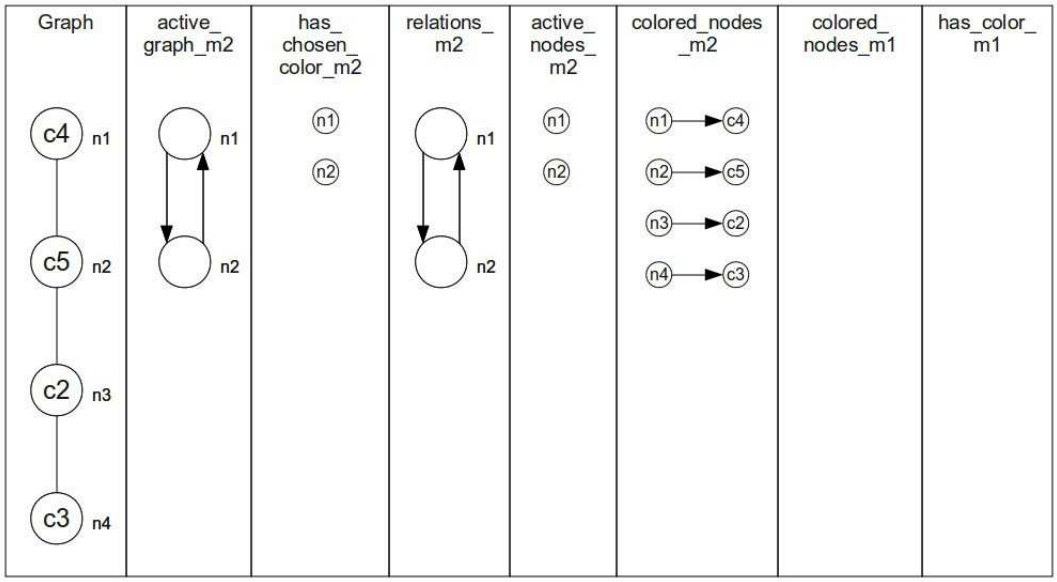

Graph
active_ graph_m2
has_ chosen_ color_m2
relations_ m2
active_ nodes_ m2
colored_nodes _m2
colored_ nodes_m1
has_color_ m1
c1
n1
c1
n2
c2
n3
c3
n4
n1
n2
n3
n4
n1
n2
n3
n4
n2
n3
n4
n1
n2
n3
n4
n1
c1
n2
c1
n3
c2
n4
c3

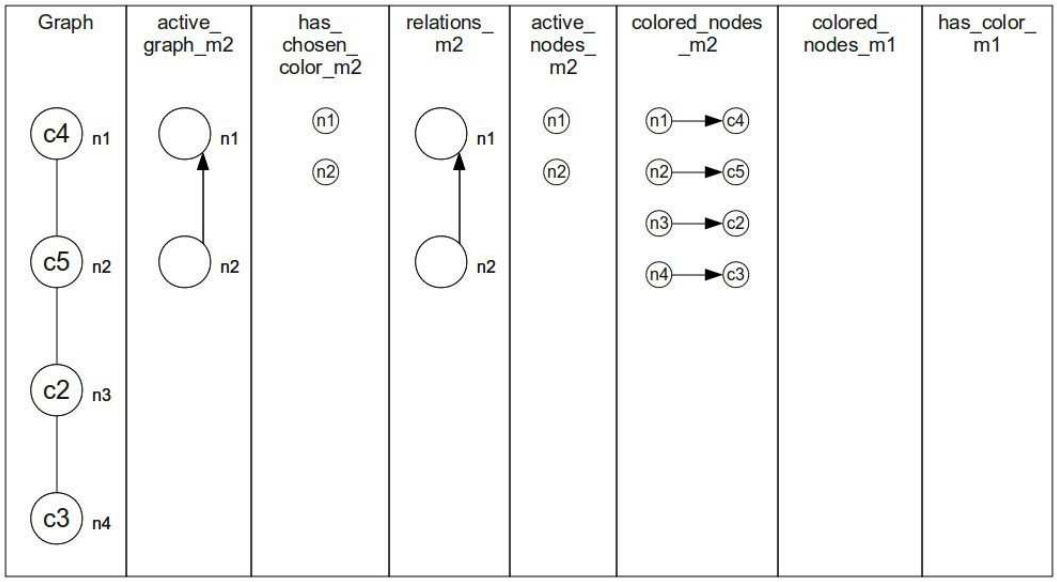

Graph
active_ graph_m2
has_ chosen_ color_m2
relations_ m2
active_ nodes_ m2
colored_nodes _m2
colored_ nodes_m1
has_color_ m1
c1
n1
c1
n2
c2
n3
c3
n4
n1
n2
n3
n4
n1
n2
n3
n4
n3
n4
n1
n2
n3
n4
n1
c1
n2
c1
n3
c2
n4
c3

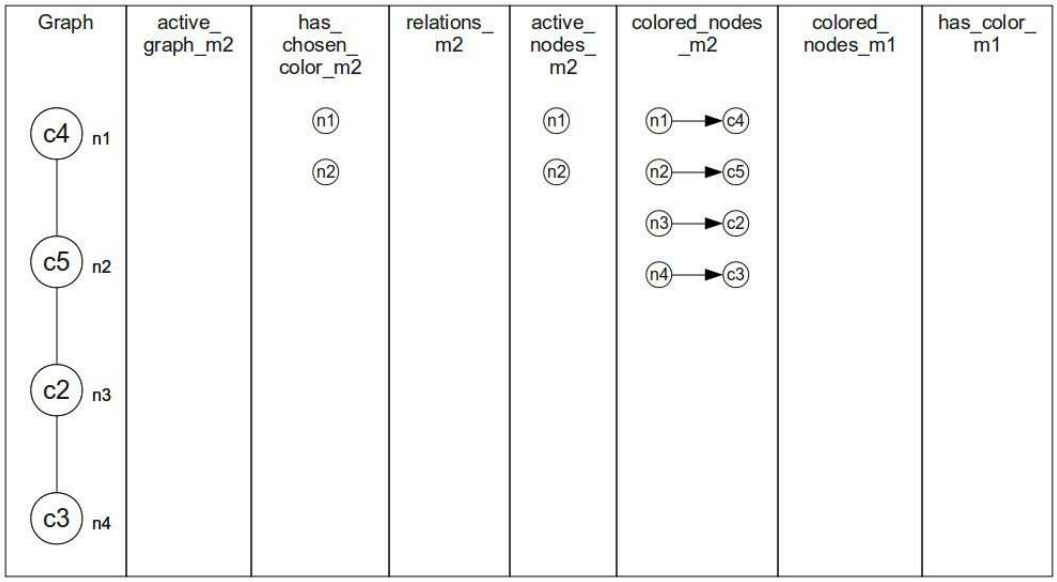
Graph	active_ graph_m2	has_ chosen_ color_m2	relations_ m2	active_ nodes_ m2	colored_nodes _m2	colored_ nodes_m1	has_color_ m1
							

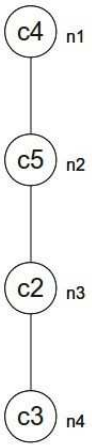
Graph	active_ graph_m2	has_ chosen_ color_m2	relations_ m2	active_ nodes_ m2	colored_nodes _m2	colored_ nodes_m1	has_color_ m1
							

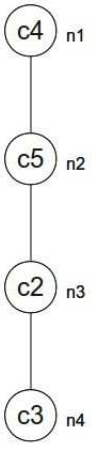

Graph
active_ graph_m2
has_ chosen_ color_m2
relations_ m2
active_ nodes_ m2
colored_nodes _m2
colored_ nodes_m1
has_color_ m1
c4
n1
c1
n2
c2
n3
c3
n4
n1
n2
n1
n2
n1
n2
n1
c4
n2
c1
n3
c2
n4
c3

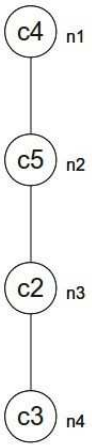
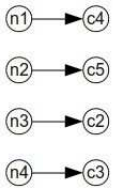
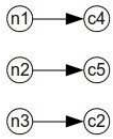

Graph
active_ graph_m2
has_ chosen_ color_m2
relations_ m2
active_ nodes_ m2
colored_nodes _m2
colored_ nodes_m1
has_color_ m1
c4
n1
c5
n2
c2
n3
c3
n4
n1
n2
n1
n2
n1
n2
n1
c4
n2
c5
n3
c2
n4
c3

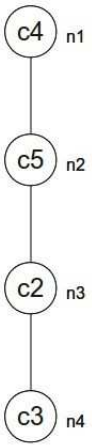
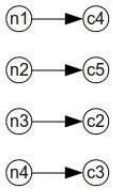
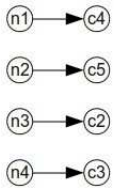

Graph
active_ graph_m2
has_ chosen_ color_m2
relations_ m2
active_ nodes_ m2
colored_nodes _m2
colored_ nodes_m1
has_color_ m1
c4
n1
c5
n2
c2
n3
c3
n4
n1
n2
n1
n2
n1
n2
n1
c4
n2
c5
n3
c2
n4
c3


Graph
active_ graph_m2
has_ chosen_ color_m2
relations_ m2
active_ nodes_ m2
colored_nodes _m2
colored_ nodes_m1
has_color_ m1
c4
n1
c5
n2
c2
n3
c3
n4
n1
n2
n1
n2
n1
c4
n2
c5
n3
c2
n4
c3

Graph	active_ graph_m2	has_ chosen_ color_m2	relations_ m2	active_ nodes_ m2	colored_nodes _m2	colored_ nodes_m1	has_color_ m1
		n1 n2		n1 n2	n1→c4 n2→c5 n3→c2 n4→c3	n1→c4	n1

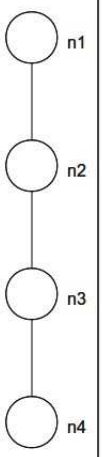
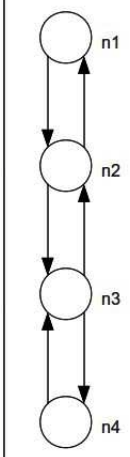
Graph	active_ graph_m2	has_ chosen_ color_m2	relations_ m2	active_ nodes_ m2	colored_nodes _m2	colored_ nodes_m1	has_color_ m1
		n1 n2		n1 n2	n1→c4 n2→c5 n3→c2 n4→c3	n1→c4 n2→c5	n1 n2

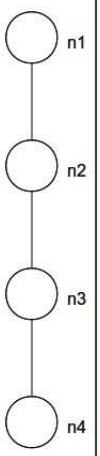
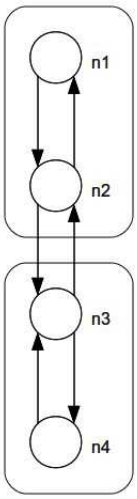
Graph	active_ graph_m2	has_ chosen_ color_m2	relations_ m2	active_ nodes_ m2	colored_nodes _m2	colored_ nodes_m1	has_color_ m1
							

