Reasoning about Universal Cubes in MCMT

Sylvain Conchon, Mattias Roux

Reasoning about Universal Cubes in MCMT

Sylvain Conchon and Mattias Roux

LRI, Université Paris Sud, CNRS, Orsay F-91405

Abstract. The Model Checking Modulo Theories (MCMT) framework is a powerful model checking technique for verifying safety properties of parameterized transition systems. In MCMT, logical formulas are used to represent both transitions and sets of states and safety properties are verified by an SMT-based backward reachability analysis. To be fully automated, the class of formulas handled in MCMT is restricted to *cubes*, i.e. existentially quantified conjunction of literals. While being very expressive, cubes cannot define properties with a global termination condition, usually described by a universally quantified formula.
In this paper we describe BRWP, an extension of the backward reachability of MCMT for reasoning about validity properties expressed as *universal cubes*, that is formulas of the form $\exists \boldsymbol{i} \forall \boldsymbol{j}.\mathcal{C}(\boldsymbol{i},\boldsymbol{j})$, where $\mathcal{C}(\boldsymbol{i},\boldsymbol{j})$ is a conjunction of literals. Our approach consists in a tight cooperation between the backward reachability loop and a deductive verification engine based on weakest-precondition calculus (WP). To provide evidence for the applicability of our new algorithm, we show how to make Cubicle, a model checker based on MCMT, cooperates with the Why3 platform for deductive program verification.

1 Introduction

In this paper, we consider the problem of verifying safety properties of parameterized systems. The systems we are interested in are called *array-based* transition systems. This is a syntactically restricted class of parameterized systems, introduced by Ghilardhi and Ranise [**?**] where states are represented as arrays indexed by an arbitrary number of processes. Distributed systems with consensus or commitment protocols are typical examples modeling with array-based systems.

The verification of array-based systems as proposed in [**?**] led to a powerful model checking technique called *Model Checking Modulo Theories* (MCMT). This is a symbolic SMT-based model checking technique where logical formulas (expressed in a fragment of first-order logic) are used to represent both transitions and sets of states, and safety properties are verified by backward reachability analysis. A safety property to be verified in MCMT is expressed in its negated form as a formula that represents unsafe states. Each unsafe formula must be a *cube*, i.e., have the form $\exists \boldsymbol{i}.\mathcal{C}(\boldsymbol{i})$, where $\mathcal{C}(\boldsymbol{i})$ is a conjunction of literals.

While the expressiveness of cubes is sufficient to encode a large class of *safety invariants*, it is usually too weak for describing safety properties with a *global*

termination condition. For instance, in a consensus algorithm, one would like to check that, at the end of the consensus, there is no process deciding a value distinct from the value chosen by the others. Unfortunately, the "at the end of the consensus" part of the sentence must take the form of a *universally* quantified formula defining the condition for the processes to terminate. To cope with such properties, MCMT must be extended to reason about *universal cubes*, that is formulas of the form $\exists \boldsymbol{i} \forall \boldsymbol{j}.\mathcal{C}(\boldsymbol{i}, \boldsymbol{j})$, where $\mathcal{C}(\boldsymbol{i}, \boldsymbol{j})$ is a conjunction of literals.

To handle such formulas, one can try to encode universal cubes as transitions with *universal guards*, i.e., guards containing universally quantified global conditions that check the state variables of all processes. However, since universal quantifiers in guards prevents the backward reachability algorithm to be fully automated, solutions based on over approximations techniques have been proposed [**?**,**?**,**?**]. One of the best solution is proposed in [**?**] as a syntactic transformation which can be seen as the implementation of a crash-failure model where an unbounded number of processes can die at any time. Unfortunately, while very efficient, this over approximation technique results in false positives for non fault-tolerant systems which are very common in distributed systems.

Another way to handle universal cubes would consist to give up model checking techniques and instead to use a more expressive and powerful Hoare-style reasoning. For instance, translating Cubicle systems to the input language of the TLA+ system [**?**] is straightforward and would allow the user to use a proof system like TLAPS [**?**]. Similarly, one can translate Cubicle's input language to DVF [**?**], a deductive verification framework dedicated to transition systems which uses SMT solvers to prove the generated verification conditions. However, while those frameworks offer automatic backends to discharge proof obligations, an important and very painful part of the proof effort consists in finding *manually* the auxiliary invariants of the system which are mandatory for the safety property to be proved.

In this paper, we propose to bridge the gap between model checking and deductive verification. Our technique consists in a tight cooperation between the backward reachability loop of MCMT and a deductive verification engine based on weakest-precondition calculus. To provide evidence for the applicability of our technique, we show how to make Cubicle, a model checker based on MCMT, cooperate with the Why3 platform for deductive program verification [**?**]. Our contributions are as follows:

- A new algorithm, called BRWP, that extends the backward reachability algorithm to handle universal cubes (Section 4).
- A translation schema from Cubicle to Why3 (Section 5).

In Section 2, we illustrate the problem of handling universal cubes in MCMT on a simplified version of a splitter, a basic building block of renaming algorithms in shared memory. We give an overview of our approach in Section 3.

