A Gentle Introduction to Verification of Parameterized Reactive Systems

Nicolas Féral[1] and Alain Giorgetti[1,2][0000−0002−0990−9611]

[1] Université de Franche-Comté, F-25000 Besançon, France
[2] Institut FEMTO-ST, UMR 6174 CNRS, France
alain.giorgetti@femto-st.fr (corresponding author)

Abstract. An introduction to symbolic model-checking and deductive verification techniques is offered to Master's students at the University of Franche-Comté. This teaching is carried out remotely. It is built around the use of the Cubicle model-checker and the Why3 platform. It shows how to verify the safety of distributed reactive systems which are parameterized by the number of processes run in parallel, when this safety is expressed as non-reachability of critical states.
For remote lab sessions, a virtual machine containing the Cubicle software and the Why3 platform is provided to the students. Examples of reactive systems are specified by the students in the input language of Cubicle and in the WhyML language of Why3. With Cubicle, students use a backward reachability algorithm that discovers dangerous states of these systems by tracing back their transitions to critical states. Safety is proven if no initial state is reached. When a system is safe, Cubicle produces a certificate in WhyML, which contains an invariant synthetized by Cubicle. This certificate can be executed with Why3 to prove that the system indeed preserves this invariant. Students also learn how to directly specify reactive systems in the WhyML language, using a primitive for non-determinism between transitions and between processes that evolve inside each transition.

Keywords: formal methods · parameterized reactive systems · safety · reachability · Why3 · Cubicle

1 Introduction

This article presents a course taught at the University of Franche-Comté, entitled *Specify and Verify*, which allows students to discover the notion of reactive system and to use an implementation of a technique of symbolic model checking, in order to verify the safety of parameterized reactive systems. The parameterization of these systems relates to the number of identical processes which run in parallel. Safety is expressed here as simply as possible, as the unreachability of critical states. The challenge is to find an invariant of the global system which excludes all the critical states. The verification then consists in proving – preferably in an automatic way – that all the initial states of the system satisfy the invariant, that all its transitions preserve it, and that no critical state satisfies

it. The objective being to verify systems of any size, independently of the number of processes, the applied technique is *symbolic model-checking*, here based on representations of states and transitions in first-order logic. In this course, the description of parameterized reactive systems and the verification of their safety use the *model checker* Cubicle [3] and the deductive verification platform Why3 [2].

This course is part of a curriculum that has been designed for many years to be delivered entirely by distance learning. It is primarily intended for students who cannot attend face-to-face classes, for various professional or personal reasons. Students must study at home, alone and at different times. The main pedagogical objective is that the students become able to apply to simple systems a process of *formal specification* and *computer-assisted verification* of the consistency of the specifications.

In addition to the verification of reactive systems, this course also covers functional specification and deductive verification of simple imperative programs, such as a function calculating the factorial or performing a search for elements in an array. This article does not address this subject, which is more classical than the verification of parameterized reactive systems, and already covered in other articles, for example [1].

This article is written by A. Giorgetti, teacher at the University of Franche-Comté, manager and tutor of the course *Specify and Verify* since 2018, and N. Féral, student of this module in 2020-21. The graduation project of N. Féral, about automated verification of parameterized reactive systems, has greatly contributed to the introduction of Cubicle in this course, from September 2021. The educational material presented in this document can be downloaded from the professional web page `https://members.femto-st.fr/alain-giorgetti/en` of the second author.

Section 2 situates this course in the master curriculum and describes its main characteristics, then its pedagogical progression. Section 3 details the notions covered during the course. Section 4 describes the working environment provided to the students, in order to assimilate the course. Course evaluation procedures are discussed in Section 5. Finally, Section 6 shares remarks and considerations inspired by the design and development of this course.

2 Description of the teaching unit

The *Specify and Verify* module is part of the last semester of the *advanced computing and applications* and *software development and validation* tracks of the computer science master's degree at the University of Franche-Comté. Both tracks are entirely delivered remotely. They benefit from a long experience and recognized know-how of the University of Franche-Comté in distance education, since 1966, for the preparation and delivery of national university diplomas. All teaching infrastructure is digitally managed and accessible online, using the Moodle e-learning platform (`https://moodle.org`).

The main Moodle page for the course *Specify and Verify* provides access to written course material, exercises and homework sheets, corrected annals, but also to a discussion forum between students and with teachers. The latter provide regular tutoring, answering students' questions on this forum and by e-mail. Students are strongly encouraged to respect a provided study schedule, similar to the content of Table 1.