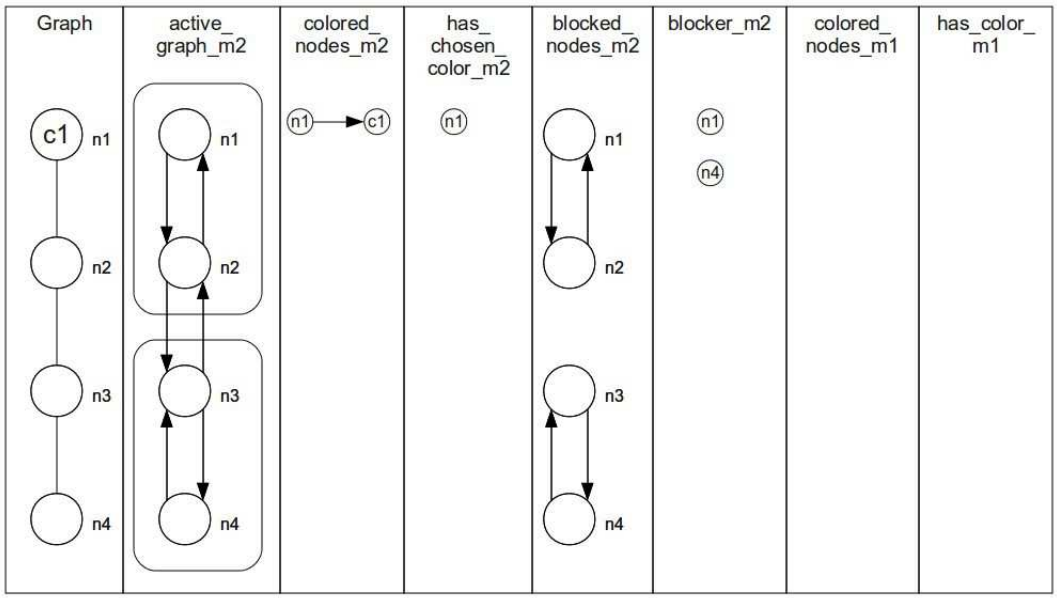
Graph	active_ graph_m2	has_ chosen_ color_m2	relations_ m2	active_ nodes_ m2	colored_nodes _m2	colored_ nodes_m1	has_color_ m1
							

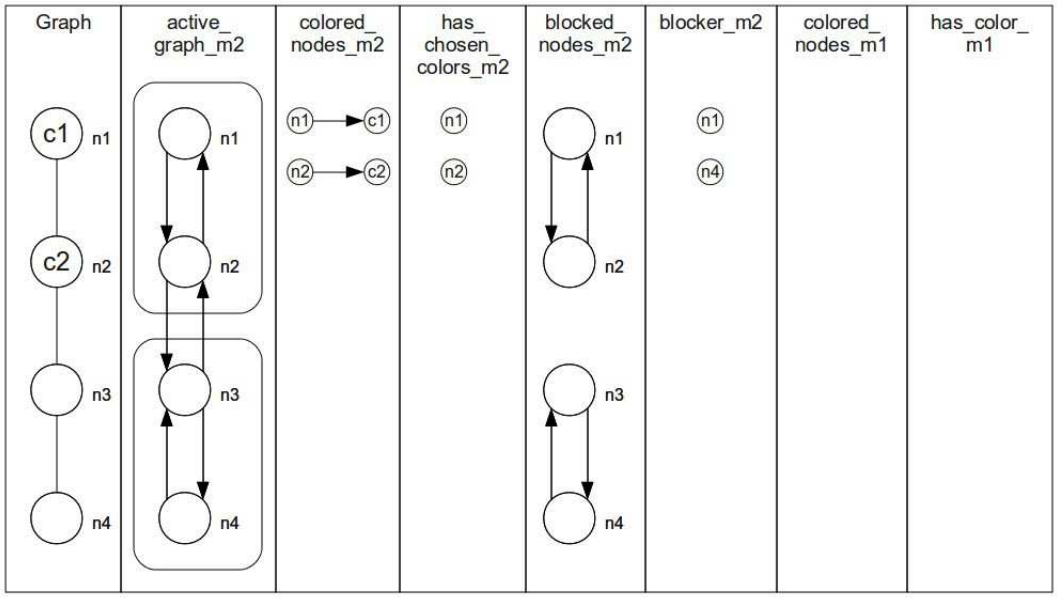
Appendix B

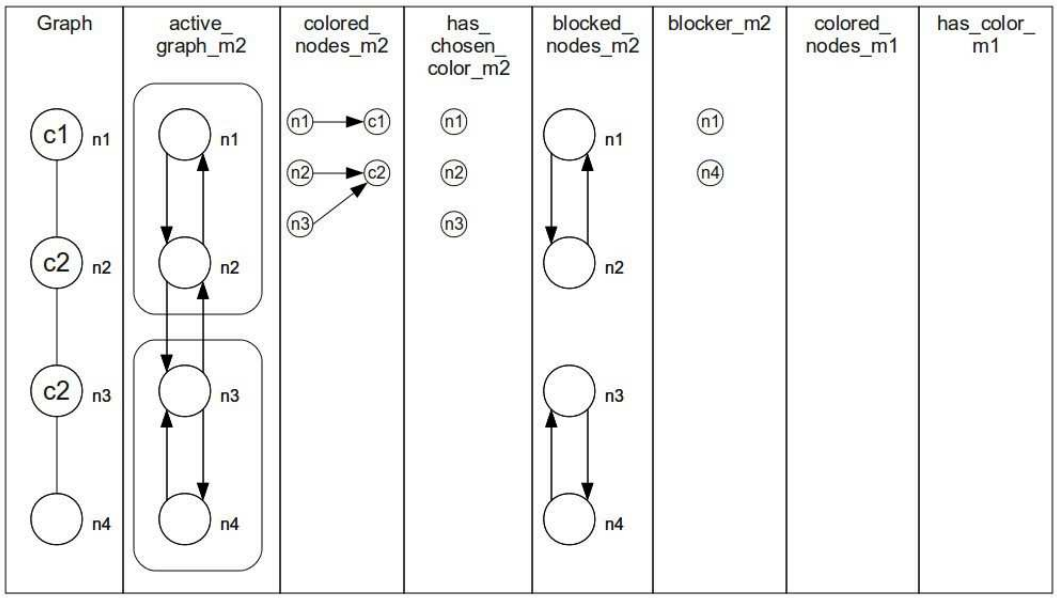
Behaviour Of The First Asynchronous Algorithm

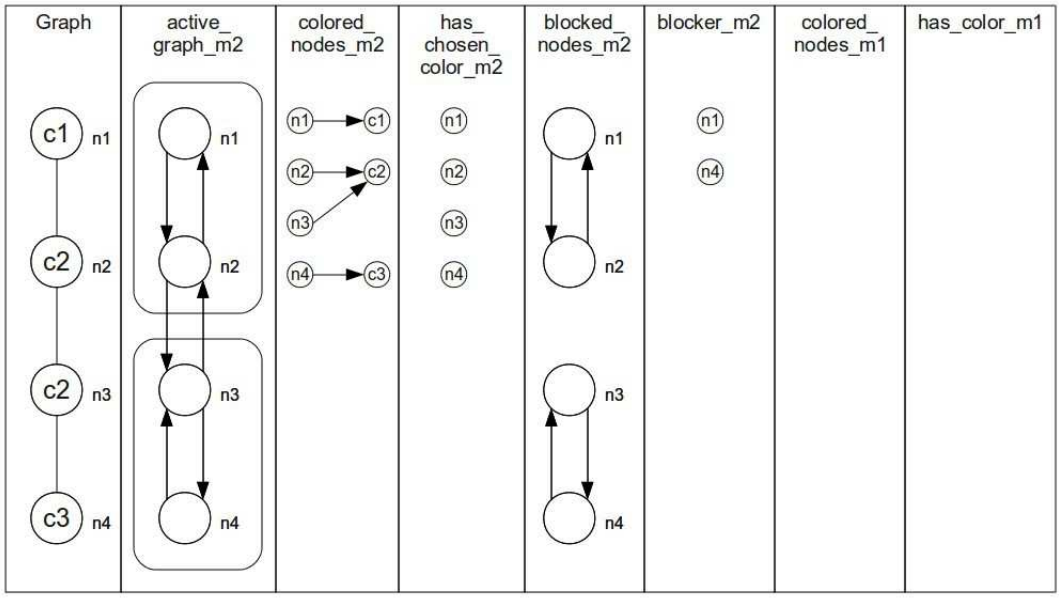
Graph	active_ graph_m2	colored_ nodes_m2	has_ chosen_ color_m2	blocked_ nodes_m2	blocker_m2	colored_ nodes_m1	has_color_ m1
							

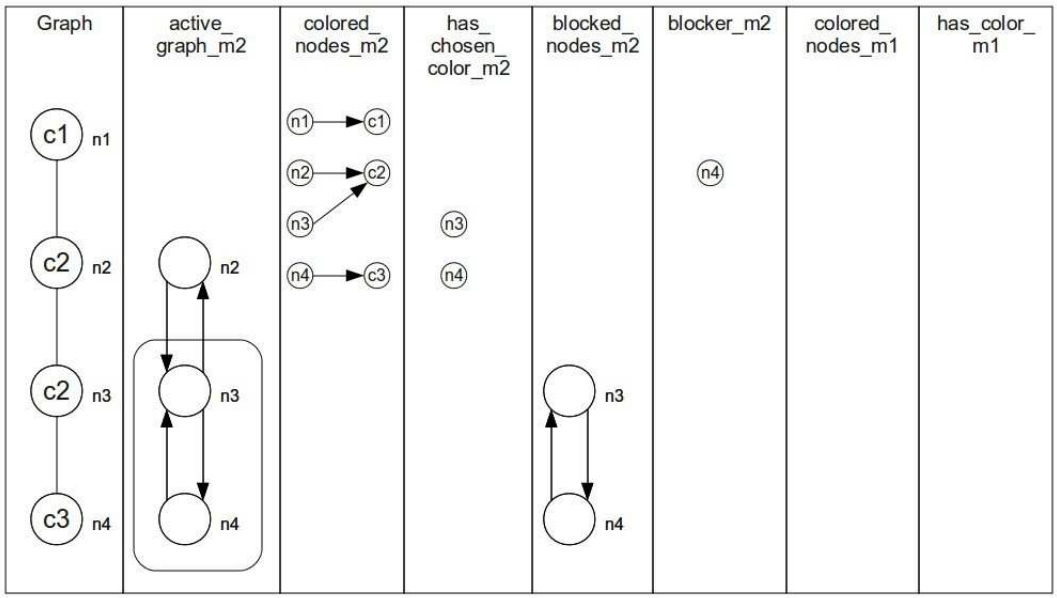
Graph	active_ graph_m2	colored_ nodes_m2	has_ chosen_ color_m2	blocked_ nodes_m2	blocker_m2	colored_ nodes_m1	has_color_ m1
					n1 n4		

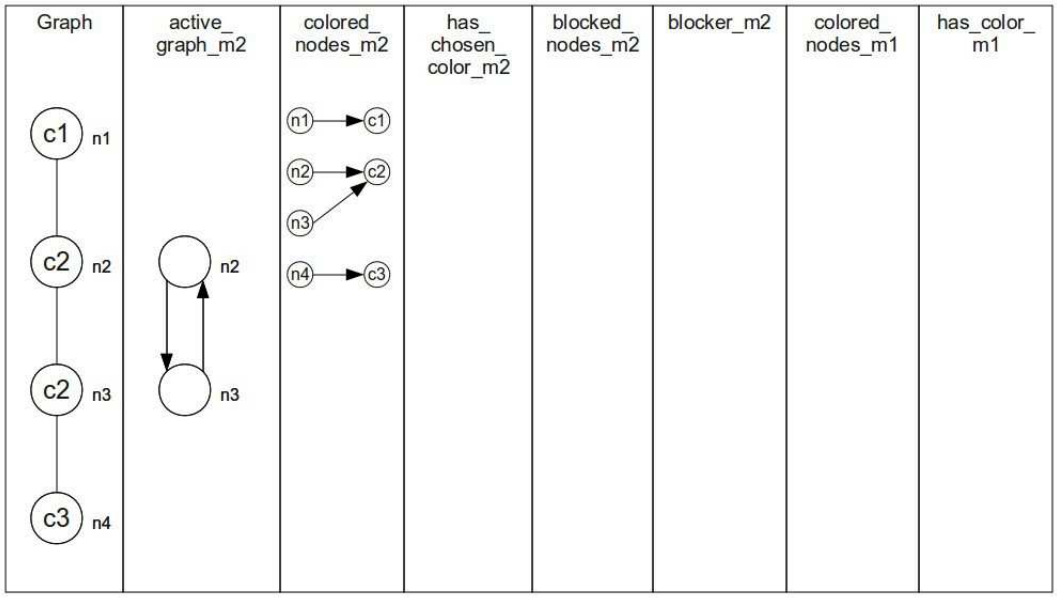

Graph
active_ graph_m2
colored_ nodes_m2
has_ chosen_ color_m2
blocked_ nodes_m2
blocker_m2
colored_ nodes_m1
has_color_ m1
c1
n1
n2
n3
n4
n1
c1