2 The Problem of Universal Quantifiers in MCMT

Throughout this paper, we use a simplified version of a splitter algorithm to illustrate the problems and solutions we are presenting.

Splitters have been first introduced by Lamport [**?**] to implement fast mutual-exclusion algorithms, then used by Moir and Anderson to solve the renaming problem in shared memory [**?**]. A splitter can be depicted graphically by the schema in Figure 1. It is a concurrent object used to distinguish an arbitrary number (n) of callers. Each process that calls the splitter gets a decision value among stop, down and right. The decision values respect the following rules:

- There are only three possible decisions : `Stop`, `Right` and `Down`
- At most one process ends in `Stop`
- At most $n-1$ processes end in `Down`
- At most $n-1$ processes end in `Right`

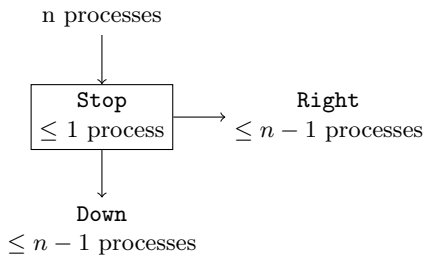

Fig. 1. Splitter

The splitter algorithm for each process p is represented in Figure 2 as an automaton with seven states ($\texttt{PC}_0$ to $\texttt{PC}_3$, `Stop`, `Down` and `Right`) and two boolean variables `X` and `Y`. The initial state is $\texttt{PC}_0$ where `Y` is supposed to be initialised to false and `X` can contain any value. The first transition should be read as follows: a process p in state $\texttt{PC}_0$ can go to $\texttt{PC}_1$ and assign `X` to p. Similarly, if `Y` is false then a process p can go from $\texttt{PC}_1$ to $\texttt{PC}_2$ else it can go to state `Right`. A transition from $\texttt{PC}_2$ to $\texttt{PC}_3$ assigns `Y` to true ($\top$). Finally, the process p can go from $\texttt{PC}_3$ to `Stop` if $\texttt{X} = p$, otherwise it can go to `Down`.

Modeling this (simplified) splitter algorithm is immediate using array-based transition systems. We assume given an enumeration type `state` with seven constructors ($\texttt{PC}_0$, ..., $\texttt{PC}_3$, `Stop`, `Down` and `Right`), two variables `X` and `Y` and an array `PC` such that, for each process p, $\texttt{PC}[p]$ contains a value of type `state`. Initially, each process is in state $\texttt{PC}_0$ and $\texttt{Y} = \bot$, which is described by the following universal formula $\mathcal{I}init$:

$$\mathcal{I}init : \forall p.\texttt{PC}[p] = \texttt{PC}_0 \wedge \texttt{Y} = \bot$$

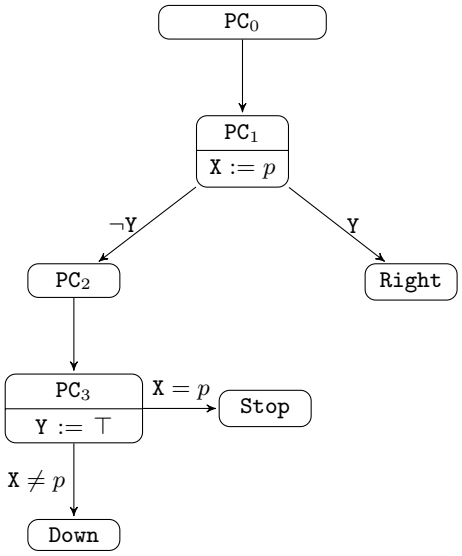

Fig. 2. Automaton for a process p representing the splitter with updates to shared variables attached to the nodes and conditions labeled to edges

The six transitions of the automaton are described by the six formulas spl_{xxx} given in Figure 3. Each formula relates the values of state variables before and after the transition. We denote by $\mathtt{X}'$ the value of the variable $\mathtt{X}$ after the execution of the transition. For instance, transition spl_0 should read as: if there exists a process p such that $\mathtt{PC}[p]$ contains $\mathtt{PC}_0$, then update $\mathtt{PC}[p]$ to $\mathtt{PC}_1$ and variable $\mathtt{X}$ to p.

According to the conditions previously stated, proving the safety of the splitter amounts to checking that states satisfying one of the following three formulas are not reachable:

$$\varphi_1 : \exists ij.\ \ i \neq j \ \wedge\ \mathtt{PC}[i] = \mathtt{Stop} \ \wedge\ \mathtt{PC}[j] = \mathtt{Stop}$$
$$\varphi_2 : \forall i.\mathtt{PC}[i] = \mathtt{Down}$$
$$\varphi_3 : \forall i.\mathtt{PC}[i] = \mathtt{Right}$$

The reachability analysis in MCMT is performed by running a symbolic backward algorithm. Starting from a formula describing the system's unsafe condition, its pre-images are iteratively computed for all transitions. Pre-images that are subsumed by already visited nodes are not expanded anymore. This process ends either when a formula in a node intersects the initial formula $\mathcal{I}nit$ or when there is no more pre-image to compute.