Table 1. Pedagogical progression of the course

Week	Type	Topic
1	Lesson 1	Introduction to model-based verification and proof of programs
1	Exercises	Lesson 1 assimilation exercises
2	Lesson 2	Specification and verification of parameterized reactive systems with Cubicle
2	Homework 1	Creation of the lab environment (with Docker), specification and verification of a first reactive system
2	Exercises	Exercises related to Lesson 2: specification and verification of examples of parameterized reactive systems with Cubicle
3	Lesson 3	Logic of Why3 (propositional and first-order logic with basic and inductive types), first contact with this platform
3	Exercises	Assimilation of the logic of Why3: propositions, quantifiers, predefined types, formalization of simple problems and properties
4	Deepening	Study of the personalized correction of the homework 1 and its provided solution
4	Lesson 4	Specification and deductive verification of simple imperative programs in WhyML language
4	Exercises	Exercises related to Lesson 3
4	Homework 2	Study of WhyML certificates generated by Cubicle, verification of imperative programs with Why3
5	Revisions	Deepening and preparation for the final exam, with the help of provided annals
6	Deepening	Study of the personalized correction of the homework 2 and its provided solution

The course consists of 4 lessons. The first one is an introductory chapter that places the model verification approach within the context of the software and system development process. It distinguishes between declarative models (specifications) and operational models. It defines and distinguishes reactive systems, open or closed, which mainly interact with their environment, and transformational systems, which carry out a calculation. This introduction also defines the methods of model-checking and deductive verification, more commonly called "program proof". The second lesson presents the model checker Cubicle, its specification language and its methods to verify the safety of a parameterized reactive system. The third lesson presents the deductive verification platform Why3 and its input language, named WhyML. The fourth lesson teaches the proof of (small) imperative programs specified and implemented in the WhyML language.

The exercises allow the students to assimilate the lessons, by looking in the course material for the relevant notions for their resolution, and the tasks to be carried out to achieve the required objectives. Each exercise sheet is associated with a lesson of the course, as detailed in Table 1. A solution of the exercises is distributed separately, sometimes after a small delay, to encourage students to solve the exercises without consulting it. Thus, the study of an exercise sheet corresponds to a tutorial session carried out remotely.

A homework subject is made up of exercises, sometimes a little more exploratory than the course's assimilation exercises. Homework assignments must be returned within a set deadline. Then, each student's homework is marked and annotated by the tutors, and returned with a standard solution and a detailed scale, so that the students can learn from their mistakes, identify their difficulties and measure their progress. In the first homework subject, the first exercise helps the students to set up the working environment, as detailed in Section 4. A second exercise proposes to the students to model and verify with Cubicle a reactive system described in natural language. The second homework subject is dedicated to the use of the Why3 platform. In a first exercise, the study of the system modeled during the first homework is completed by the generation with Cubicle of a certificate of proof of safety for the Why3 platform. Students should be able to identify the different parts of the certificate, in particular the invariant synthetized by Cubicle and the logical goals for its preservation proof. In a certificate, each transition is formalized by a logical relation between any state s of the system before the transition and any state s' after the transition. This primed notation for states after the transitions is also used to define by a primed predicate I' the invariant I after the transitions. Thus, reading certificates introduces students to the before-after relational semantics of action systems, and to a definition of the notion of inductive invariant formalized in first-order logic. A second exercise can require the programming of a reactive system in WhyML, then the design and the realization of a proof of an invariant for this system. In addition to reactive system verification, the second homework may also contain imperative program verification exercises.

3 Teaching content

This section details the essential notions on the modeling and verification of systems studied in the module *Specify and Verify*, and then the educational documentation provided for the use of Cubicle and Why3 when carrying out the practical questions of the exercises and homeworks.

3.1 Taught concepts

The course distinguishes between transformational systems, which calculate results from data and according to an algorithm, and reactive systems, which carry out few calculations, but a lot of control of the interactions between their components and with their environment. The methods for specifying these two types

of systems are different, since the transformational systems are specified using pre- and post-conditions, while reactive systems are specified declaratively (usually by temporal properties) and operationally, by a set of transitions between states, guarded by conditions of transition, called *guards*.

The reactive systems studied are said to be *parameterized*, because they consist of any (fixed) number of processes, and *uniform*, because all these processes are identical. The executions are assumed to respect the interleaving hypothesis, according to which at most one process evolves simultaneously with each event. The behavior of the system derives from the interactions of the processes with each other and the environment. These reactive systems are said to be *closed* when the hypotheses concerning the environment and the events likely to occur are taken into account in the modelling, for example using non-deterministic transitions. The studied properties are the simplest safety properties, which require that no execution of the system reaches certain states, called *critical*. A typical example, used in the course and in the first exercises, is the *mutual exclusion property*, which requires exclusive access to a shared resource between all processes. The safety of a system is established by first looking for an *invariant candidate*, which is a characterization of a subset of the states of the system excluding the critical states, then by formally demonstrating that this formula is not satisfied by any critical state and that it effectively constitutes an *(inductive) invariant* of the system, satisfied by all the initial states of the system and preserved by the action of each of its transitions. During homework, the study of Cubicle certificates is an opportunity to better assimilate this definition of an inductive invariant. Indeed, all these certificates contain an explicit formalization, in first-order logic, of the condition on the initial states and of the condition of preservation by any transition.