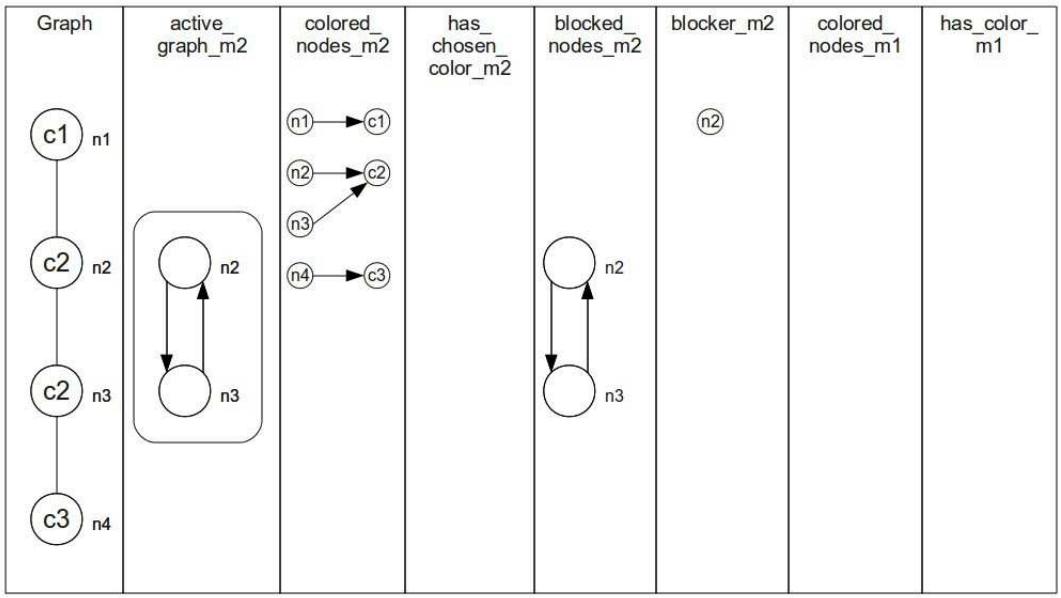

Graph
active_ graph_m2
colored_ nodes_m2
has_ chosen_ colors_m2
blocked_ nodes_m2
blocker_m2
colored_ nodes_m1
has_color_ m1
c1
c2
n1
n2
n3
n4
n1
c1
n2
c2

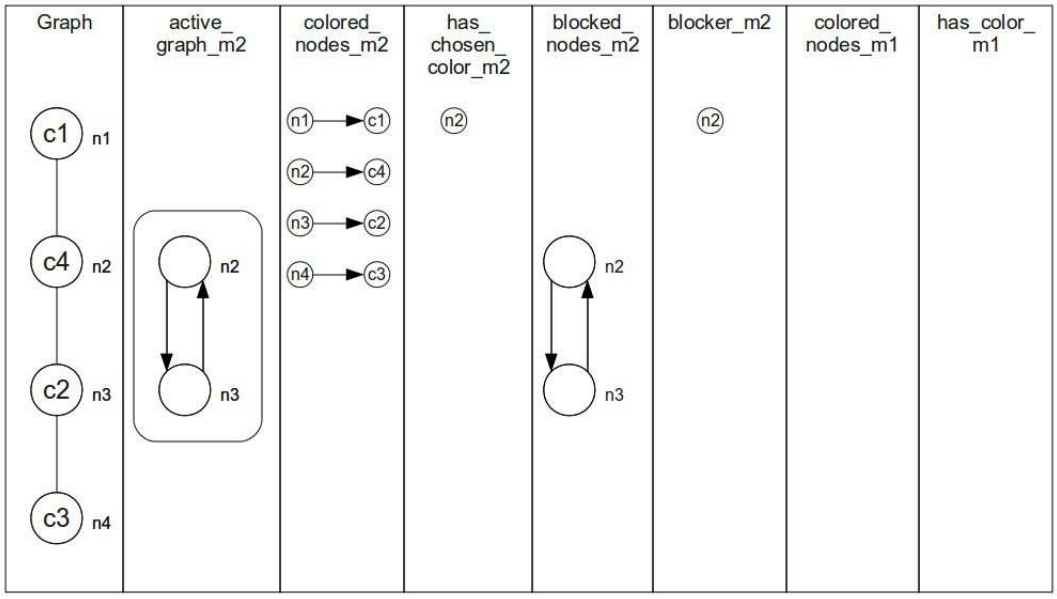

Graph
active_ graph_m2
colored_ nodes_m2
has_ chosen_ color_m2
blocked_ nodes_m2
blocker_m2
colored_ nodes_m1
has_color_ m1
c1
n1
c2
n2
c2
n3
n4
n1
n2
n3
n4
n1
c1
n2
c2
n3
n1
n2
n3
n1
n4
n1
n2
n3
n4

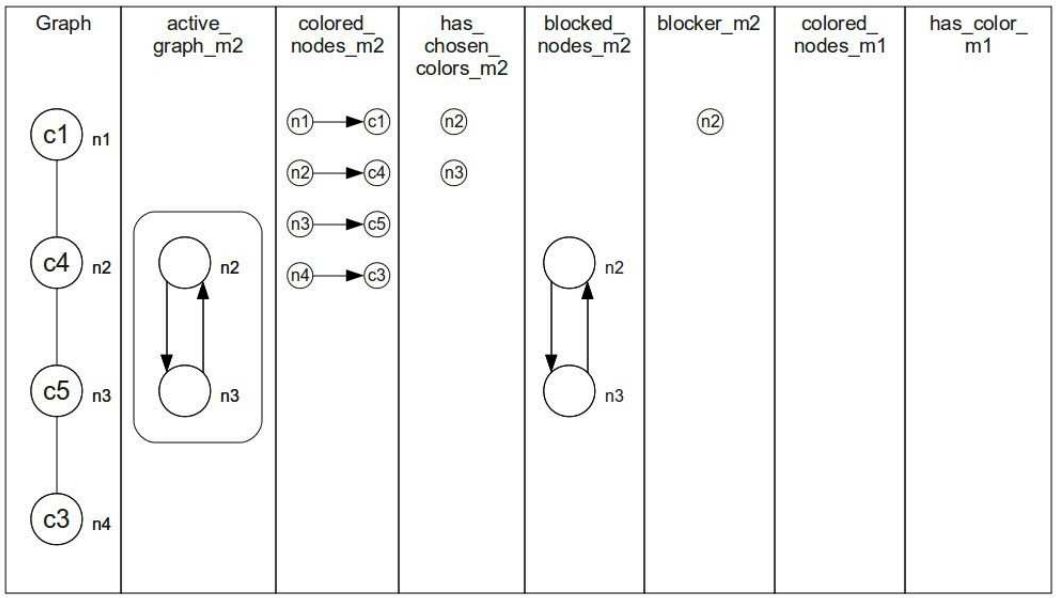

Graph
active_ graph_m2
colored_ nodes_m2
has_ chosen_ color_m2
blocked_ nodes_m2
blocker_m2
colored_ nodes_m1
has_color_m1
c1
n1
c2
n2
c2
n3
c3
n4
n1
n2
n3
n4
n1
c1
n2
c2
n3
c3
n4
n1
n2
n3
n4
n1
n4
n1
n2
n3
n4

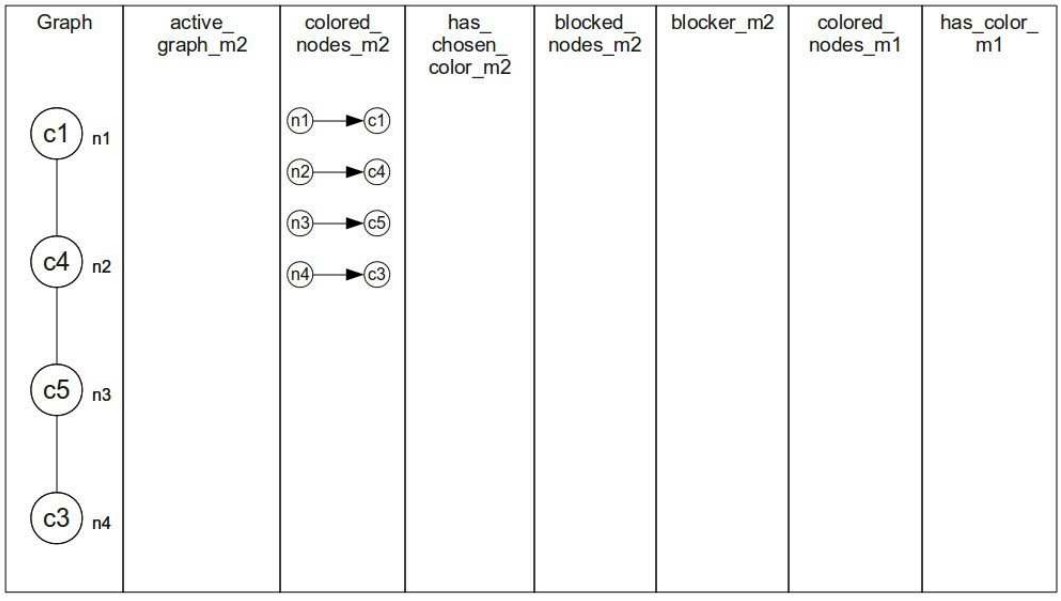

Graph
active_ graph_m2
colored_ nodes_m2
has_ chosen_ color_m2
blocked_ nodes_m2
blocker_m2
colored_ nodes_m1
has_color_ m1
c1
n1
c2
n2
c2
n3
c3
n4
n2
n3
n4
n1
c1
n2
c2
n3
n4
c3
n3
n4
n3
n4
n4

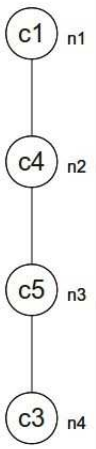
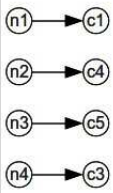

Graph
active_ graph_m2
colored_ nodes_m2
has_ chosen_ color_m2
blocked_ nodes_m2
blocker_m2
colored_ nodes_m1
has_color_ m1
c1
n1
c2
n2
c2
n3
c3
n4
n2
n3
n1
c1
n2
c2
n3
n4
c3

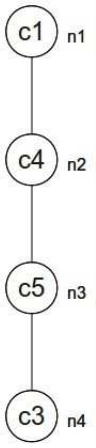
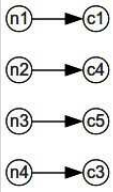
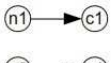

Graph
active_ graph_m2
colored_ nodes_m2
has_ chosen_ color_m2
blocked_ nodes_m2
blocker_m2
colored_ nodes_m1
has_color_ m1
c1
n1
c2
n2
c2
n3
c3
n4
n2
n3
n1
c1
n2
c2
n3
n4
c3
n2
n3
n2

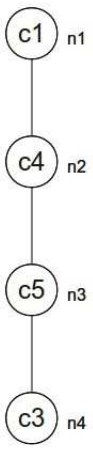
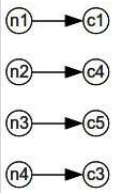
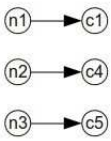

Graph
active_ graph_m2
colored_ nodes_m2
has_ chosen_ color_m2
blocked_ nodes_m2
blocker_m2
colored_ nodes_m1
has_color_ m1
c1
n1
c4
n2
c2
n3
c3
n4
n2
n3
n1
c1
n2
c4
n3
c2
n4
c3
n2
n2
n3

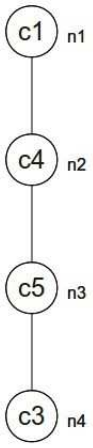
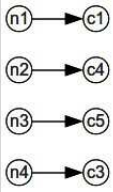
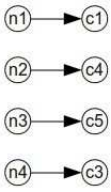

Graph
active_ graph_m2
colored_ nodes_m2
has_ chosen_ colors_m2
blocked_ nodes_m2
blocker_m2
colored_ nodes_m1
has_color_ m1
c1
n1
c4
n2
c5
n3
c3
n4
n1
c1
n2
c4
n3
c5
n4
c3
n2
n3
n2
n3
n2