An important result about array-based systems is that pre-images of cubes (existentially quantified conjunction of literals) are computable and can be represented as union (disjunction) of cubes [**?**]. Thus, starting from a cube, the backward reachability analysis produces only cubes and is therefore automat-

$$
\begin{array}{lll}
spl_0 : & \exists p. & \begin{array}{l} \texttt{PC}[p] = \texttt{PC}_0 \ \wedge \\ \texttt{PC'}[p] = \texttt{PC}_1 \ \wedge \ \texttt{X'} = p \end{array} \\
spl_{right} : & \exists p. & \begin{array}{l} \texttt{PC}[p] = \texttt{PC}_1 \ \wedge \ \texttt{Y} \ \wedge \\ \texttt{PC'}[p] = \texttt{Right} \end{array} \\
spl_1 : & \exists p. & \begin{array}{l} \texttt{PC}[p] = \texttt{PC}_1 \ \wedge \ \neg\texttt{Y} \ \wedge \\ \texttt{PC'}[p] = \texttt{PC}_2 \end{array} \\
spl_2 : & \exists p. & \begin{array}{l} \texttt{PC}[p] = \texttt{PC}_2 \ \wedge \\ \texttt{PC'}[p] = \texttt{PC}_3 \ \wedge \ \texttt{Y'} = \top \end{array} \\
spl_{stop} : & \exists p. & \begin{array}{l} \texttt{PC}[p] = \texttt{PC}_3 \ \wedge \ \texttt{X} = p \ \wedge \\ \texttt{PC'}[p] = \texttt{Stop} \end{array} \\
spl_{down} : & \exists p. & \begin{array}{l} \texttt{PC}[p] = \texttt{PC}_3 \ \wedge \ \texttt{X} \neq p \ \wedge \\ \texttt{PC'}[p] = \texttt{Down} \end{array}
\end{array}
$$

Fig. 3. Splitter transition system

able. For instance, the pre-image of the cube φ_1 by spl_{stop} is the following formula φ'_1 which describes the states from which a state characterized by φ_1 can be reached by taking the transition $spl_{stop}(i)$ (where the parameter i indicates which process is concerned by the transition) :

$$\varphi'_1 : \exists ij.\ \ i \neq j \ \wedge \ \texttt{X} = i \ \wedge \ \texttt{PC}[i] = \texttt{PC}_3 \ \wedge \ \texttt{PC}[j] = \texttt{Stop}$$

The termination of this reachability analysis is guaranteed as long as one can exhibit a well-quasi-ordering on the set of cubes generated during the algorithm [?].

Concerning the second and third formulas φ_2 and φ_3, they are not cubes as they contain universal quantifiers. The computation of their pre-images will introduce existential quantifiers. For example, the pre-image of φ_2 by $spl_{down}(j)$ is the following φ'_2 formula:

$$\varphi'_2 : \exists j \forall i.\ \ i \neq j \implies \texttt{X} \neq j \ \wedge \ \texttt{PC}[j] = \texttt{PC}_3 \ \wedge \ \texttt{PC}[i] = \texttt{Down}$$

and the pre-image of φ'_2 by the same transition will give the new formula φ''_2:

$$\varphi''_2 : \exists jk \forall i. \begin{array}{l} i \neq j \neq k \implies \texttt{X} \neq j \ \wedge \ \texttt{X} \neq k \ \wedge \\ \texttt{PC}[j] = \texttt{PC}_3 \ \wedge \ \texttt{PC}[k] = \texttt{PC}_3 \ \wedge \ \texttt{PC}[i] = \texttt{Down} \end{array}$$

From φ'_2 and φ''_2, it may seem obvious that the reachability analysis of φ_2 will generate an infinite sequence of formulas where new existentially quantified processes will be piled up, leading to the impossibility of reaching a fixpoint and thus terminating.

For those reasons, the MCMT framework is restricted to the analysis of cubes. However, as illustrated by the properties φ_2 or φ_3, some problems involve formulas that are not cubes and that need to be handled.

As mention in the introduction, there exists techniques for extending MCMT to universal quantifiers. In [**?**], a syntactic transformation is proposed which can be interpreted as the implementation of a crash-failure model. The main idea is to view a universal formula $\forall i.\varphi$ as an infinite conjunction and to over approximate it as a *finite* conjunction $\exists i_1, \ldots, i_n.\varphi(i_1) \wedge \cdots \wedge \varphi(i_n)$, by considering that, except for those n processes, all other processes crashed before reaching the states described by this formula. For instance, using this technique, φ_2 could be seen as a cube ψ_2 of the form

$$\psi_2 : \exists i.\mathtt{PC}[i] = \mathtt{Down}$$

by considering that the number of processes that did not crash is exactly one. Computing the pre-image of ψ_2 is immediately simpler but we can see with utter certainty that this state is not unsafe if more than one processes are involved in subsequent transitions. From this example it is obvious that protocols that are not fault-tolerant (like the splitter) would produce wrong results.