3.2 Cubicle

The second lesson of the course presents the Cubicle tool (`https://cubicle.lri.fr/`), its main features and its input language. Cubicle is an open source model checker resulting from the thesis work of A. Mebsout [10]. This work improves and implements techniques of model checking modulo theory and invariant synthesis [5,6,7,8]. Confidence in Cubicle results is enhanced by producing certificates which are separately verifiable with Why3.

Cubicle's input language makes it possible to define variables and arrays that model the data of a system and its processes, to specify the sets of initial and critical states, as well as the transitions whose activation is conditioned on the existence of an n-tuple of processes allowing a guard to be crossed. This language is documented in the thesis of A. Mebsout [10] and in the materials for a course given by S. Conchon in a school for young researchers [4]. The second lesson of the course *Specify and Verify* describes pedagogically and in detail the syntax and semantics of the fragment of this language useful for this course, illustrating them with simple examples. This exempts the students from having to consult external sources of documentation.

Cubicle is dedicated to the specification and verification of parameterized reactive systems of any size whose states are described by global variables and arrays. It allows the modeling of systems evolving in a discrete and non-deterministic way, under the action of guarded transitions. In order to favor the verification process, the internal subtleties of Cubicle are deliberately not detailed in the course, which only mentions that Cubicle uses a backward reachability algorithm, which goes back the transitions from critical states to try to reach the initial states. The discovered states, even if they are not critical, are *dangerous states* insofar as they are the starting point of at least one path leading to a critical state.

The course illustrates the modeling and verification process with Cubicle using a variant of the distributed mutual exclusion algorithm created and named "bakery" by L. Lamport [9]. This system models a bakery, in which customers obtain a numbered ticket that defines the order of waiting before placing an order with the baker. The customers, in any number, are the processes. In the original algorithm, the order of access to the baker is defined according to a lexicographic order relating to the ticket numbers and, in the event of a tie, the numbers identifying the customers. In the course variant, the ticket numbers issued to customers are distinct and are sufficient to determine the order of service. Therefore, the process that can access the critical section can always be known, if it exists.

In Cubicle, the states of the global system for "bakery" are defined by the following types and variables:

```
type status = WA | SE | AS
var Ticket : int
array CustomerStatus[proc] : status
array CustomerTicket[proc] : int
```

The `Ticket` variable, of integer type, stores the value of the last ticket delivered by the ticket dispenser. The `CustomerStatus` array stores the status of each customer, among the three states of the enumerated type `status`, described later. The `CustomerTicket` array stores the ticket value for each customer. For a customer without a ticket, this value is arbitrary.

All customers behave the same way, as follows. A customer in the requesting state `AS` (for `AS`king) can spontaneously obtain a unique numbered ticket. The number on this ticket is obtained by incrementing the counter `Ticket`. Once in possession of this ticket, this customer enters the waiting state `WA`. This action is formalized by the following Cubicle transition, parameterized by the process identifier `i`:

```
transition getTicket (i)
  requires { CustomerStatus[i] = AS } {
  CustomerStatus[i] := WA;
  Ticket := Ticket + 1;
  CustomerTicket[i] := Ticket + 1;
}
```

The access to the shared resource is formalized by the Cubicle transition `access` reproduced in Listing 1.1. If there is at least one customer waiting, then

the baker serves the customer who has the smallest ticket (among the tickets of the waiting and served customers). The customer then switches to the served state `SE`.

Listing 1.1. Access to the served state, in Cubicle syntax

```
transition access (i)
  requires { CustomerStatus[i] = WA &&
    forall_other j.
      CustomerStatus[j] = WA || CustomerStatus[j] = SE
      => CustomerTicket[i] <= CustomerTicket[j] } {
  CustomerStatus[i] := SE;
}
```

Once served, a customer releases her/his place by switching to the `AS` state of customers likely to (re)request a ticket. This action is formalized by a simple Cubicle transition, named `leave`, which is not detailed here.

As specified in the following Cubicle clause, parameterized by the process identifier `i`, all customers are initially assumed to be ticket requesters (status `AS`) and to have a ticket numbered 0, which is also assumed to be the last ticket delivered by the ticket dispending machine. It may seem problematic that all customers initially have the same ticket number, but this is not an issue because no transition uses the ticket number of a requesting customer to determine the effects of a transition or to evaluate a guard. As the system evolves, the tickets owned by waiting and served customers are unique before being examined to decide who is the next served customer.

```
init (i) {
  CustomerStatus[i] = AS && CustomerTicket[i] = 0 &&
  Ticket = 0
}
```

Critical states are declared with one or more `unsafe` specifications. The following example expresses the existence of two distinct processes `i` and `j` in the critical state `SE`, which fails mutual exclusion.

```
unsafe (i j) {
  CustomerStatus[i] = SE && CustomerStatus[j] = SE && i <> j
}
```

When the modeling is finished, the execution of Cubicle in command line outputs a result concerning the safety of the system, reproduced in Figure 1. In addition to the `SAFE` or `UNSAFE` verdict, Cubicle indicates sets of critical or dangerous states computed by the backward reachability algorithm. These are the *nodes* 1 to 8. A trace indicates a succession of transitions which makes it possible to reach a set of critical states from a set of dangerous states.