Graph
active_ graph_m2
colored_ nodes_m2
has_ chosen_ color_m2
blocked_ nodes_m2
blocker_m2
colored_ nodes_m1
has_color_ m1
c1
n1
c4
n2
c5
n3
c3
n4
n1
c1
n2
c4
n3
c5
n4
c3

Graph	active_ graph_m2	colored_ nodes_m2	has_ chosen_ color_m2	blocked_ nodes_m2	blocker_m2	colored_ nodes_m1	has_color_ m1
							

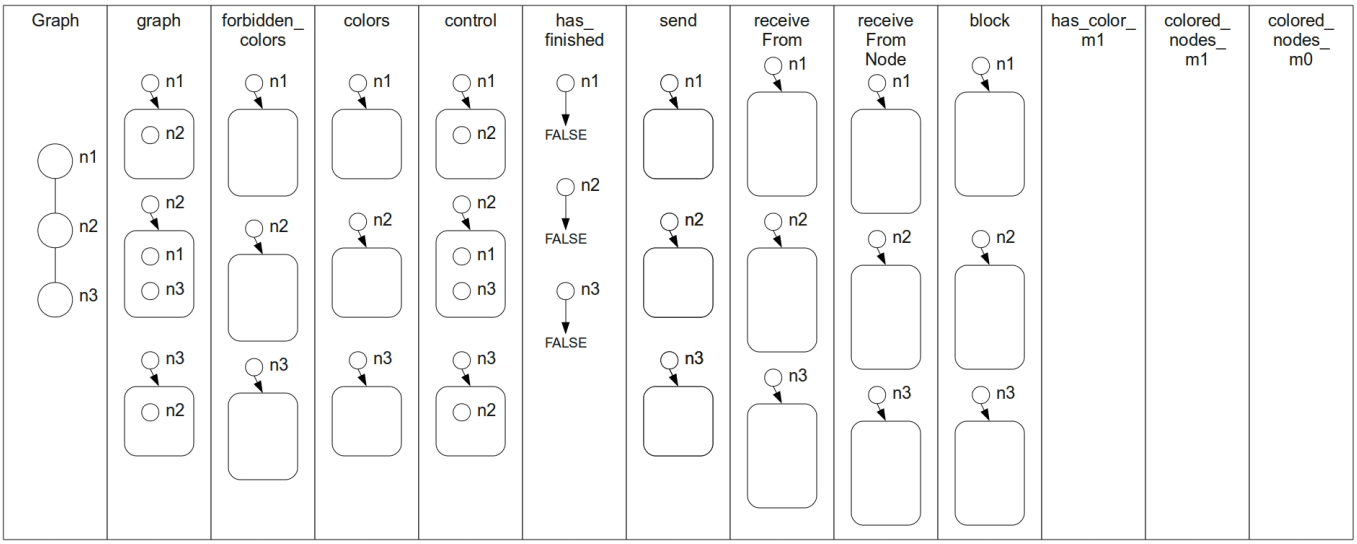
Graph	active_ graph_m2	colored_ nodes_m2	has_ chosen_ color_m2	blocked_ nodes_m2	blocker_m2	colored_ nodes_m1	has_color_ m1
							

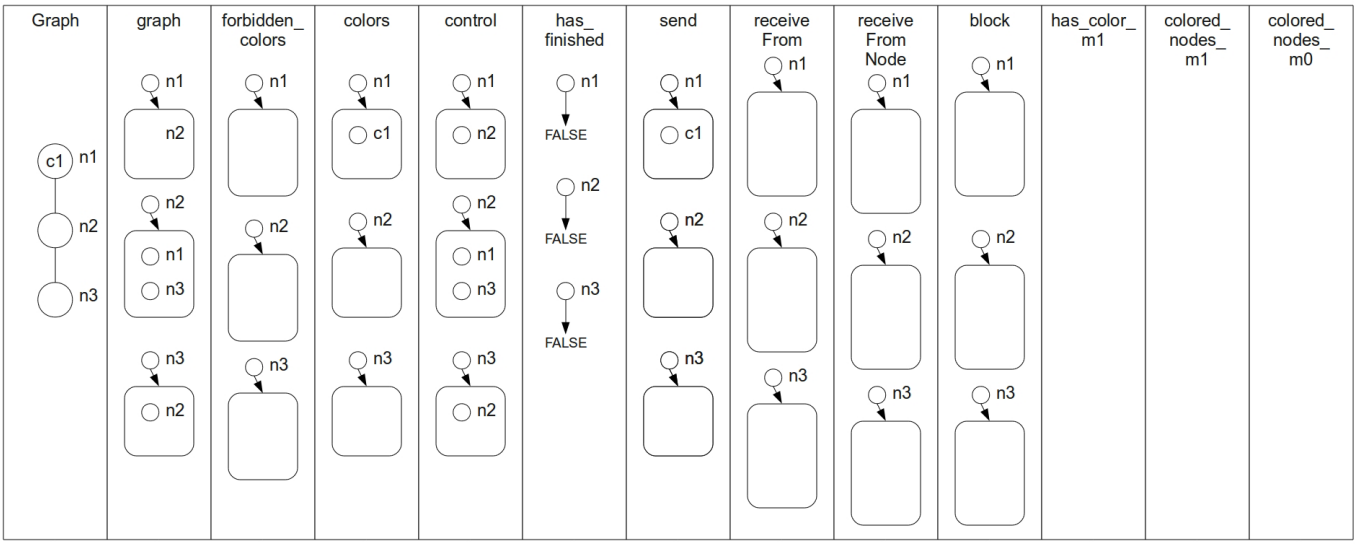
Graph	active_ graph_m2	colored_ nodes_m2	has_ chosen_ color_m2	blocked_ nodes_m2	blocker_m2	colored_ nodes_m1	has_color_ m1
							

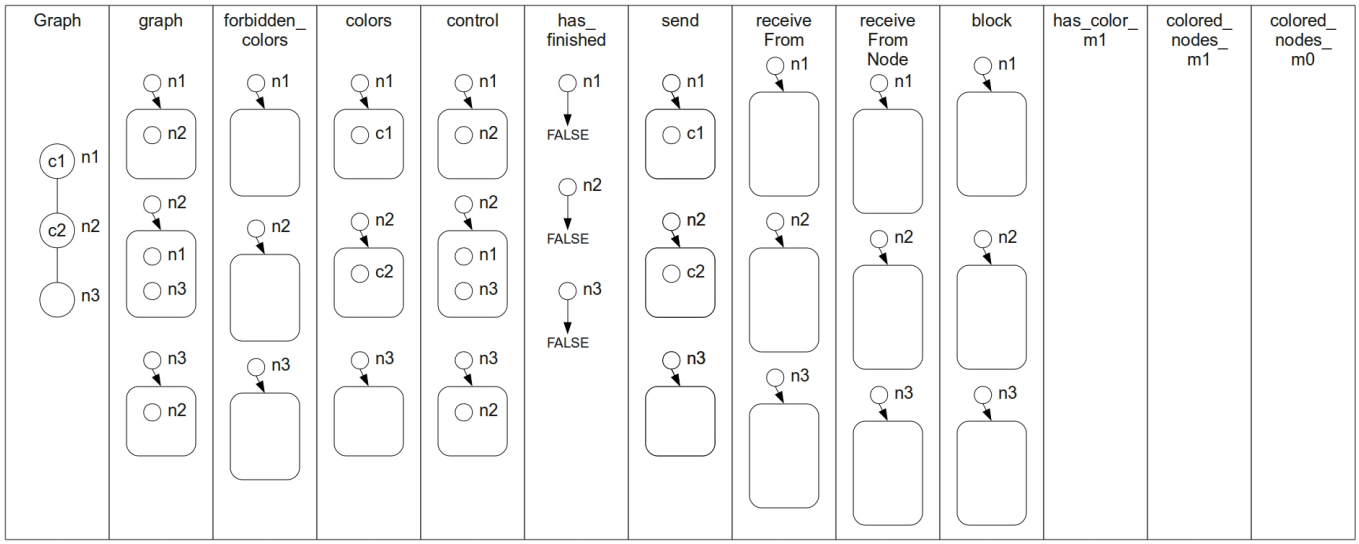
Graph	active_ graph_m2	colored_ nodes_m2	has_ chosen_ color_m2	blocked_ nodes_m2	blocker_m2	colored_ nodes_m1	has_color_ m1
							

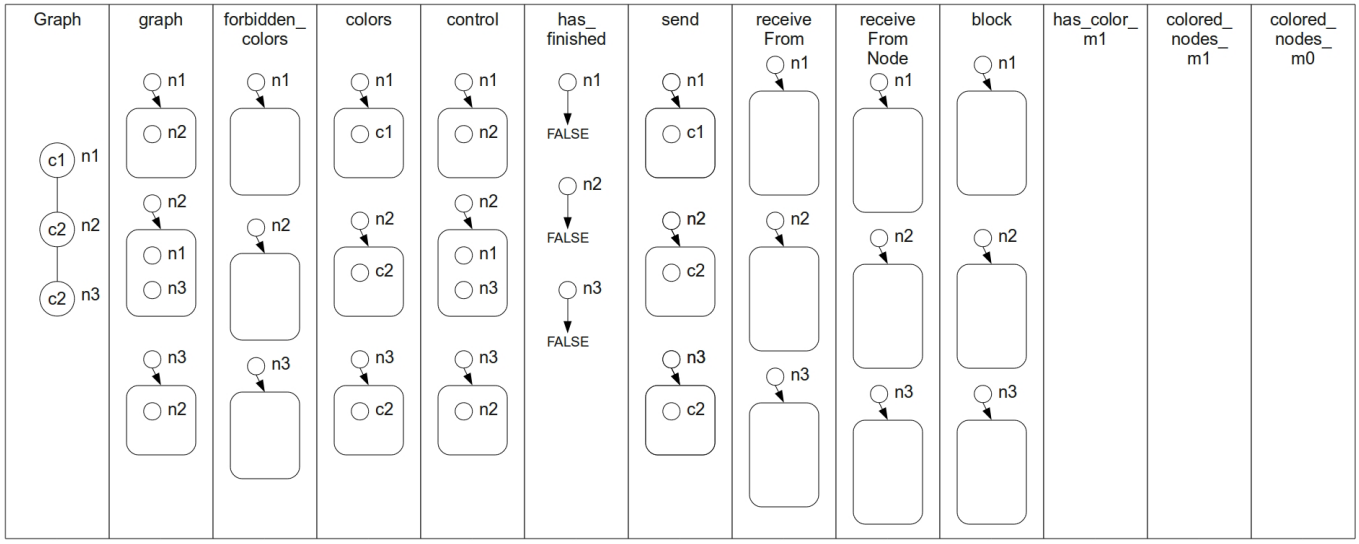
Appendix C

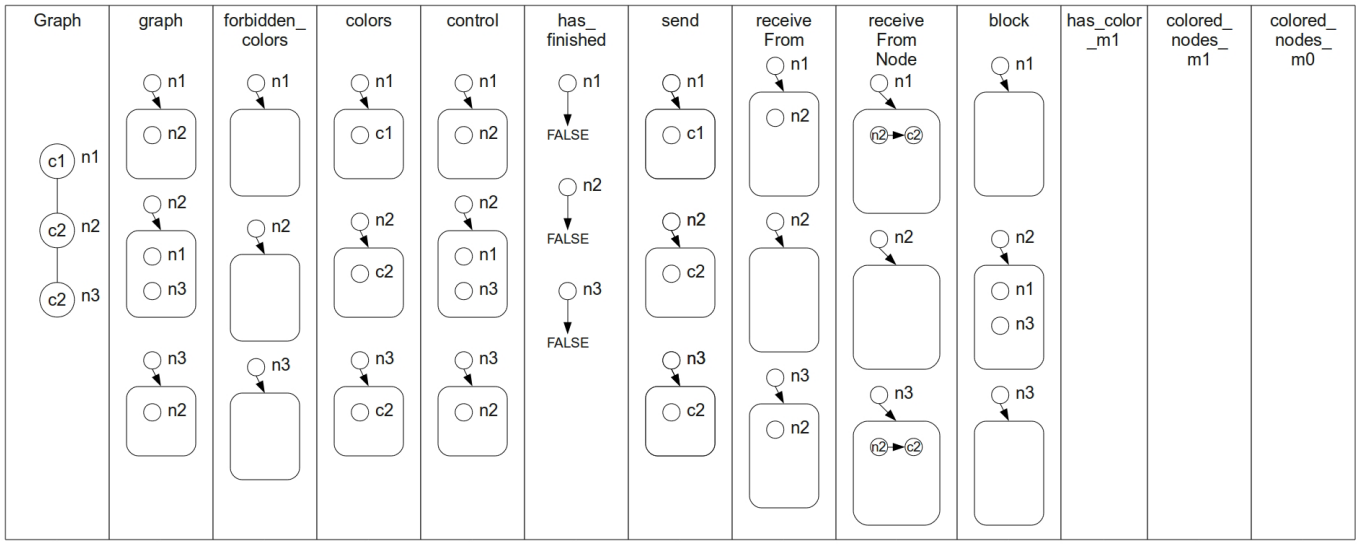
Behaviour Of The Local Asynchronous Algorithm