3 Reasoning about Universal Cubes in MCMT

In this section, we illustrate BRWP, an extended version of the backward reachability of MCMT for reasoning about universal quantifiers using the splitter given in Section 2. Our extension applies to *universal cubes* (u-cubes) which are formulas of the form $\exists \boldsymbol{i} \forall \boldsymbol{j}.\mathcal{C}(\boldsymbol{i}, \boldsymbol{j})$, where $\mathcal{C}(\boldsymbol{i}, \boldsymbol{j})$ is a conjunction of literals parameterized by two vectors $\boldsymbol{i}$ and $\boldsymbol{j}$ of distinct process variables. We proceed in three steps to reason about u-cubes.

Step 1 : Reachability analysis in a finite domain. Instead of considering the parameterized case, we first restrict the domain of processes to a finite set of process identifiers (denoted in the rest of the paper by the symbols $\#_1$, $\#_2$, ...). The number chosen for the cardinality of the domain is arbitrary, but in our case studies we fix the domain to contain only 3 or 4 processes.

Fixing the cardinality allows us to instantiate the universal part of u-cubes and convert them to cubes. For instance, in a domain restricted to 3 distinct processes $\#_1$, $\#_2$ and $\#_3$, the formula φ_2 is transformed in the following cube $\varphi_2^{\#3}$ (with no quantifiers):

$$\varphi_2^{\#3} : \mathtt{PC}[\#_1] = \mathtt{Down} \ \wedge \ \mathtt{PC}[\#_2] = \mathtt{Down} \ \wedge \ \mathtt{PC}[\#_3] = \mathtt{Down}$$

From these cubes, we run the (traditional) backward reachability algorithm of MCMT, bounded by the finite cardinality of the domain of processes. Thereby, for instance, the first pre-image of $\varphi_2^{\#3}$ by $spl_{down}(\#_1)$ is the following $\varphi'^{\#3}_2$ formula:

$$\varphi'^{\#3}_2 : \mathtt{PC}[\#_1] = \mathtt{PC}_3 \ \wedge \ \mathtt{X} \neq \#_1 \ \wedge \ \mathtt{PC}[\#_2] = \mathtt{Down} \ \wedge \ \mathtt{PC}[\#_3] = \mathtt{Down}$$

It is important to remark that $\varphi'^{\#3}_2$ has the same number of processes as $\varphi^{\#3}_2$. Indeed, the cardinality of the domain prevents us to add new (existential) quantifiers.

If the reachability algorithm terminates with a pre-image that intersects the initial formula, then we can conclude that the system is unsafe in the parameterized case. Otherwise, if a fixpoint is reached (which is the case for the splitter), we can only conclude that the property is valid for the chosen number of processes, and we proceed to Step 2.

Step 2 : Generalising invariants. To go further and prove the properties defined by u-cubes for the parameterized case, we try to exploit (a subset of) the pre-images computed in Step 1 by trying to generalise those formulas for an infinite domain. This is done by abstracting process constants by existential or universal quantified variables.

The problem of generalising a pre-image computed in Step 1 is that it can characterize unreachable states in a finite domain but reachable ones in an infinite domain (which seems normal since this kind of algorithms are not fault-tolerant). For instance, consider the previous formula $\varphi'^{\#3}_2$ obtained by computing the pre-image of $\varphi^{\#3}_2$ by $spl_{down}(\#_1)$. The states described by this formula are unsafe (and unreachable from $\mathcal{I}nit$) if the domain is limited to three processes, but they becomes safe if a fourth process exists as it could be in any state of the automaton ($\mathtt{PC}_0$, $\mathtt{PC}_1$, $\mathtt{Right}$,...) as shown by the graph in Figure 4.

To check if a pre-image represents unreachable states in an infinite domain, we first transform it into a cube by replacing process constants with existential variables, then we replay the reachability algorithm of MCMT. If this cube is shown to be unreachable, then we keep it for Step 3. Otherwise, we transform the pre-image as a u-cube by only abstracting with existential variables the process constants involved in a transition and using universal quantifiers for abstracting the other constants. The u-cubes generated by this generalisation technique are safe (but nevertheless less informative) invariants that we keep for Step 3.

For instance, the pre-image $\varphi'^{\#3}_2$ is first generalised as a cube by abstracting the process constants $\#_1$, $\#_2$ and $\#_3$ by three existentially quantified variables p_1, p_2 and p_3:

$$\varphi'^{\exists}_2 : \exists p_1 p_2 p_3 \ \begin{array}{l} p_1 \neq p_2 \neq p_3 \ \wedge \ \mathtt{X} \neq p_1 \ \wedge \ \mathtt{PC}[p_1] = \mathtt{PC}_3 \ \wedge \\ \mathtt{PC}[p_2] = \mathtt{Down} \ \wedge \ \mathtt{PC}[p_3] = \mathtt{Down} \end{array}$$

Running a backward reachability from $\varphi'^{\exists}_2$ shows that it describes states reachable from $\mathcal{I}nit$. Therefore, we can filter this formula out as it is actually safe and can't be treated as an invariant of the system.