When the system is not safe, Cubicle returns an error trace indicating the transitions to follow to reach a critical state from an initial state. The student

```
guest@c33289fc8d66:~/data$ cubicle Bakery.cub
node 1: unsafe[1]
node 2: access(#2) -> unsafe[1]
node 3: access(#1) -> access(#2) -> unsafe[1]
node 4: getTicket(#2) -> access(#2) -> unsafe[1]
node 5: getTicket(#2) -> access(#1) -> access(#2) -> unsafe[1]
node 6: access(#1) -> getTicket(#2) -> access(#2) -> unsafe[1]
node 7: leave(#2) -> access(#1) -> getTicket(#2)
  -> access(#2) -> unsafe[1]
node 8: access(#2) -> leave(#2) -> access(#1)
  -> getTicket(#2) -> access(#2) -> unsafe[1]
...
The system is SAFE
```

Fig. 1. Result of a safety check with Cubicle.

can check this trace by reconstructing the evolution of the system manually. Cubicle does not offer a tool to automate this trace analysis.

Cubicle also makes it possible to generate a certificate of this proof, in the input language of the Why3 platform. A certificate contains a logical description of the verified system, a candidate invariant and goals to prove, which formalize that the candidate invariant excludes critical states, and that it is indeed an invariant of the system, satisfied by all initial states and preserved by all transitions of this system.

Implementing verification with Cubicle is therefore straightforward. If it is possible to write the specifications of a system respecting the Cubicle input language syntax, then the system is verifiable with Cubicle. Otherwise, you have to turn to other tools, such as the Why3 platform, which is more generic.

3.3 Why3 platform

The Why3 platform enables proofs of propositional or first-order logic formulas, or of conformity between imperative or functional programs and their logical specification. Its language, called WhyML, is more general and more complex than that of Cubicle. Its use allows students to discover and implement the notions of contract, loop invariant, etc, and to develop several skills. First, they learn to solve decision problems with Why3, formalizing them as lemmas or goals. For simplicity, the course is limited to the case where these logical formulas concern variables of predefined Boolean or integer type, or of enumerated type defined by the user, and/or of array type storing data of these types. Then, the students learn to specify in WhyML a simple imperative program contract, manipulating the same data types, and to annotate its loops, until making the verification of this contract automatic with the Why3 platform. In addition, students learn to read and modify certificates generated by Cubicle. Finally, students learn to specify reactive systems directly in WhyML, possibly beyond the expressiveness limits of Cubicle, and to verify their safety with Why3, as detailed in Section 3.4.

3.4 Specification and verification of reactive systems only with Why3

Through its input language and its internal mechanisms for invariant strengthening and WhyML certificate generation, Cubicle greatly facilitates the specification and verification of parameterized reactive systems. However, it is a black box, applicable only to systems whose sets of critical states can be described by formulas, called *cubes*, whose syntax is limited to conjunctions of literals existentially quantified by identifiers of distinct processes. How to check a property of a system that cannot be specified in this way?

So that this course remains introductory, it does not provide a general answer to this question, which would be too technical, but provides intuitions based on examples. In the first place, some advanced exercises, or even some questions of an exam subject, may propose to the students to modify a certificate generated by Cubicle, for example to add a property to check. However, this first approach has a major limitation, inherent to Cubicle's translation of transitions into logical relations: the user cannot execute these specifications. But WhyML is also a programming language, of which a fragment is directly executable, and a larger fragment is executable by extraction in OCaml. On the example of the bakery algorithm, the rest of this section presents a way to describe reactive systems by WhyML programs, and then discusses the issue of their verification.

The following code proposes a way to describe the states of the bakery system in WhyML, for a number of customers fixed by the constant `n`, which must be a positive integer (condition `Pos`, required because the WhyML type `int` corresponds to relative mathematical integers).

```
type status = WA | AS | SE
val constant n : int
axiom Pos : n > 0
type sys = {
  mutable ticket : int;
  customerStatus : array status;
  customerTicket : array int
} invariant {
  invariant_candidate ticket customerStatus customerTicket
} by {
  ticket = 0;
  customerStatus = make n AS;
  customerTicket = make n 0
}
```

Cubicle variables and arrays are grouped here as fields of a record of type `sys`. These fields are all *mutable*, i.e. modifiable in place, either because they are declared with the `mutable` keyword, or because they are arrays, always mutable in WhyML. The type `sys` is defined with a *type invariant*, which imposes conditions on its fields. These conditions are hidden here behind the predicate `invariant_candidate`. Some technical conditions require the ticket numbers to be non-negative integers, and the two arrays `customerStatus` and `customerTicket` to

be of the same size `n`. The other conditions characterize any set of supposedly safe states, with respect to given critical states.