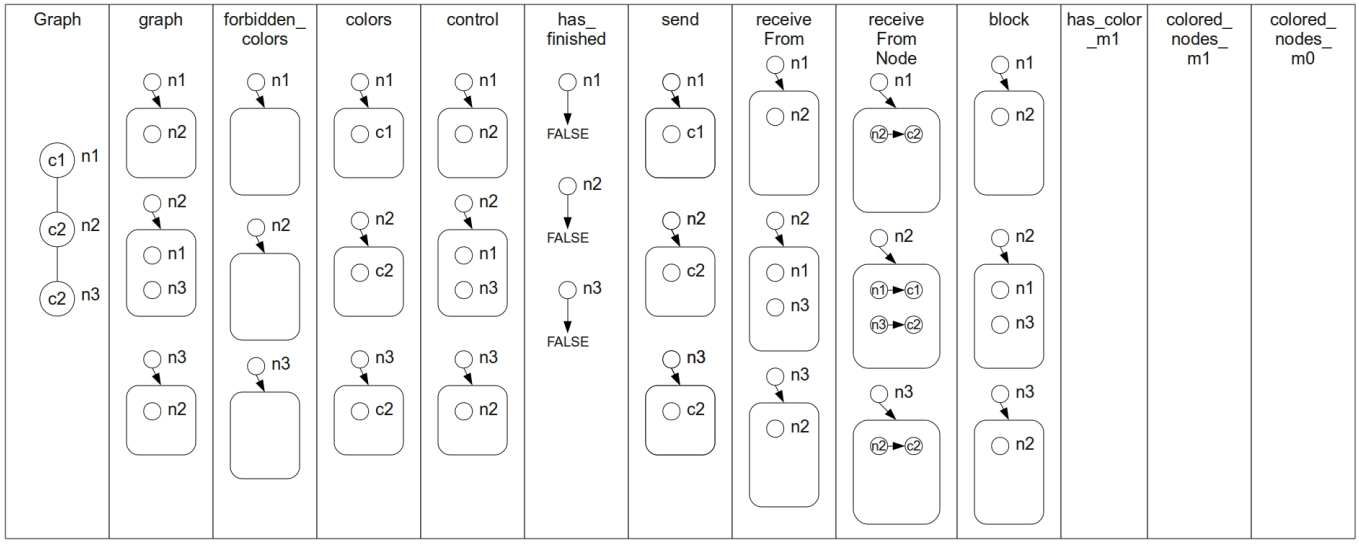

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color_ m1
colored_ nodes_ m1
colored_ nodes_ m0
n1
n2
n3
FALSE
FALSE
FALSE

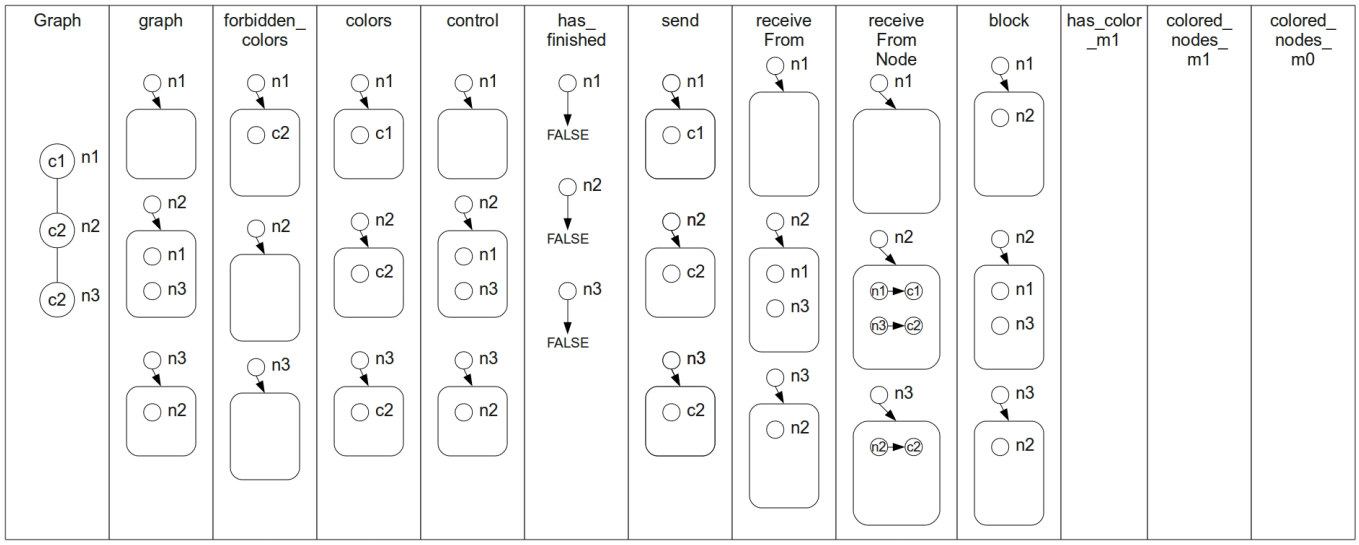

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color_ m1
colored_ nodes_ m1
colored_ nodes_ m0
c1
n1
n2
n3
FALSE
FALSE
FALSE

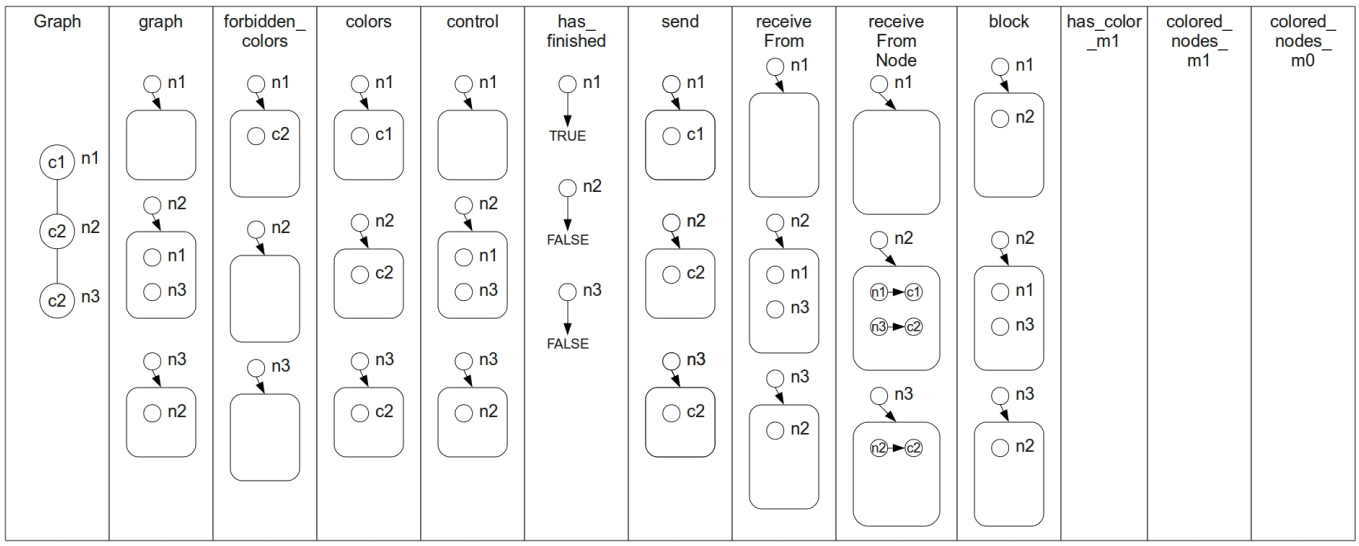

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color_ m1
colored_ nodes_ m1
colored_ nodes_ m0
c1
n1
c2
n2
n3
FALSE

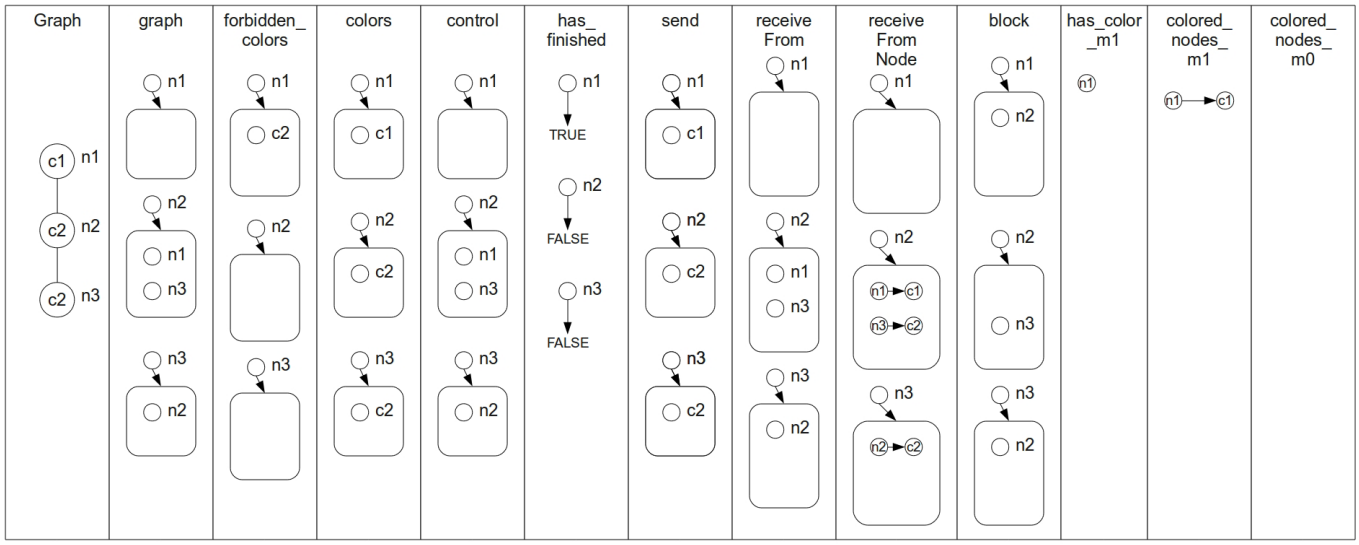

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color_ m1
colored_ nodes_ m1
colored_ nodes_ m0
c1
n1
c2
n2
c2
n3
FALSE

