Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 743–754

DOI: 10.15439/2015F190
ACSIS, Vol. 5

Behavior-Preserving Abstraction of ESTEREL Programs

Nir Koblenc
Department of Mathematics and Computer Science
Open University of Israel
Ra'anana, Israel
Email: skoblenc@gmail.com

Shmuel Tyszberowicz
School of Computer Science
Academic College of Tel-Aviv Yaffo
Tel-Aviv, Israel
Email: tyshbe@tau.ac.il

***Abstract*—Reactive programs often control safety-critical systems, thus it is essential to verify their safety requirements. ESTEREL is a synchronous programming language for developing control-dominated reactive systems, and XEVE is a verification environment that analyzes circuit descriptions generated from ESTEREL programs. However, a circuit generated by the ESTEREL compiler from non-pure ESTEREL program often displays behaviors which may violate safety properties even when the source program does not. We introduce an automatic abstraction process for ESTEREL programs developed to tackle this problem. When the process is applied to a program augmented with observers to monitor the program's behavior, it results in a pure program that preserves the behavior of the source program, replacing value-carrying objects with pure signals. We have built a prototype tool that implements the abstraction and used it to purify control programs and robotic systems.**

***Index Terms*—Verification, Abstraction, Reactive Systems, ESTEREL.**

I. INTRODUCTION

Reactive systems are computer systems that continuously react to their environment at a speed determined by the environment. Most industrial real-time systems are reactive [3]. ESTEREL[1] is an imperative concurrent language for the development of industrial-strength reactive systems, which is especially well-suited for control-dominated reactive systems such as real-time process control systems, embedded systems, communication protocols, peripheral drivers, human-machine interfaces, and others [4]. ESTEREL belongs to the family of *synchronous languages*— languages that are based on the synchrony hypothesis, which states that a program instantaneously reacts to its input. Control is assumed to takes no time and thus output is broadcast right when the input arrives. The notion of simultaneity is captured by the concept of *event*, which is a set of simultaneous occurrence of (possibly valued) signals [3]. ESTEREL offers significant advantages over traditional languages used in industrial settings [5], such as verification (due to the precise mathematical semantics and the existence of verification tools and techniques), reduction of testing (automatic verification covers safety requirements), high-level abstraction, and better code structuring. A full definition of the language, as of version 5.91, can be found in [4].

ESTEREL programs communicate with their environment by means of signals and sensors. Signals can be used both for input and output, and may convey values; sensors can only be input and always convey values. A signal can be either present or absent; no such concept exists for sensors. A signal that carries values is called a *valued signal*, and a signal that does not convey values is called a *pure signal*.

Reactive systems are often used to control safety-critical systems. Hence they require rigorous design methods, and formal verification must be considered [3]. A complete, consistent and precise specification is constructed, often employing formal language to avoid ambiguity. While this process is very potent in detecting errors already at the formal specification development phase [6], there still is a risk that there would be inconsistencies between the formal specification and the eventual product. Even while the specification is formally verified, the product itself may still be erroneous, and we want to verify that the program satisfies its safety properties.

Verification by observers [7] is an approach to verify code. *Observers* are program modules monitoring the program, testing that a property is satisfied and broadcasting specific signals when the property is violated. The observers are composed in parallel to the original program, and the resulting program is compiled using an ESTEREL compiler into a finite automaton. The properties of the automaton are verified using tools such as the X ESTEREL VERIFICATION ENVIRONMENT (XEVE) [7], reducing the verification problem to reachability problem in finite automata – finding if there exists an execution trace from the initial state to a state emitting one or more of those special observer signals.

The XEVE verification environment requires the program to be compiled into Berkeley Logic Interchange Format (BLIF), a logic-level hardware hierarchical circuit description in a textual form; however, ESTEREL compiler, as of version

This work has been partially supported by GIF (grant No. 1131-9.6/2011). The paper is based on the master's thesis of the first author. For the thesis and the prototype tool see [1]. This paper extends a very short paper we have published in [2].

[1]We refer to ESTEREL v5.92, which we use for teaching. The toolset and the documentation are available at `http://www-sop.inria.fr/esterel.org/filesv5_92/`.