The WhyML language requires that any type with an invariant be declared with a `by` clause that justifies that the type is not empty, by describing an inhabitant of it. Here, we suggest as inhabitant the initial state of the bakery system, with the provided WhyML function `make`, which constructs an array whose elements all have the same value. When verifying a type declaration with an invariant, Why3 tries to prove that the state described by its `by` clause satisfies the type invariant described by its `invariant` clause. By this design choice, without anything else to write, we obtain the verification condition that the invariant candidate is true in the initial state of the bakery system.

In order to formalize in WhyML the non-deterministic choice of a process which satisfies the guard of a transition, the abstract function `any_int_where`, specified by the following code, is provided to students. It models the non-deterministic choice of an integer `i` satisfying the condition `(p s)`, for any given executable predicate `p` and any inhabitant `s` of any type `'a`. (In WhyML, as in many functional languages, the application of the function `f` to the argument `x` is denoted `f x`, instead of `f(x)`.)

```
val any_int_where (s: 'a) (p: 'a -> int -> bool) : int
  requires { exists i. p s i } ensures { p s result }
```

As illustrated by an example below, in the WhyML code of a transition parameterized by a single process, the type `'a` will be the type of the states, the variable `s` will be a state and the predicate `p` will be an executable version of the guard of this transition. The function `any_int_where` only exists if there exists at least one integer `i` satisfying the condition `p s`. This is required by the precondition `(exists i. p s i)`. Under this condition, the postcondition `(p s result)` expresses that the function returns such an integer, represented in this expression by the reserved word `result` of the WhyML language.

In the presence of a declaration with the keyword `val`, Why3 admits that the declared object – here, a function – exists, without requiring or providing any justification for this existence. However, if such a function cannot exist, assuming the contrary would make the logic inconsistent. In order to show that this is not the case, we can provide the following implementation of this function.

```
let any_int_where (s: 'a) (p: 'a -> int -> bool) : int
  requires { exists i. p s i } ensures { p s result }
= any int ensures { p s result }
```

The expression `any ... ensures ...` non-deterministically evaluates to a value that satisfies the logical formula after the keyword `ensures`, when this value is assigned to the dummy variable `result`. For this expression, Why3 always produces a verification condition that this logical formula is indeed satisfiable. Due to the precondition (`requires` clause), this condition of existence of the function `any_int_where` is easily discharged by the Why3 platform.

Then, each transition of the system is implemented as a WhyML function, as in the example reproduced in Listing 1.2. The guard of this transition is that there is a customer in the `AS` state. This guard is defined by the predicate

Listing 1.2. Transition of a customer obtaining a ticket, in WhyML

```
let predicate p_getTicket (s: sys) (c: int)
= 0 <= c < length s.customerStatus &&
  s.customerStatus[c] = AS
let getTicket (s: sys) : sys
  requires { exists i. p_getTicket s i }
= let c = any_int_where s p_getTicket in
  s.customerStatus[c] <- WA;
  s.customerTicket[c] <- s.ticket + 1;
  s.ticket <- s.ticket + 1;
  s
```

`p_getTicket`, which is also a Boolean function, thanks to the keyword `let` which allows to use it in programs. A technical point here is that equality (`=`) is defined in WhyML as a logical predicate for any type, but does not exist by default as a Boolean function on user-defined types, such as `status`. For executability, a Boolean equality on the type `status` must be implemented, for example by the following code.

```
let (=) (x y: status) : bool ensures { result <-> x = y }
= match x with
  | WA -> match y with WA -> True | _ -> False end
  | AS -> match y with AS -> True | _ -> False end
  | SE -> match y with SE -> True | _ -> False end
  end
```

A second technical point concerns guards that include a quantified condition on processes, such as the guard of the `access` transition. In order for the corresponding predicate to be executable, students are provided with the following Boolean function for universal quantification over a bounded integer interval.

```
let predicate forAll (s:'a) (p:'a->int->bool) (l u:int)
  ensures { result <-> forall i. l <= i <= u -> p s i }
= for j = l to u do
    invariant { forall i. l <= i < j -> p s i }
    if not (p s j) then return False
  done;
  True
```

The `any_int_where` function is similar to the `ANY ...WHERE ...THEN ...END` construct of the B language, and to the `any` keyword of WhyML, whose execution in a program realizes a non-deterministic choice of a typed value satisfying a given condition. Instead of using this instruction, we have preferred to provide and suggest to use the `any_int_where` function, in order to facilitate various implementations of non-determinism. Indeed, it is simpler to associate an implementation to a single function than to have to replace each `any` expression by an implementation.