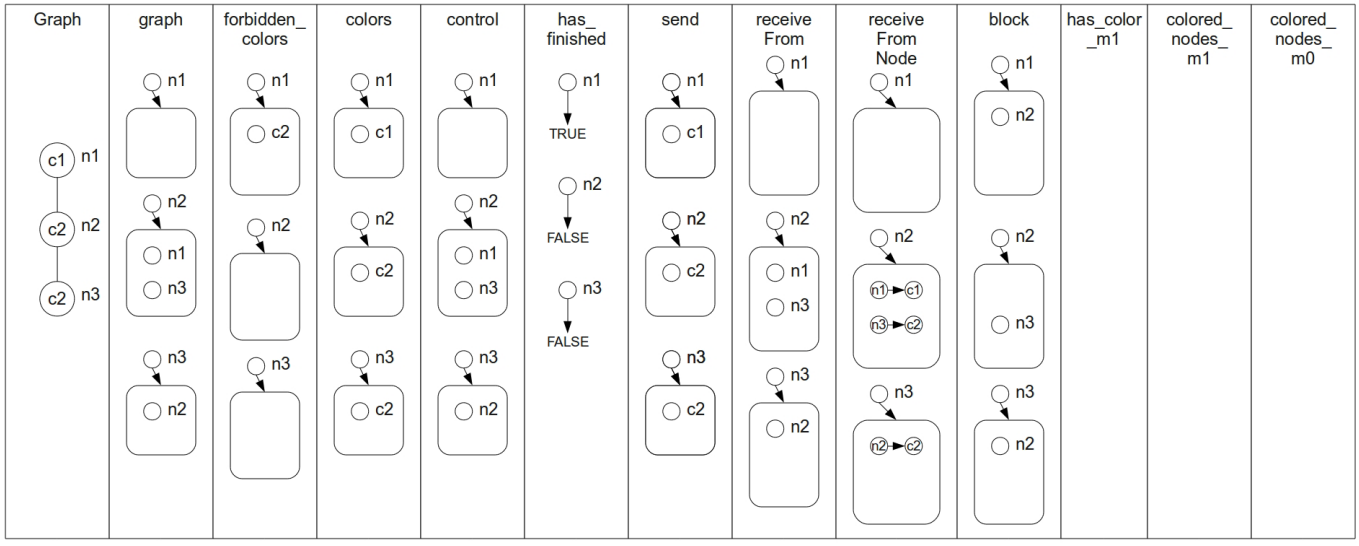

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
c1
n1
c2
n2
c2
n3
FALSE
FALSE
FALSE

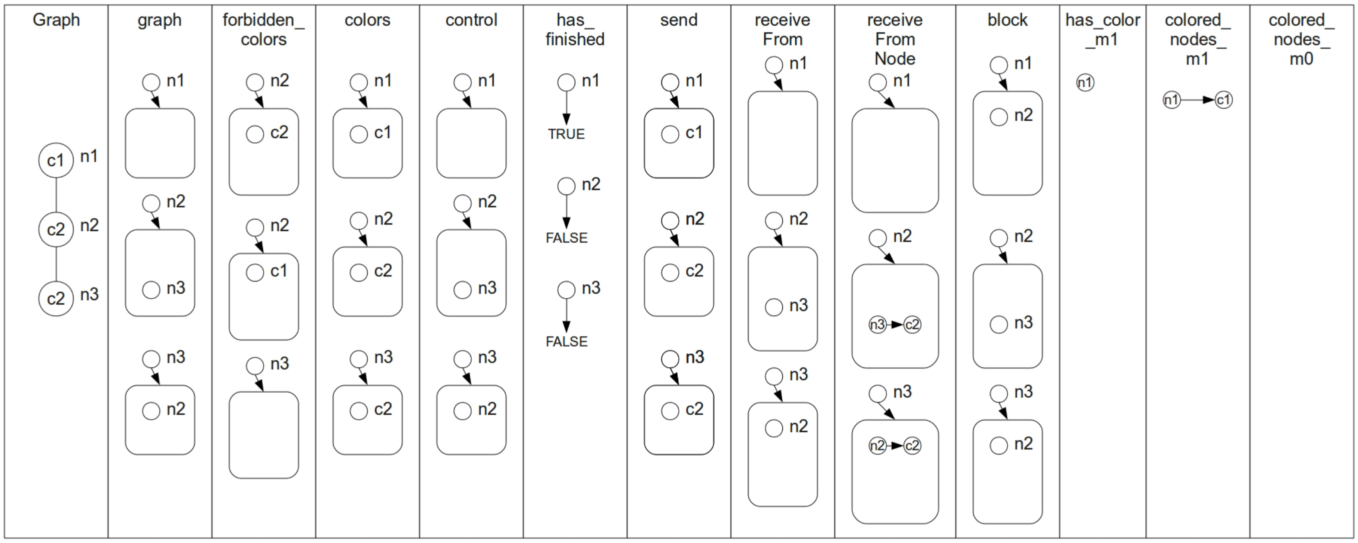

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
c1
n1
c2
n2
c2
n3
FALSE
FALSE
FALSE

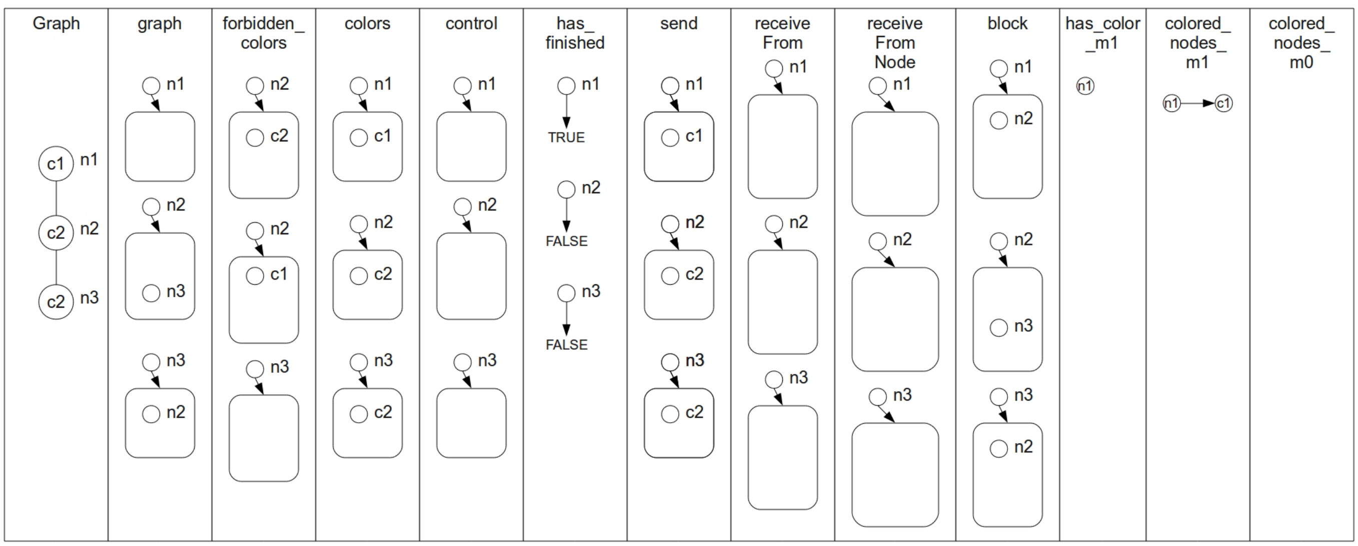

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
c1
n1
c2
n2
c2
n3
n1
n2
n3
c2
c1
c2
c2
c2
c1
c2
c2
c2
FALSE
FALSE
FALSE
n1
n3
n2
n1
c1
n3
c2
n2
n1
n3
n2

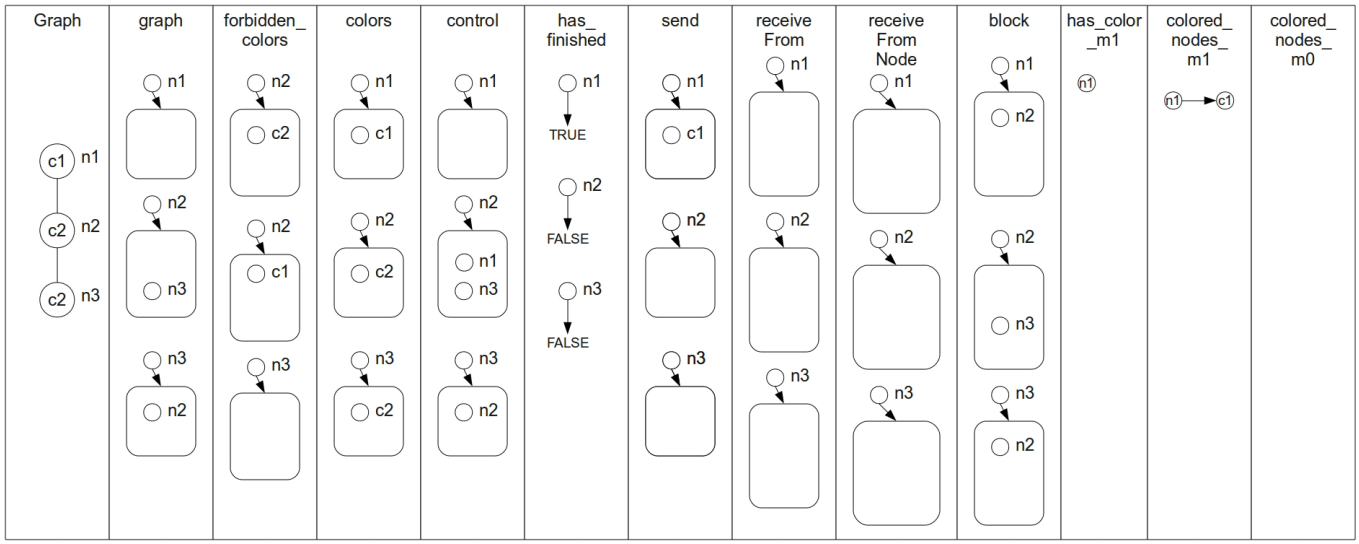

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
c1
n1
c2
n2
c2
n3
n1
n2
n3
c2
c1
c2
c2
c2
c1
c2
c2
c2
TRUE
FALSE
FALSE
n1
n3
n2
n1
c1
n3
c2
n2
n1
n3
n2

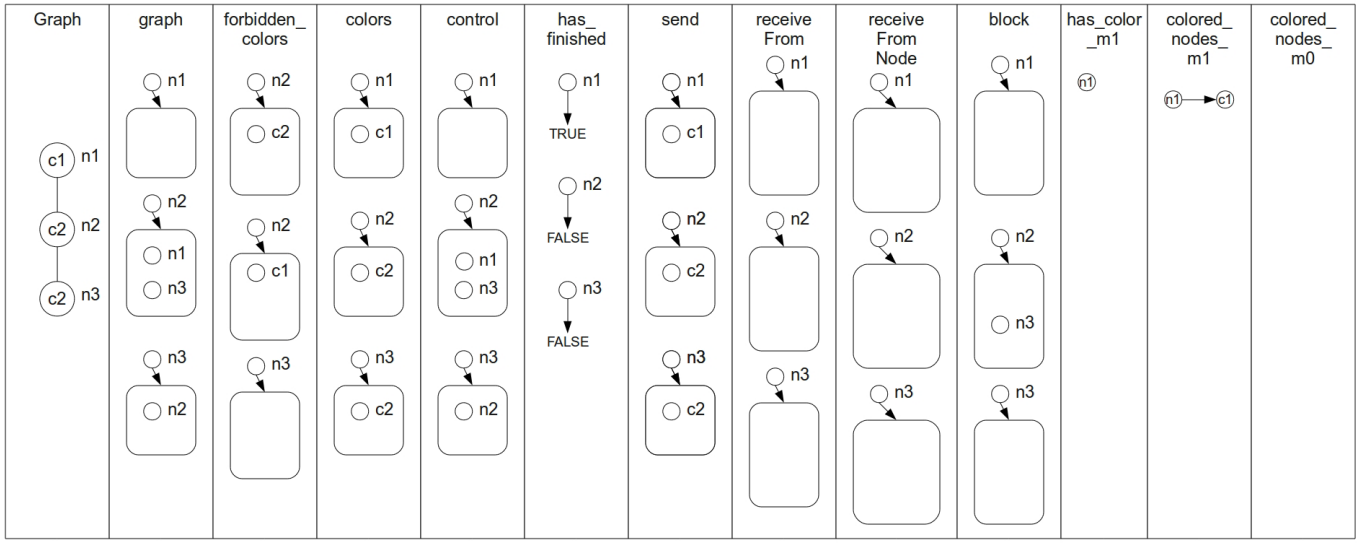

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
c1 n1
c2 n2
c2 n3
n1
n2
n3
c1
c2
TRUE
FALSE
FALSE