5.91[2], can either compile pure ESTEREL programs into BLIF files without changing their semantics, or, using the `-soft` option, abstract the data and compile only the control aspect into BLIF [9]. Pure ESTEREL programs only handle pure signals, i.e., they involve no valued signals, types, constants, functions, procedures, tasks, or variables [9]. In this work we collectively refer to valued signals, sensors, and variables as *valued objects*. The problem is that XEVE may fail to verify properly circuits generated from observed programs in which data is involved in the control when using the `-soft` option. The reason is that the abstraction might add behaviors not displayed by the original program, possibly including behaviors in which observer signals are emitted. Based on the false observation, the user might reject a program which is actually correct. For example, consider the following program:

```
module SpuriousError
   output Error;
   var v := 1 : integer in
      if (v <> 1) then
         emit Error
      end if;
      pause
   end var
end module
```

This program declares a variable `v` and initializes it to 1. It is obvious that the signal `Error` is never emitted because the condition tested by the `if` statement is never fulfilled. We can compile this program into a BLIF file using the `-soft` option, yet XEVE suggests that `Error` is possibly emitted, probably since the role of the data in the control flow is ignored and every transition dependent on run-time values is always enabled.

Many control schemes, however, receive numerical inputs, conduct numerical calculations, and emit numerical outputs. Example for such schemes are signal processors and closed-loop feedback controllers. We *purify* such programs to allow automatic verification of their properties. This is a transformation of ESTEREL programs handling valued objects into pure ESTEREL programs. It abstracts an unbound, concrete system that handles data by replacing objects that take values from theoretically-infinite domains with pure signals to receive a finite system. The abstraction preserves the external observable behavior of the original program, i.e. there is a correspondence in terms of inputs, outputs, and timings between the two programs. The difference between the two programs is that the abstract program uses pure signals where the original one employs valued signals, sensors, and variables.

Our approach is largely based on *predicate abstraction* [10]. Predicate abstraction is an automatic mapping of an unbounded system (the *concrete system*) to a finite system (the *abstract system*). An abstract system is defined by a concrete system and a finite set of predicates. Its states correspond to truth assignments to these predicates. The predicates define the *abstraction function*, which maps the states of the concrete system to the states of the abstract system. A state of the abstract system is reachable if it is an abstraction of a reachable concrete state. A user who wants to prove certain invariants supplies them as part of the predicate set. If the predicates stating the invariants are true in all reachable abstract states, then it means that the invariants hold in any reachable state of the concrete system. Applying this idea to ESTEREL, we automatically derive predicates about current and previous values of valued objects, of the form "the value of object x is in the range I". Each such predicate is translated to a pure signal. We can say that a state of the concrete system where such predicate is held is abstracted to a state of the purified program where the corresponding signal is present.

Usually, abstraction-based proof techniques are sound but not complete, since the abstraction is done such that every property proven to be satisfied by the abstract system has a concrete version which holds on the concrete system, yet the other way around is unnecessarily true [10]. However, verification using our abstraction technique is both sound and complete, since the abstract program adds no new behaviors to those displayed by the concrete program; in particular every pure signal is emitted by the observer-augmented concrete program if and only if it is emitted by its purified version. The abstraction alters the semantics of a few operations; however, these changes remain internal, i.e. the interaction with the environment is unaffected.

We suggest an algorithm that automatically purifies a certain group of programs and we characterize the class of programs verifiable using our technique. The main contribution of this work is extending the class of programs verifiable using observers with XEVE. We have implemented a prototype tool that purifies ESTEREL programs based on the algorithm described in Section II. This section also presents a running example of a *proportional controller*. We discuss the challenges we have faced when abstracting variables in Section III. The programs to which the method is applicable must comply with certain constraints, which are discussed in Section IV. Section V characterizes the class of programs to which the solution can be applied and provides test cases. We conclude and provide suggestions for future work in Section VI.

II. ESTEREL PROGRAM PURIFICATION

The XEVE verification environment takes as input BLIF files. Compiling an ESTEREL program into this format preserves the program's behavior only if the source code is pure. Hence, we have to substitute valued objects with pure signals and to modify the statements controlling the flow of the program and manipulating these objects to use pure signals instead. In this section we explain how to transform a program that complies with certain necessary constraints, into a pure program preserving the behavior of the original program.

We describe only the fundamentals of the algorithm.[3] We use a running example to demonstrate our method. It shows the application of our method to a *proportional controller*. This

[2] We chose to focus on ESTEREL v5 since it is free, suitable for teaching, and can be verified using the free XEVE tool which is part of the ESTEREL v5_92 distribution, whereas ESTEREL v7 [8] is commercial. XEVE is used in the industry [7].