Each transition is a WhyML function parameterized by the complete system (named `s` in this example, of type `sys`) and which returns the new state of this system, of the same type `sys`. When the Why3 tool verifies such a function, it

tries to show that the returned state satisfies the invariant of type `sys`, under the hypothesis that its parameter `s` satisfies this invariant. This verification condition is exactly the second condition for this type invariant to be an invariant of the transition system, without anything else to specify about it. Without this use of a Why3 type with an invariant, it would have been necessary to repeat the invariant candidate as a precondition and postcondition of each transition, which is more cumbersome and presents a risk of error by omission.

When the system is fully programmed, the student verifies its safety by calling the SMT solvers available through the Why3 platform. If the proof of safety succeeds, the exercise comes to an end. Otherwise, the student has to verify her/his code and specifications, to reinforce the invariant candidate or to question the capacities of the SMT solvers to discharge the verification conditions. As these last two tasks are difficult, the exercises are accompanied by advice.

A *reinforcement* of an invariant candidate is any additional formula that approximates more accurately the reachable states, and whose conjunction with the invariant candidate is expected to form an inductive invariant. As first invariant candidate, it is natural to choose the negation of a characteristic predicate for critical states. For any mutual exclusion algorithm, such as the bakery system, this invariant is the mutual exclusion property, saying that no two distinct processes simultaneously use the shared resource. For the bakery system, this property of simultaneous non-existence of two served customers is not an inductive invariant. Indeed, this property holds for instance in the state with two customers, one served and the other one waiting and holding a smaller ticket. From this state, the second client can be served by the transition `access` (whose code is reproduced on Listing 1.1), leading to the critical state. Fortunately, the combination of the other transitions and the initial states makes these dangerous states unreachable. The Cubicle output

```
node 2: access(#2) -> unsafe[1]
```

on the third line of Figure 1 means that Cubicle identifies these dangerous states and choose their negation as first invariant reinforcement. Nodes 3 to 8 reproduced in Figure 1 similarly correspond to sets of dangerous states found by Cubicle, and used by Cubicle to produce other reinforcements included in the certificate. Their conjunction with the initial invariant candidate forms an inductive invariant.

When we choose as reinforcements the invariants synthesized by Cubicle, and provided to us in the certificate it generates, and we encode all the guards with `let predicate`, then Why3 automatically proves that the resulting invariant is preserved by the `getTicket` and `leave` transitions, but not by the `access` transition. This last proof becomes automatic if we separately define the guard by a predicate and a Boolean function, along the pattern reproduced in Listing 1.3.

Figure 2 illustrates the proof results when the system is specified in this way. The verification condition `sys'vc` is the condition that the initial states satisfy the invariant, while the verification conditions associated with the transitions deal with the preservation of the invariant.

Listing 1.3. Pattern for the access transition, in WhyML

```
predicate p_access (s: sys) (i: int) = ...
let g_access (s: sys) (i: int) : bool
  ensures { result <-> p_access s i } = ...
let access (s : sys) : sys
  requires { exists i. p_access s i }
= let c = any_int_where s g_access in
  s.customerStatus[c] <- SE;
  s
```

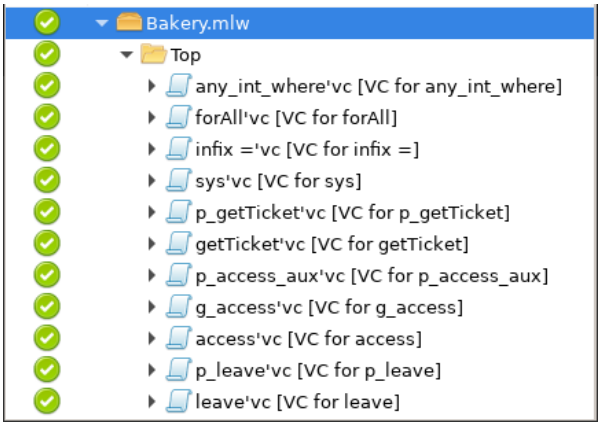

Fig. 2. Verification of a bakery system with the Why3 platform

Generally speaking, finding reinforcements is all but an easy task, beyond the expected level of the students at the end of this course. If the exercise could be handled with Cubicle, students are asked to adapt the invariants that Cubicle wrote in its proof certificate. Otherwise, some exercise questions suggest reinforcements in natural language, which the student only has to formalize in WhyML.

In conclusion, while the process of verification of parameterized reactive systems with Cubicle is relatively simple, this process with the Why3 platform is more complex, since reinforcements may be required to complete the proofs. Therefore, exercises and homework are necessary to facilitate its learning by students.