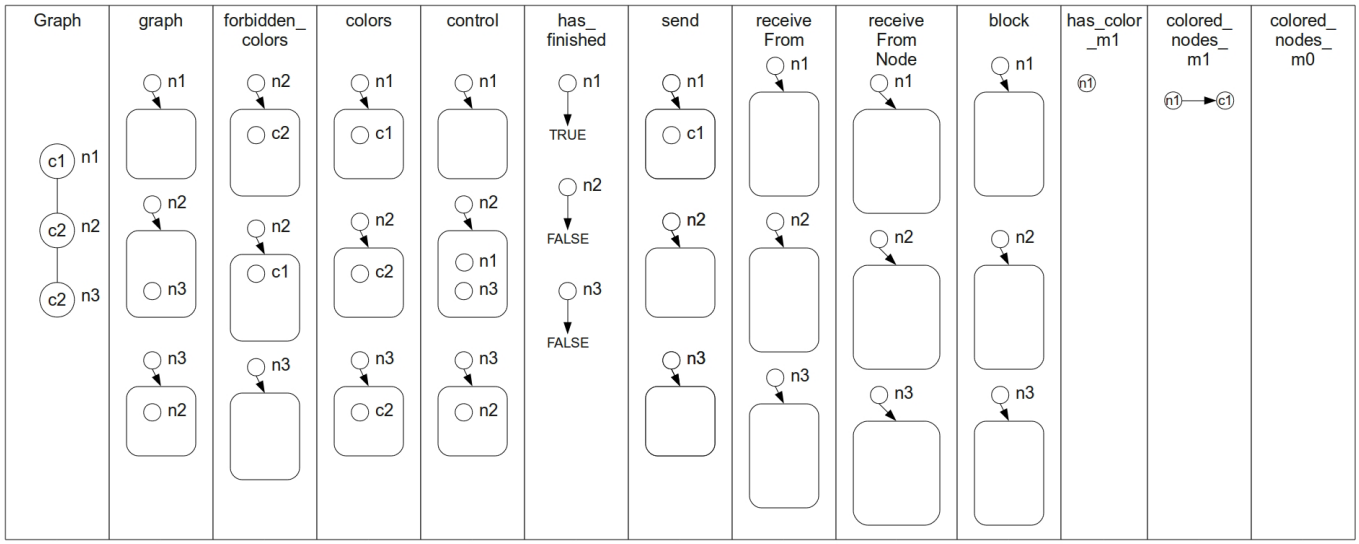

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
c1 n1
c2 n2
c2 n3
n1
n2
n3
c1
c2
TRUE
FALSE
FALSE

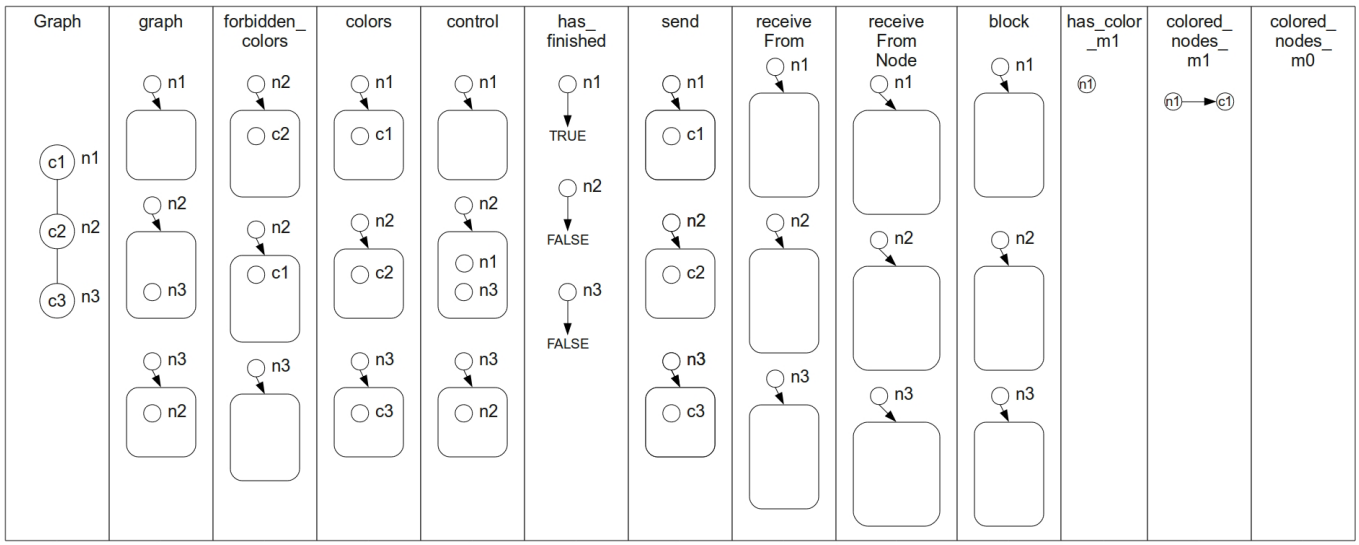

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
c1 n1
c2 n2
c2 n3
n1
n2
n3
n2
c2
c1
c2
TRUE
FALSE
FALSE
n3
c2
n2
c2
n1
c1

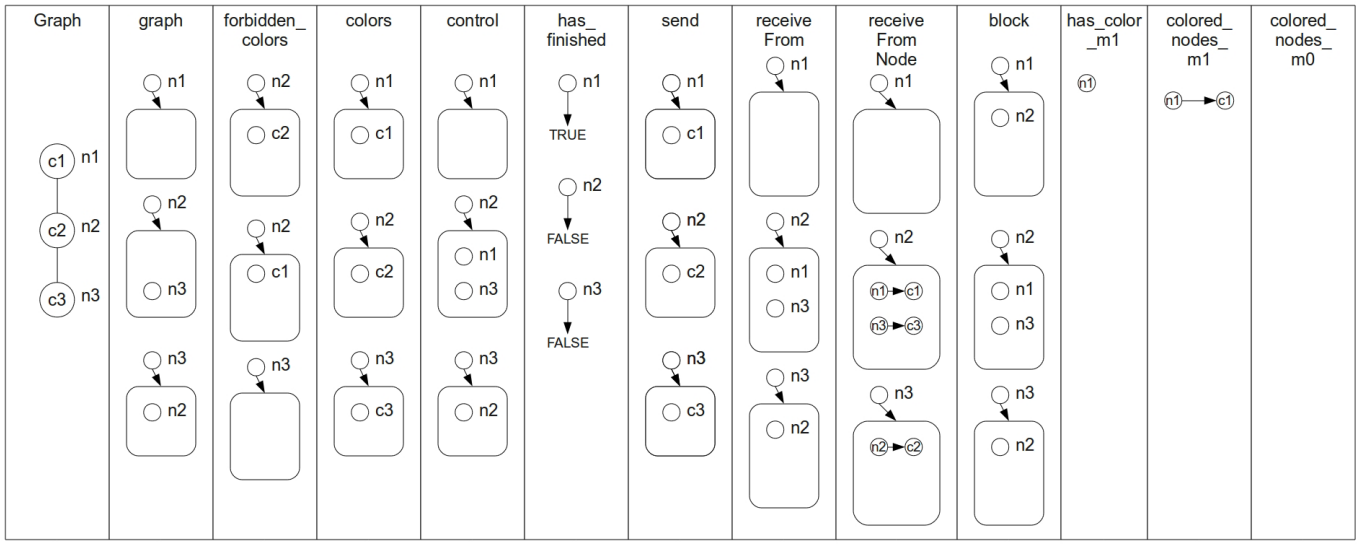

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
c1 n1
c2 n2
c2 n3
n1
n2
n3
c2
c1
TRUE
FALSE
FALSE
n1
c1

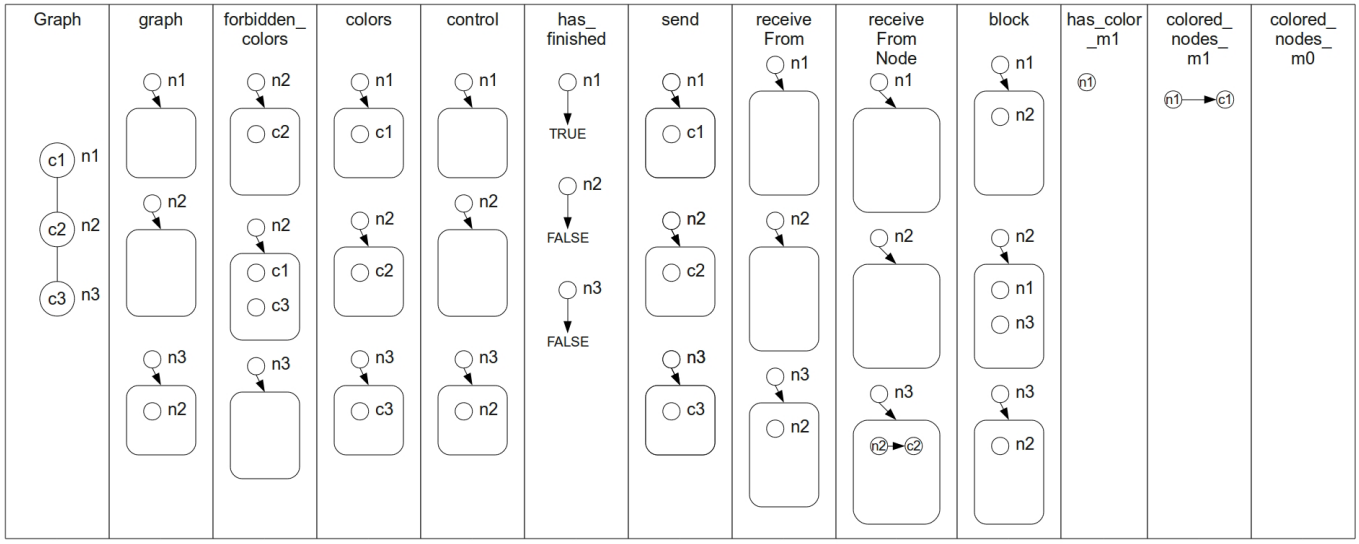

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
c1 n1
c2 n2
c2 n3
n1
n2
n3
c2
c1
TRUE
FALSE
FALSE
n1
n2
n3

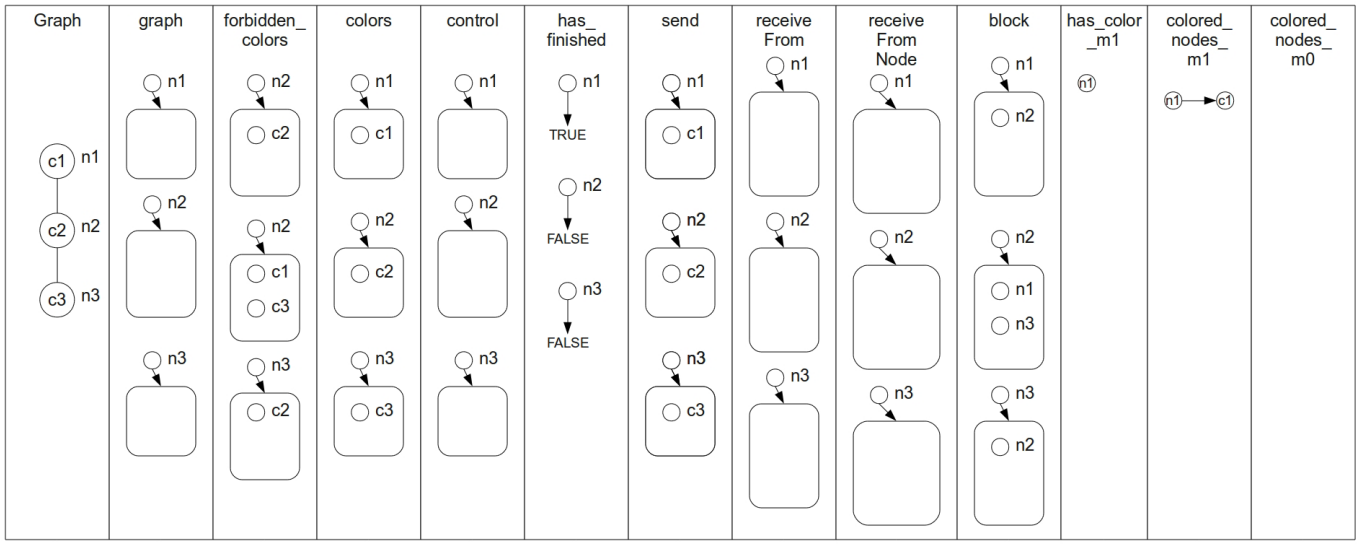

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
c1 n1
c2 n2
c2 n3
n1
n2
n3
c2
c1
TRUE
FALSE
FALSE
n1
n2
n3