[3]Due to lack of space we omit some details. The full description can be found in [1].

is a closed-loop feedback controller whose control signal is proportional to the *error* – the difference between the *set point*, also known as the *reference* (the ideal point) and the measured quantity under control, i.e., the control signal is calculated by multiplying the *error signal* by a *gain* [11]. Following is the ESTEREL code before its purification:

```
module PropMotor:
   input SampleTime;
   sensor Speed : double;
   output MotorForce : double;
   output AC_ON, AC_OFF;
   loop
      % proportional controller for the motor force
      present SampleTime then
         emit MotorForce (3.0 * (100.0 - ?Speed))
      end present;
      await SampleTime
   end loop ||
   loop
      % bang-bang controller for the cooling sub-system
      present MotorForce then
         if ?MotorForce > 270.0 or
            ?MotorForce < -270.0 then
            emit AC_ON
         else
            if ?MotorForce > -30.0 and
               ?MotorForce < 30.0 then
               emit AC_OFF
            end if
         end if
      end present;
      await tick
   end loop
end module
```

In addition to the proportional controller, the example uses a *bang-bang controller* that is composed in parallel and controls a cooling system. Bang-bang controllers are feedback controllers that switch abruptly between two states [11]. The controller receives a measured quantity of interest, and outputs a certain value if that quantity is above a certain threshold, and a different value otherwise.

The example displays a control system for a motor. The set point is 100 km/h; the measured quantity is the motor's current speed, received by the `Speed` sensor; the gain is 3.0. The output signal, `MotorForce`, is the force that the motor should produce, and is obtained by multiplying the difference between 100 km/h and the current speed by the gain. The motor is cooled by an air conditioning unit when exerting more than a certain amount of force (270.0). The air conditioning stops when the amount of force exerted by the motor drops below a certain threshold (30.0) (the difference between the thresholds is deliberate and used for hysteresis[4]). To start the air conditioning, it emits the `AC_ON` signal, and to stop the air conditioning, it emits `AC_OFF`.

We want to verify several safety properties of the given program using observers that monitor the system's behavior and emit special signals once one of these properties is violated. The observers are assembled in threads parallel to the main program. We would be able to verify that these signals are never emitted by compiling the observer-augmented program into BLIF format; however, we cannot do so as long as the program handles data other than pure signals.

[4]Hysteresis can be introduced by setting "dead zones" of no reaction around the set point in bang-bang controllers to avoid rapid on-off cycling [11].

The algorithm removes from the program all variables, sensors, and valued signals. Instead, we represent their values using pure signals. Each Boolean-valued object needs only one signal – that we call a *value signal* – to represent its value (present when true, absent when false). Since a Boolean signal actually has three states (absent, present and true, present and false) it takes another signal, which we call a *presence signal*, to denote whether the simulated Boolean signal is considered present or absent.

As for numerical-valued objects whose values range over infinite domains, we partition their domains into non-overlapping intervals, which we hereby call *ranges*, in such way that two goals are achieved. The first one is being able to decide any condition containing an occurrence of some numerical valued object by knowing the range within which that object's value resides. For example, the condition $?s > 3.0$ for a sensor s ($?s$ is the current value of s) can be decided if the information whether $?s \in (-\infty, 3.0]$ or $?s \in (3.0, +\infty)$ is available. The second goal is maintaining relationships between dependent objects. For instance, for the assignment $v := a * ?x + b$, where v is a variable, x is a valued signal and a and b are literal constants, we can determine the range within which the v's value would reside following the assignment based on the range of x. E.g., for $v := 2 * ?x + 1$ where $?x$ is in (1,3], the value of v is in $(2 \cdot 1 + 1, 2 \cdot 3 + 1] = (3, 7]$. Note that actually we calculate the ranges for x given the partition for v.

Our prototype tool works in two stages that run in a sequence by a shell script. In the first stage a standalone tool parses the source ESTEREL program according to v5_91 grammar specification found in [4] and outputs an intermediate file containing a list of syntax rules it identifies. The second stage is another program that reads that file and constructs a parse tree representation of the program. The tree data structure holds the information about all objects and statements of the source program, such that it is possible to reconstruct the program entirely from the tree. During the construction of the tree items that are of interest to the abstraction process, such as valued objects or statements that manipulate or test data are identified and stored in collections from which the abstraction algorithm can efficiently access them later. The abstraction is performed on the tree. The tool automatically calculates the predicates and performs the abstraction, implementing the algorithm discussed in this work. At the final step, a pure, abstract program is written to a target file based on the transformed tree. For sake of readability, we have edited in the paper the code produced by the tool.