Now, when looking closely at the pre-image $\varphi'^{\#3}_2$, it appears that process $\#_1$ has been used by a transition when $\#_2$ and $\#_3$ remained untouched. In terms of

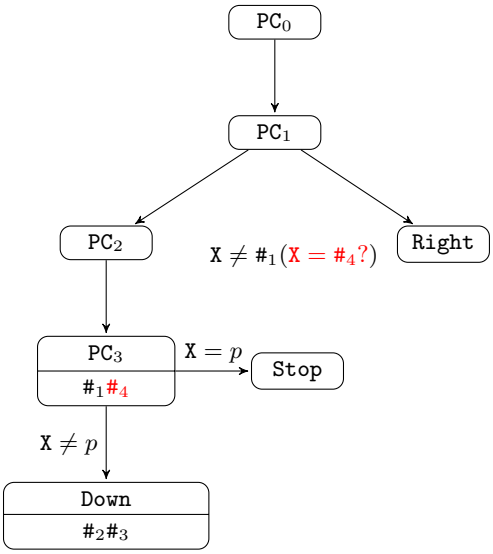

Fig. 4. Processes $\#_2$ and $\#_3$ are in Down and process $\#_1$ is in $\mathtt{PC}_3$, ready to go in Down, but a fourth process $\#_4$ could be in all other states

quantifiers, this can be semantically captured by (1) adding a fresh existential variable p_1 for representing $\#_1$ and (2) by representing processes $\#_2$ and $\#_3$ with the same universally quantified variable p_2. Therefore, $\varphi'^{\#3}_2$ can be generalised by the following u-cube

$$\varphi'^{\exists\forall}_2 : \exists p_1.\forall p_2.p_1 \neq p_2 \implies \mathtt{X} \neq p_1 \ \wedge \ \mathtt{PC}[p_1] = \mathtt{PC}_3 \ \wedge \ \mathtt{PC}[p_2] = \mathtt{Down}$$

which represents states that are unsafe if there exists a process p_1 in $\mathtt{PC}_3$ such that $\mathtt{X} \neq p_1$ and that all other processes are in Down.

Step 3 : Deductive verification. Given a property φ, the result of Steps 1 and 2 is a set of (u-)cubes $\{\mathcal{I}^{\exists\forall}_{k_1}, \ldots, \mathcal{I}^{\exists\forall}_{k_p}\}$ representing invariants of the original system computed from the finite backward reachability of φ.

To prove φ, our last step consists in proving the following conjunction ψ using a deductive verification technique.

$$\psi : \varphi \wedge \mathcal{I}^{\exists\forall}_{k_1} \wedge \cdots \wedge \mathcal{I}^{\exists\forall}_{k_p}$$

For that, we translate the array-based parameterized transition system, as well as the formula ψ, in the input language of a deductive verification engine.

Considering the impressive number of back-ends (SMT or TPTP solvers) supported by the Why3 platform, we have chosen to translate array-based systems to WhyML, the input language of Why3. However, the translation requires great attention as (1) the semantic gap between array-based systems and WhyML is

important, and (2) the way transition systems are represented may have a strong impact on the deductive engine (see Section 5).

4 Formalising BRWP

The implementation of BRWP is given in Algorithm 1. It's an extended version of the backward reachability procedure of MCMT for reasoning about universal cubes.

This algorithm takes as input a formula φ and an integer c. It starts by initiating two variables, $\mathcal{V}$, the set of the visited nodes initially empty and $\mathcal{Q}$, the queue of pending nodes initialised with the instantiated version $\varphi^{\#c}$ of φ. The formula $\varphi^{\#c}$ is instantiated as seen in Step 1 with a cardinality fixed to c. `BRWP` iteratively computes the transitive closure of pre-images $\textsc{FinitePre}^*(\varphi^{\#c})$ until it reaches one of the two following termination conditions :

- the safety check (line 7) fails which means that the treated node corresponds to a possible initial state and thus that the system is unsafe
- there are no more nodes in $\mathcal{Q}$ which means that a fixpoint has been reached and the system is safe (for a finite domain)

If the first termination condition has been reached, the system is not safe for a finite number of processes and can not be safe for an infinite number of them. However, if the second condition has been reached, the visited nodes need to be treated to correspond to the infinite domain as seen in Step 2. When these filtering and generalisation have been computed (see next subsections for a description of this step), the new invariants are delivered to a deductive verification engine by calling the function `Check_inductive_invariant` (see Section 5).

4.1 Generalisation and Filtering

The code of the generalisation and filtering function `Generalize_and_filter` is given in the Algorithm 2. It takes as input the set $\mathcal{V}^{\#c}$ of instantiated formulas computed during the finite backward reachability. Its goal is to transform those formulas in cubes by renaming the processes and binding them to existential quantifiers. However, before doing so, a simplification step (function `Simplify`) is performed since the finitness of our domain allows us to transform multiple differences in an equality. For example, considering again the formula $\varphi'^{\#3}_2$ seen in Section 3:

$$\varphi'^{\#3}_2 : \mathtt{PC}[\#_1] = \mathtt{PC}_3 \ \wedge \ \mathtt{X} \neq \#_1 \ \wedge \ \mathtt{PC}[\#_2] = \mathtt{Down} \ \wedge \ \mathtt{PC}[\#_3] = \mathtt{Down}$$