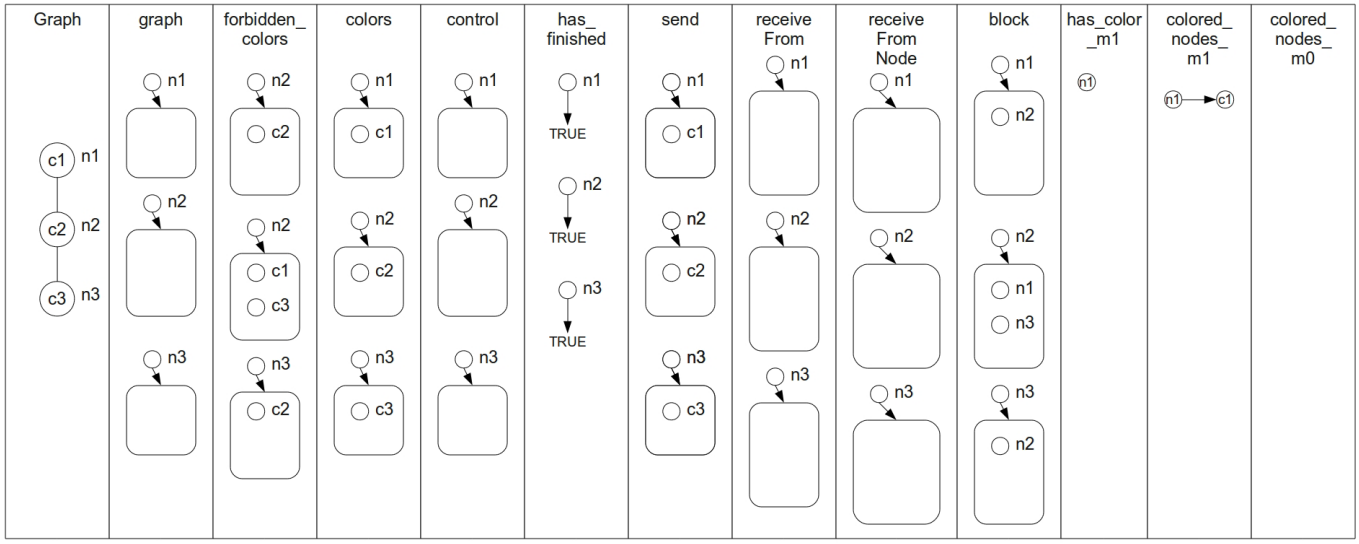

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
c1 n1
c2 n2
c2 n3
n1
n2
n3
c2
c1
c1
c2
c2
n1
n3
n2
TRUE
FALSE
FALSE
c1
c2
c3
n1
n3
n2
n1
n2
n3
n1
n2
n3
n1
n2
n3
n1
c1

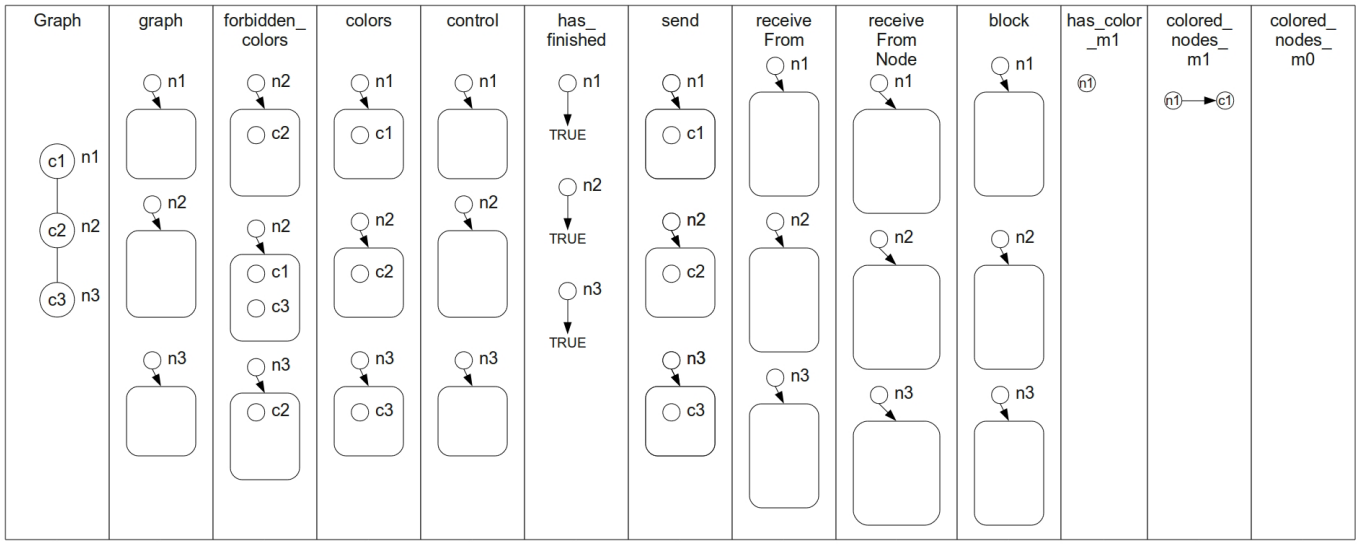

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
c1 n1
c2 n2
c3 n3
n1
n2
n3
c2
c1
c1
c2
c3
n1
n3
n2
TRUE
FALSE
FALSE
c1
c2
c3
n1
n2
n3
n1
n2
n3
n1
n2
n3
n1
c1

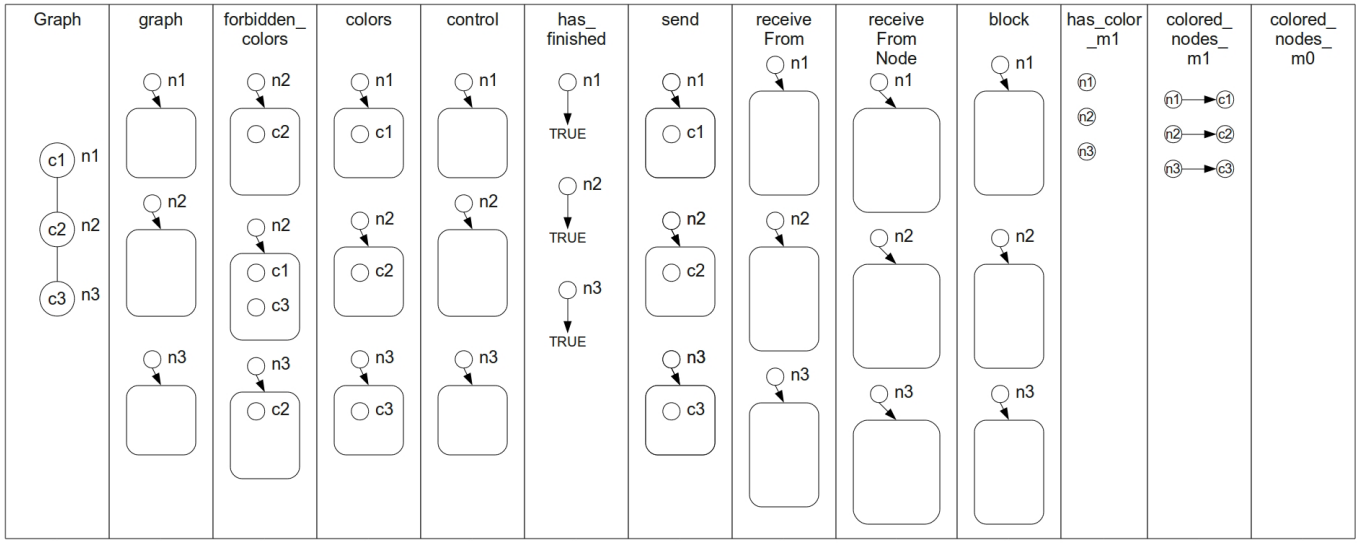

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
c1 n1
c2 n2
c3 n3
n1
n2
n3
TRUE
FALSE
FALSE

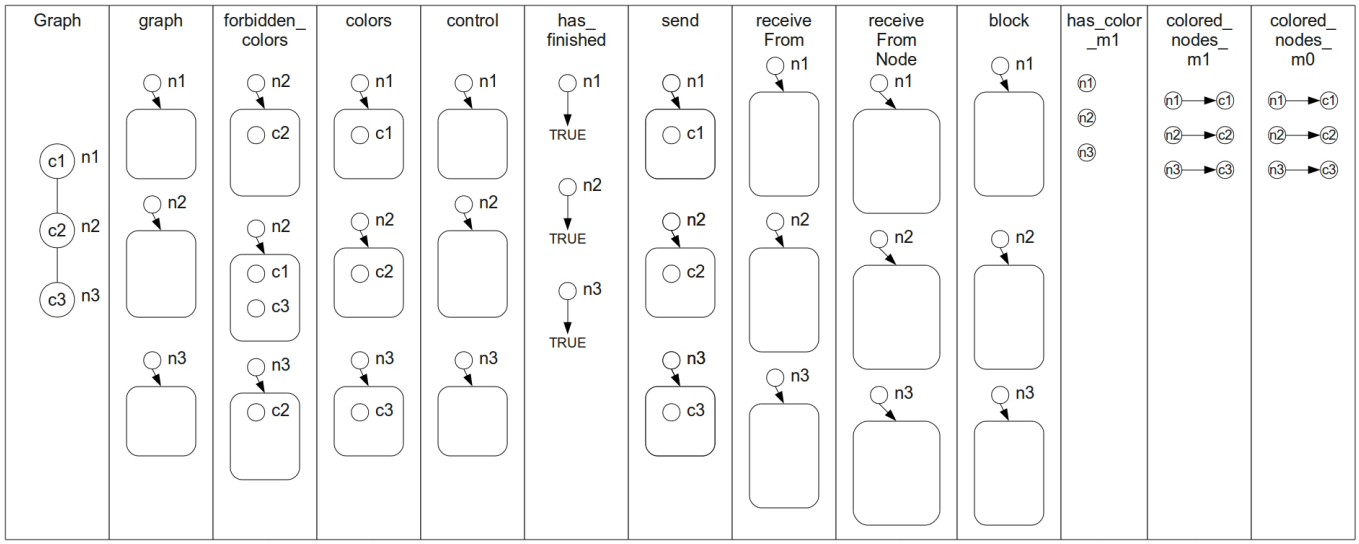

Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
c1 n1
c2 n2
c3 n3
n1
n2
n3
TRUE
FALSE
FALSE


Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
c1 n1
c2 n2
c3 n3
n1
n2
n3
c1
c2
c3
TRUE
FALSE


Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
c1 n1
c2 n2
c3 n3
n1
n2
n3
c1
c2
c3
TRUE


Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
n1
n2
n3
c1
c2
c3
TRUE


Graph
graph
forbidden_ colors
colors
control
has_ finished
send
receive From
receive From Node
block
has_color _m1
colored_ nodes_ m1
colored_ nodes_ m0
n1
n2
n3
c1
c2
c3
TRUE


Graph
c1
n1
c2
n2
c3
n3
graph
n1
n2
n3
forbidden_ colors
n2
c2
n2
c1
c3
n3
c2
colors
n1
c1
n2
c2
n3
c3
control
n1
n2
n3
has_ finished
n1
TRUE
n2
TRUE
n3
TRUE
send
n1
c1
n2
c2
n3
c3
receive From
n1
n2
n3
receive From Node
n1
n2
n3
block
n1
n2
n3
has_color _m1
n1
n2
n3
colored_ nodes_ m1
n1
c1
n2
c2
n3
c3
colored_ nodes_ m0
n1
c1
n2
c2
n3
c3