After the source program is parsed, the abstraction algorithm starts with a pre-processing step to simplify the program when needed. Currently it expands each `sustain` statement (a statement emitting a specified signal in every instant once started and remains active forever) for a valued signal by a loop in which that signal is emitted in every instant, such that we can handle these statements just like instantaneous signal emissions (`emit` statements). This is also the place to perform other pre-processing activities; for example, for the simplicity of the algorithm, we require the user to guarantee a few preconditions the source program must fulfill (see Section IV) in order to apply the tool/algorithm to it; these assumptions can be obtained automatically by performing some pre-processing steps on the original program. This step can be expanded in future versions to include them as well.

We implement some interval-scalar arithmetic operations required by this algorithm. Let *I* be an interval, and *a* and *b* be scalars:

- Multiplication of an interval by a scalar: $a \cdot I$ (equivalent to scaling an interval), and
- Addition of a scalar to an interval: *I* + *b* (equivalent to translating an interval).

Additionally, we define a *product* of two partitions P_1 and P_2 as $\{I_1 \cap I_2 | I_1 \in P_1 \wedge I_2 \in P_2\}$. This is a "mutual refinement" of P_1 and P_2 by one another. We denote the partition of the domain of a numerical valued object *x* by PARTITION(*x*). The partitioning process starts from statements that assign a constant value to a valued object (i.e. *var* := *const* or `emit` *val_sig* `(`*const*`)`). The object's domain is partitioned according to the assigned constant. Suppose, e.g. that at some stage of the partitioning process PARTITION(*v*) = {(–∞, 1), [1, 1], (1, +∞)} for a variable *v*, once considering an assignment *v* `:= 2.0`, PARTITION(*v*) is refined to {(–∞, 1), [1, 1], (1, 2), [2, 2], (2, +∞)}. This refinement allows us to refer later to *v* = 2 (equivalent to *v* ∈ [2, 2]) as a predicate in our abstract system, which is true once the assignment takes place in the original program. Note that this is the product of {(–∞, 1), [1, 1], (1, +∞)} and {(–∞, 2), [2, 2], (2, +∞)}.

Next, we consider the Boolean data expression in which a valued object occurs. Let (*a***D*+*b*) *R c* be a Boolean data expression, where *a*, *b* and *c* are literals ($a \neq 0$), *D* is either the current or the previous (i.e. `pre(?`*x*`)`) value of *x*, all have the same data type, and *R* is a relational operator. We denote $k = \frac{c-b}{a}$. If *a* divides $c - b$ (i.e. $\lfloor k \rfloor = k$) or *x* is floating-point real, the partition of *x* is refined with the partition {(–∞, *k*), [*k*, *k*], (*k*, +∞)}; otherwise, *x*'s partition is refined with $\{(-\infty, \lfloor k \rfloor], [\lceil k \rceil, +\infty)\}$. During the partitioning process we distinguish `integer` objects from `float` and `double` objects. For `integer` objects, non-integer values are illegal, therefore – an interval that does not contain integers is irrelevant. Also, non-integer values can be excluded from ranges computed for `integer` objects. When we compute ranges for an object of type `integer`, if an interval (range) end is neither at $+\infty$ nor at $-\infty$, we round it to the nearest integer inside the interval, and close the end. For example, if we compute a range (−10, 3.5] for an `integer` object, then this range contains exactly the same values as [–9, 3] in the `integer` domain, hence (−10, 3.5] is replaced by [–9, 3].

The partitioning process continues based on dependency between valued objects. We consider a valued object *y* to be *dependent* on another valued object *x* if *x*'s current or previous value occurs in an assignment to, or an emission of, another valued object *y* (the data expression is of the form $a * D + b$ where *a* and *b* are literals and *D* denotes *x* for a variable *x*, `?`*x* for a sensor *x*, or either `?`*x* or `pre(?`*x*`)` when *x* is a signal). Suppose *y*'s partition is $\{Y_1, Y_2, ..., Y_n\}$, then *x*'s partition is refined with $\{{}^1\!/_a \cdot Y_1 - {}^b\!/_a, {}^1\!/_a \cdot Y_2 - {}^b\!/_a, ..., {}^1\!/_a \cdot Y_n - {}^b\!/_a\}$. The order by which these dependencies are considered is detailed in Section IV.