Its pre-image by transition $spl_{down}(\#_2)$ gives the following formula:

$$\mathtt{X} \neq \#_1 \ \wedge \ \mathtt{X} \neq \#_2 \ \wedge \ \mathtt{PC}[\#_2] = \mathtt{PC}_3 \ \wedge \ \mathtt{PC}[\#_3] = \mathtt{Down}$$

In this case, the fact that we're facing a finite domain actually helps us. Since there are only three processes, the literals $\mathtt{X} \neq \#_1$ and $\mathtt{X} \neq \#_2$ implies that $\mathtt{X} = \#_3$. This formula is thus transformed first as follows:

Algorithm 1: Backward reachability and deductive verification

```
   Variables :
       V: visited nodes
       Q: work queue
 1 function BRWP(φ, c) : begin
 2 |   φ^#c := Instantiate (φ, c);
 3 |   V:= ∅;
 4 |   push(Q, φ^#c);
 5 |   while not_empty(Q) do
 6 |   |   φ^#c := pop(Q);
 7 |   |   if (I ∧ φ^#c sat) then
 8 |   |   |   return unsafe
 9 |   |   else if (φ^#c ⊭ V) then
10 |   |   |   V:= V ∪ {φ^#c};
11 |   |   |   push(Q, FinitePre(φ^#c, c));
12 |   |   end
13 |   end
14 |   S1, V' := Generalize_and_filter(V);
15 |   S2:= Universal_Generalization(V');
16 |   Check_inductive_invariant(φ ∧ S1 ∧ S2)
17 end
```

$$\mathtt{X} = \#_3 \ \wedge \ \mathtt{PC}[\#_2] = \mathtt{PC}_3 \ \wedge \ \mathtt{PC}[\#_3] = \mathtt{Down}$$

then, it is generalised as the following cube:

$$\exists p_2, p_3. \ \ \mathtt{X} = p_3 \ \wedge \ \mathtt{PC}[p_2] = \mathtt{PC}_3 \ \wedge \ \mathtt{PC}[p_3] = \mathtt{Down}$$

After this generalisation and simplification transformation has been performed, the cube φ thus obtained is given to the same backward reachability engine (`BWD`), but this time without the finite domain constraint. If the model checker returns safe, φ is saved in the set variable $\mathcal{S}_1$ for the deductive verification engine and the instantiated formula $\varphi^{\#c}$ is filtered out from the set $\mathcal{V}$ of formulas to be given to the second generalisation algorithm implemented in `Universal_Generalization`.

4.2 Universal Generalisation

The code of the function `Universal_Generalization` is given in Algorithm 3. Similarly to the previous generalisation function, `Universal_Generalization` takes as input the set $\mathcal{V}^{\#c}$ of instantiated cubes.

To explain the main part of this algorithm, we illustrate its uses in Figure 5, starting from the following property φ of the splitter

$$\varphi : \forall p. \mathtt{PC}[p] = \mathtt{Down}$$

Algorithm 2: Generalise and filter

```
1  function Generalize_and_filter(V^#c) : begin
2      V := V^#c;
3      S1 := ∅;
4      forall φ^#c ∈ V^#c do
5          Δ∃ := ⊤ ;
6          forall p ∈ V⃗^#c do
7              v := Fresh_Variable();
8              Δ∃ := Δ∃ ∧ v;
9              Replace(p, v, φ^#c); /* Replace all occurrences of the process
                p with the fresh process v */
10         end
11         D = Distinct(Δ∃);            /* All variables are different */
12         φ = Δ∃ ∧ D ∧ φ^#c;
13         Simplify(φ);
14         if BWD(φ) safe then
15             S1 := S1 ∪{φ};
16             V := V \{φ^#c}
17         end
18     end
19     return (S1, V)
20 end
```

which, after instantiation (for instance when c is 3), is given to the generalisation function as the following formula:

$$\varphi^{\#c} : \texttt{PC}[\#_1] = \texttt{Down} \wedge\ \texttt{PC}[\#_2] = \texttt{Down} \wedge\ \texttt{PC}[\#_3] = \texttt{Down}$$

This formula is first tagged with a vector of processes $\overrightarrow{V}^{\#c}$ describing which processes are from the same quantifier. Here, $\overrightarrow{V}^{\#3} = \{\#_1, \#_2, \#_3\}_\forall$, where the meaning of the annotation $\forall$ is that the 3 processes come from the universal quantifier. When computing the pre-image from the transition $\texttt{spl}_{down}$: $\exists i.\texttt{PC}[i] = \texttt{PC}_3\ \wedge\ \texttt{X} \neq i\ \wedge\ \texttt{PC'}[i] = \texttt{Down}$, we end up with the new formula

$$\varphi_1^{\#3} = \texttt{PC}[\#_1] = \#_3\ \wedge\ \texttt{X} \neq \#_1\ \wedge\ \texttt{PC}[\#_2] = \texttt{Down}\ \wedge\ \texttt{PC}[\#_3] = \texttt{Down}$$

and the new vector $\overrightarrow{V}^{\#3} = \{\#_1\}_\exists, \{\#_2, \#_3\}_\forall$. The reasoning behind this comes from the fact that transitions in Cubicle are existentially quantified. Thus, since processes $\#_2$ and $\#_3$ have not been involved in the transition, they remain attached to the universal quantifier. On the contrary, process $\#_1$ becomes attached to a new existential quantifier.