4 Virtual machine for labs

Each year, the first exercise of the first homework guides the students to build their lab environment, in the form of a Docker (`https://www.docker.com`) container. At the end of this exercise, the students have a virtual machine (called a *Docker container*) in which they can run Cubicle (version `1.1.2`) and Why3

(version `1.4.0`) software command line, but also the graphical interface of Why3. The virtualization technology Docker is available for Linux, macOS, and Windows. The Docker container being the same virtual machine for all students, it limits installation problems and allows tutors to reproduce the actions of the students identically, and thus better help them to use the tools. Using this container allows students to do homework with their personal computer without disrupting their work habits. A working directory is shared between the host machine and the container. Files for Cubicle and the Why3 platform stored in this directory can be accessed and edited from the host machine or in the running container.

The Docker image of this working environment is formally described in a `Dockerfile` provided to students. It expands the Docker image `registry.gitlab.inria.fr/why3/why3:1.4.0` distributed by Inria, which contains the Why3 platform (version `1.4.0`) and the three CVC4 SMT solvers (version `1.7`), Alt-Ergo (version `2.0.0`) and Z3 (version `4.8.4`). The `Dockerfile` complements this image with Opam (*OCaml package manager*, `https://opam.ocaml.org`) and an installation of the Cubicle software with Opam. For Windows users, a VcXsrv server (`https://sourceforge.net/projects/vcxsrv`) is used as an X server, for the graphical interface of the Why3 platform. The creation and starting of the working environment are made easier thanks to provided scripts for Linux and macOS, and `.bat` batch files for Windows.

No integrated development environment (IDE) is suggested to edit input files for Cubicle. Any text editor is suitable, even without syntax highlighting, since Cubicle's input language is very readable and the requested models are achievable in a few dozen of lines.

The creation of the lab environment being an essential step, the students are invited to communicate on the mutual aid forum the difficulties they encountered. The exercise represents 8 points out of 20 in the grade for the first homework. Its evaluation is carried out on the basis of a report on the application of the homework subject instructions and the student's participation in the mutual aid forum on this activity.

5 Evaluations

We first describe how students are evaluated, and then how the teaching unit is perceived by its students.

5.1 Student assessment

The exercises are not graded. Homeworks being works done without time limit and without supervision, their grade are not taken into account in student assessment. These grades, however, have an indicative value: they quantify the quality of the content of the returned assignments and inform students of the level expected at the exam.

By using verification tools during their works at home, the students see themselves whether their answers are correct. These verification tools could also be used to automatically generate a grade based on the number of successful verifications. However, such an automated grading tool is of little interest, for several reasons. First, automated verification for complete specifications being rare, it can only be one element of evaluation among others, with a small coefficient. The other evaluation criteria relate to the understanding of the concepts, which requires a human analysis producing personalized written recommendations. Finally, the number of students returning homework being low (from 10 to 20), automated grading would not represent a significant time saving for teachers.

The students are evaluated on 20 points, during a written final exam (on paper), supervised and lasting 2 hours. The examination must be carried out without using electronic devices and without consulting documents other than the examination subject sheets. The fragments of the Cubicle and Why3 language syntaxes useful for this subject are recalled in an appendix at the end of the subject. Thus, the only noticeable difference between the exam requirements and what the students are trained to do during the semester is that they cannot run the Cubicle and Why3 tools during the exam to detect errors in the codes they are writing on their exam copy. This limitation encourages students to produce (almost-)correct code more rationally, without resorting too much to the empirical trial-and-error cycle. The negative effect is minimized by an evaluation that ignores minor syntax errors. Students are informed of the exam conditions at the start of the semester, and can practice them thanks to annals provided with their answers. The *Specify and Verify* module counts for 3 ECTS credits (*European Credit Transfer System*) among 30 for one semester of the master.

5.2 Course assessment

Once the course, the exercises, the homework and the final exam have been practiced by the students, an evaluation is carried out to ensure that the module finds a favorable reception. No automatic evaluation system is used, because the numbers of students and hours devoted to this teaching are small. The evaluation of the course is carried out directly by questioning the students, either during an in-person review meeting, or through a digital forum during the COVID-19 pandemic. It appears, for example, that the speed and efficiency of the use of the Cubicle *model checker* is appreciated by the students. Difficulties encountered by the students during exercises and homework, expressed as questions in the forum, are taken into account as they arise, in the form of additional explanations or course modifications. Final exam results also help determine course improvements for the following year.

6 Discussion

The prerequisites to follow the formal verification process taught in the *Specify and Verify* module are few. The students have generally already practiced

the modeling of systems with states evolving under the effect of transitions. Reminders concerning discrete event systems and first-order logic, as well as short introductions to the Cubicle and Why3 tools, are sufficient to facilitate the acquisition of new knowledge and know-how.