There are two valued objects in the example: the sensor `Speed` and the output signal `MotorForce`. Since `MotorForce` is set with `Speed`'s transformed value, `MotorForce`'s ranges are computed before `Speed`'s ranges. `MotorForce` is computed the following nine ranges: {(–∞, –270.0), [–270.0, –270.0], (–270.0, –30.0), [–30.0, –30.0], (–30.0, 30.0), [30.0, 30.0], (30.0, 270.0), [270.0, 270.0], (270.0, +∞)}. We denote them by $R_1^{\text{MotorForce}}$, ..., $R_9^{\text{MotorForce}}$, respectively. The only valued signal emission found in the entire program is: `emit MotorForce (3.0 * (100.0 – ?Speed))`. Let us denote this statement by E_1. Once executed, `?MotorForce` = –3.0 * `?Speed` + 300.0. Since `Speed` = 100.0 – `MotorForce` / 3, the partitioning provides the following ranges for `Speed`: {(–∞, 10.0), [10.0, 10.0], (10.0, 90.0), [90.0, 90.0], (90.0, 110.0), [110.0, 110.0], (110.0, 190.0), [190.0, 190.0], (190.0, +∞)}, denoted by R_1^{Speed}, ..., R_9^{Speed}. Note that when this emission is executed, `MotorForce` is in $R_1^{\text{MotorForce}}$ if and only if `Speed` is in R_9^{Speed}, `MotorForce` is in $R_2^{\text{MotorForce}}$ if and only if `Speed` is in R_8^{Speed}, and so forth.

Each range is matched with a pure *range signal*. For a valued object *x*, for which PARTITION(*x*) = $\{R_1^x, R_2^x, ..., R_n^x\}$, the ranges' corresponding signals are named `R1_`*x*, `R2_`*x*, etc. A configuration of the abstract, purified program in which `R`*i_x* is present corresponds to a configuration of the original, concrete program where the value of *x* is in R_i^x. In other words, an event in which `R`*i_x* is present in the abstract program stands for an event in which the value of *x* resides within range R_i^x in the concrete program. In addition, if *x* in the source program is a signal, an occurrence of any of its range signals in the abstract program means that *x* is present in the corresponding configuration of the concrete program.

When we declare range signals for a numerical input signal or a numerical sensor (replacing the original signal or sensor definitions), we also declare an exclusion relation[5] among the range signals, such that the environment would not be able to provide more than one range for each original valued object at the same instant. As for sensors, since every sensor is ever-present and always carries a value, a thread is composed in parallel to the program, emitting the range signal for its first range when its other range signals are absent.

The declarations of sensor `Speed` and valued output signal `MotorForce` in the example are replaced with the following declarations:

```
input    R1_Speed, R2_Speed, R3_Speed, R4_Speed,
         R5_Speed, R6_Speed, R7_Speed, R8_Speed,
         R9_Speed;
relation R1_Speed # R2_Speed # R3_Speed # R4_Speed #
         R5_Speed # R6_Speed # R7_Speed # R8_Speed #
         R9_Speed;
output   R1_MotorForce, R2_MotorForce, R3_MotorForce,
```

[5]An *exclusion relation*, also known as *incompatibility*, is a declaration that asserts that no two signals listed by the relation declaration can be present simultaneously in the environment [4].

```
        R4_MotorForce, R5_MotorForce, R6_MotorForce,
        R7_MotorForce, R8_MotorForce, R9_MotorForce;
```

To implement Speed's "ever-presence" property, we compose the following code segment in a parallel thread:

```
loop
   present not ( R2_Speed or R3_Speed or R4_Speed or
                 R5_Speed or R6_Speed or R7_Speed or
                 R8_Speed or R9_Speed ) then
      emit R1_Speed
   end present;
   await tick
end loop
```

The original variable assignment and valued signal emission statements are replaced by emit statements, emitting pure local signals to denote the execution of their respective statements. The effect of the assignments and the emit statements from the program is simulated by parallel components, calculating the range signal that should be emitted every instant for each numerical valued object of the program, to represent its state in every instant. For a variable, the simulation code tests whether a signal denoting an assignment to that variable has occurred in the previous instant, and if so – it emits the range signal representing that variable's state following the assignment. If no "assignment" is performed, then the previous range signal is emitted once again. For full details regarding variable simulation using pure signals, see Section III. For a valued signal, the simulation code works similarly, with two differences: the previous range signal emitted need not be emitted in an instant if no "emission" takes place in that instant, and every emission operation can be handled independently.