When generalised, all literals containing an existential-tagged processes (we use the notation $\overrightarrow{V}^{\#3_\exists}$ to denote this set of variables) are kept with their processes being renamed in new distinct existential processes and all the literals containing an universal-tagged process (we use the notation $\overrightarrow{V}^{\#3_\forall}$ to denote this set of

variables) are merged into one literal quantified by a fresh universal process. For instance, the formula $\varphi_1^{\#3}$ is generalised as follows:

$$\exists p_1.\forall p_2.p_1 \neq p_2 \implies \texttt{PC}[p_1] = \texttt{PC}_6 \wedge \texttt{PC}[p_2] = \texttt{Down}$$

Algorithm 3: Universal Generalisation

```
1  function Universal_Generalization(V^{#c}) : begin
2      V := ∅;
3      forall φ^{#c} (tagged by V⃗^{#n}) ∈ V^{#c} do
4          Δ∃ := ⊤ ;
5          Δ∀ := ⊤ ;
6          forall p ∈ V⃗^{#n∃} do
7              v := Fresh_Variable();
8              Δ∃ := Δ∃∧ v;
9              Replace(p, v, φ^{#c}); /* Replace all occurrences of the process
                p with the fresh process v */
10         end
11         if V⃗^{#n∀} ≠ ∅ then
12             v := Fresh_Variable();
13             Δ∀ := v;
14             Remove_and_Replace(p, v, φ^{#c});     /* Remove all the literals
                parameterized by processes from V⃗^{#n∀} and adds a new
                literal parameterized by v */
15         end
16         binop := if Δ∀ = ⊤ then ∧ else ⟹ ;
17         D = Distinct(Δ∃, Δ∀);          /* All variables are different */
18         φ = Δ∃ ∧ Δ∀ ∧ D binop φ^{#c};
19         V := V ∪{φ};
20     end
21 end
```

5 Deductive Verification

The last function call `Check_inductive_invariant`$(\varphi \wedge \mathcal{S}_1 \wedge \mathcal{S}_2)$ of BRWP requires the help of a deductive verification engine. In our implementation, we are using the Why3 platform [**?**].

Given a program $\mathcal{P}$ and its specification $\mathcal{T}$ (a set of theories, program invariants and properties), Why3 tries to check that $\mathcal{P}$ satisfies $\mathcal{T}$ by performing an inductive invariant check with a compositional reasoning and a weakest-precondition (WP) engine. Verification conditions generated by the WP calculus of Why3 are discharged to a large number of automatic or interactive solvers (SMT, TPTP, Coq, etc.).

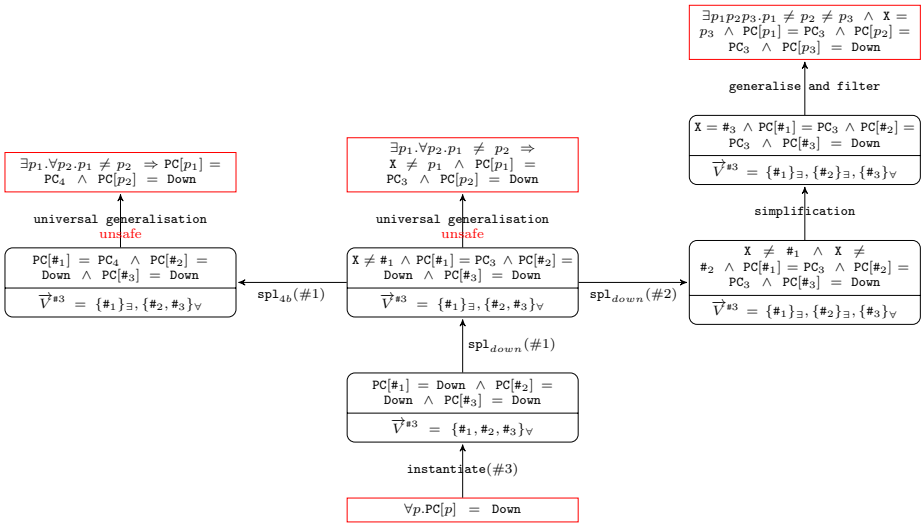

Fig. 5. First nodes and their simplification, filtration and generalisation for the splitter

The implementation of `Check_inductive_invariant` is essentially based on the translation of array-based transition systems to WhyML, the input language of Why3. However, the gap between the semantics of MCMT and WhyML is important. Indeed, Why3 is a platform designed to work with sequential, deterministic and terminating programs, while the semantics of array-based transition systems is concurrent, non-deterministic and non-terminating. To see how to bridge the gap between these two languages, we illustrate our translation using the splitter example (Section 2).