If reading and writing formulas in first-order logic is an uncommon practice in the context of students' previous programming activities, writing specifications in logic remains accessible to them, since the guards of the transitions to be formalized correspond to simple logical formulas. The difficulty in proving certain parameterized reactive systems lies above all in the design of an invariant approximating the non-dangerous states, given the specifications. This difficulty is reduced thanks to the automated invariant reinforcement mechanism implemented in Cubicle.

The formal approach to specification and verification of parameterized reactive systems provides students with a concrete example of *symbolic model checking* based on the decision procedures implemented in SMT (Satisfiability Modulo Theory) solvers. However, no temporal logic is taught to specify the properties to be checked, since the only property dealt with is safety, which is reduced to a state reachability analysis. The usefulness of this verification by exploration (of sets) of states (model checking), and of its automation, are highlighted by offering students examples of systems whose reachable states are difficult to predict intuitively, while their transitions are simple to define. In particular, the search for reinforcements of invariants is not very intuitive, which illustrates the difficulty of predicting the states reached by the system.

The teacher who designs the exercises must ensure that safety is not trivially ensured by the system, but is really a property emerging from the system specifications. Otherwise, checking it becomes obvious and its formal verification loses its interest. For instance, mutual exclusion does not constitute an emergent property of a system whose transitions would explicitly check (in their guard) that no other process owns the critical resource.

When an invariant has been proven, it is possible to continue the study of the system to verify other complementary properties. The proven invariant is completed with an additional property and a new verification of the preservation of the invariant is carried out. If the modified invariant remains preserved, then the additional property is verified for all the states reached by the system.

7 Conclusion

The *Specify and Verify* module is built around the use of the model checker Cubicle and the Why3 platform. These tools have proven to be suitable for teaching the verification of the safety of simple parameterized reactive systems. This practical approach to verification is not very demanding in terms of prior knowledge and formalization skills. It provides students with a concrete example of model checking. However, when a system is safe, but it is necessary to carry out reinforcements oneself to find an invariant, the effectiveness of verification with the Why3 platform is greatly reduced.

Although it is possible to use the invariant reinforcement technique applied by Cubicle in a black box, it would be interesting to add in this module a lesson on invariant reinforcement (semi-)algorithms, such as those of theses of J.-F. Couchot [5] and A. Mebsout [10], also to show how to formalize them in WhyML and to verify some of their properties with Why3. This would add to the module a complementary aspect of formal semantics, in particular about non-determinism.

Acknowledgments

This project is supported by the EIPHI Graduate School (contract ANR-17-EURE-0002). We thank the three anonymous reviewers for their comments and suggestions that helped us improve our original manuscript.

References

1. Blazy, S.: Teaching deductive verification in why3. Formal Method Teaching 2019 **LNCS 11758** (2019)
2. Bobot, F., Filliâtre, J.-C., Marché, C., Melquiond, G., Paskevich, A.: The Why3 Platform Release 1.5.1 (2022), `http://why3.lri.fr/manual.pdf`
3. Conchon, S., Goel, A., Krstić, S., Mebsout, A., Zaïdi, F.: Cubicle: A parallel SMT-based model checker for parameterized systems. In: CAV'12 (Computer Aided Verification). LNCS, vol. 7358, pp. 718–724. Springer (2012), `http://www.lri.fr/~conchon/publis/conchon-cav2012.pdf`
4. Conchon, S.: Model checking, part 1 : Model checking modulo theories (mcmt) (2015), cours de l'EJCP (École des Jeunes Chercheurs en Programmation), `https://www.lri.fr/~conchon/EJCP/ejcp-mcmt.pdf`
5. Couchot, J.-F.: Vérification d'invariants de systèmes paramétrés par superposition. Ph.D. thesis, Laboratoire d'Informatique de l'Université de Franche-Comté, Besançon, France (April 2006)
6. Couchot, J.-F., Giorgetti, A., Kosmatov, N.: A Uniform Deductive Approach for Parameterized Protocol Safety. In: ASE '05: Proceedings of the 20th International Conference on Automated Software Engineering. pp. 364–367. IEEE (2005)
7. Ghilardi, S., Ranise, S.: Goal-directed invariant synthesis for model checking modulo theories. In: TABLEAUX (Automated Reasoning with Analytic Tableaux and Related Methods). LNCS, vol. 5607, pp. 173–188. Springer (2009)
8. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. Logical Methods in Computer Science **volume 6, issue 4** (2010)
9. Lamport, L.: A New Solution of Dijkstra's Concurrent Programming Problem **17**(8), 3 (1974)
10. Mebsout, A.: Invariants inference for model checking of parameterized systems. Thèse de doctorat, Université Paris Sud - Paris XI (2014), `https://tel.archives-ouvertes.fr/tel-01073980`