State declaration. Our encoding starts with types declarations. The type `proc` of processes is represented by integers (`int` in WhyML). The system's state is encoded by a record with two mutable variables `x` and `y`, as well as an array `pc` (implicitly indexed by integers) containing values of type `state`.

```
type proc = int
type system = {
  mutable x : proc;
  mutable y : bool;
  pc : array state;
}
```

Initial states. The initial formula of the splitter defines initial states with the following formula $\mathcal{I}nit$

$$\mathcal{I}nit : \forall p . \mathtt{PC}[p] = \mathtt{PC}_0 \wedge \mathtt{Y} = \bot$$

where only `Y` and `PC[]` are given a value, the other variable `x` can contain an arbitrary value. Since Why3 expects every variable to be initialised, we give to `x` a random value in the range of possible values.

```
let s = {
  y = false;
  pc = Array.make _n PC0;
  x = Random.random_int _n;
} in
```

Infinite execution. The semantics of an array-based transition system is given by a single infinite loop which repeatedly execute two steps:

1. evaluate all the guards of transitions, given the current values of the global state
2. arbitrarily choose one of the commands whose guard is true and execute it, updating the variables

Translating *infinite* loops in Why3 is problematic, in particular when one want to check invariants when exiting it. A solution to this problem is to consider that the loop ends when it reaches a bound given as a parameter of our system. The resulting program in Why3 is then bounded by the number of processes and the number of steps allowed in the loop.

```
let splitter1 (_n : int) (maxsteps : int) : system
  requires { 0 < _n }
  ...
  =
  (* ... *)
  while ( !nbsteps < maxsteps ) do
    variant { maxsteps - !nbsteps }
    incr nbsteps;
    (* ... *)
  done;
  s
end
```

Nondeterminism. There are two sources of nondeterminism in array-based systems. The first one can be illustrated by considering the following transitions t_1 and t_2:

$$t_1 : \exists p.\texttt{PC}[p] = \texttt{PC0} \wedge \texttt{PC'}[p] = \texttt{PC1}$$

$$t_2 : \exists p.\texttt{PC}[p] = \texttt{PC0} \wedge \texttt{PC'}[p] = \texttt{PC2}$$

If $\texttt{PC}[p] = \texttt{PC0}$ for some process p, then both transition can be triggered, resulting in a state where $\texttt{PC}[p]$ equals to `PC1` or `PC2`.

In order to mimic this nondeterminism in Why3, we add a coin toss to each translation of a transition's guard. This coin toss does not need to be specified, it just allows Why3 to explore all the possibilities.

```
val coin () : bool
if coin () && pc.[i] = PC0
then pc.[i] <- PC1
if coin () && pc.[i] = PC0
then pc.[i] <- PC2
```

The second source of nondeterminism comes from the fact that, at each step of the loop, transitions need to be taken by random unique processes. The Why3 program thus needs to know that it can take the transition of its choice with the processes of its choice if the guards hold true with such processes.

This is done by specifying a function that takes two arguments, the max number of processes n and the number of needed processes k (*e.g.* the max number of processes involved in transition guards, updates, etc.). This function ensures that all the processes it creates will be different. There is no need to implement it as we just use its specification to help the deductive verification. Why3 having difficulty with lists and algebraic data types to reason inductively, the result returned by this function is an array of the size of the number of processes needed. The value k is determined by the maximum number of parameters (processes) in a transition. In the splitter case, k will then be equal to 1.

```
val k_random (k:int) (n:int) : (result:array int)
  requires { 0 <= k }
  requires { k <= n }
  ensures { length result = k }
  ensures { forall i j:int. 0 <= i < n /\ 0 <= j < n /\ i <> j ->
              result[i] <> result[j] }
  ensures { forall i:int. 0 <= i < n -> 0 <= result[i] < n }
```

Invariants. Finally, as it is (when all the transitions have been added), this file can not be proven by Why3 since it lacks important loop invariants. The algorithms to find such invariants are independent of BRWP and by lack of space, we omit to describe them. In our implementation, these invariants are found during the backward reachability loop using the BRAB technique of Cubicle [**?**,**?**], a model checker based on MCMT. Those invariants are automatically added to the Why3 file as invariant formulas.

```
while ( !nbsteps < maxsteps ) do
  invariant { 0 <= s.x < _n }
  invariant { forall _p1 : int. 0 <= _p1 < _n /\
                           s.x = _p1 -> s.pC[_p1] <> Down }
  invariant { exists _p1 : int. 0 <= _p1 < _n /\ s.pC[_p1] <> Down }
  (* ... *)
```

6 Conclusion and Perspectives

In this paper, we have presented an extension of the MCMT framework for reasoning about universal cubes, that formulas with both existential and universal quantifiers. Our approach tightly combines the backward reachability algorithm of MCMT with a deductive verification engine.

We have implemented our framework in the Cubicle model checker, with the help of the Why3 platform for program verification. Our first experiments are very promising as we have been able to prove automatically algorithms like the splitter which were out of scope the Cubicle model checker.

As future work, we plan to design an even more tight integration between our backward reachability algorithm and a weakest-precondition calculus in order to implement a complete roundtrip loop between these algorithms.