We need to translate the emission of MotorForce in the example to range signal terms. The original emit statement is replaced with emit E1. A new code segment is composed in a thread running in parallel to the current module's body, emitting the correct range signal for MotorForce in response to E1's presence. The local signal E1 must be declared as well, and its scope must include both the transformed module's original threads and the new thread shown below.

```
... || [ loop
            present E1 then
               present R1_Speed then
                  emit R9_MotorForce
               end present;
               present R2_Speed then
                  emit R8_MotorForce
               end present;
               ...
               present R9_Speed then
                  emit R1_MotorForce
               end present;
               await tick
         end loop ] || ...
```

Variable abstraction also employs pure signals and resembles valued signal abstraction; however, it is different because a variable can store values for future instants and can change value multiple times in every instant. It is elaborated on in Section III.

We now discuss transforming expressions. ESTEREL features three types of expressions [4]: *data expressions*, which are built by combining basic objects using operators and function calls; *signal expressions*, which are Boolean expressions over signal statuses; and *delay expressions*[6], which are used by temporal statements such as await[7] and abort[8].

Signal expressions consist of: signal identifiers, which may appear within pre operators (meaning their status from the previous instant); parentheses; and logical operators (and, or and not). They are tested by present statements (conditional statements testing signal statuses instead of data) or occur within delay expressions used by temporal statements. These expressions are transformed as follows:

- An identifier of a signal S having a numerical data type, whose ranges are $R_1^S, R_2^S, ..., R_n^S$, is replaced by the sub-expression (R1_*S* or R2_*S* or ... or R*n*_*S*).
- An identifier of a signal whose data type is boolean is replaced by the identifier of its respective presence signal.

In our example, there is a present statement testing whether MotorForce is present before testing its value. Accordingly, we replace the MotorForce identifier with a chain of or operators over the range signals representing MotorForce in the purified program: present (R1_MotorForce or R2_MotorForce or ... or R9_MotorForce) then...

Numerical data expressions whose values are assigned to variables or carried by signals are handled by the variable assignment and valued signal emission simulation mechanisms. Hence we would like to focus on Boolean data expressions, whose values are not only used in variable assignments and signal emissions, but also in if statements.

Boolean data expressions are recursively translated to signal expressions. They are tested at each instant in a loop composed in parallel to the program. If the expression is true, a special signal is broadcast and is used in the purified code:

1) To calculate the results of simulated assignments to boolean-typed variables and emission of boolean-typed signals; and
2) To simulate the evaluation of Boolean data expressions by if statements. An if statement testing a Boolean data expression B_i is replaced with a present statement that tests its corresponding signal B*i* in the transformed program.

In the example there are two Boolean data expressions. We translate the first, ?MotorForce > 270.0 or ?MotorForce < -270.0, into the signals expression R9_MotorForce or R1_MotorForce and create a thread that runs in parallel to the module's original threads. The new thread contains a loop in which the signal B1 is emitted in every instant in which this signal expression is satisfied. Afterwards, if ?MotorForce > 270.0 or

[6]A delay expression is an expression defining a delay that begins when the temporal statement bearing it starts and elapses in some later instant, possibly in the same instant in which the delay starts (delays that may elapse immediately are called *immediate delays*, and they start with the immediate keyword). There are three types of delays (standard, immediate, and count delays), but all delay expressions use signal expressions.

[7]The await statement pauses program execution until a delay elapses [4].

[8]The abort statement kills its body when a delay elapses [4].

`?MotorForce < -270.0 then...` can be replaced by the statement `present B1 then...`.

Additional transformations include removing the original interface and local declarations for the valued objects and modifying input relations. There are two types of input relations offered by ESTEREL: *exclusion relations* and *implication relations*. Exclusion relations, shown earlier as means to ensure that no two range signals representing a numerical input signal, assert that no two signals listed by the relation declaration can be present in the environment simultaneously. If the original program contains an exclusion relation where a numerical valued signal appears, than the declaration is expanded to include all range signals representing it. Identifiers of valued Boolean signals are replaced by the identifiers of their respective presence signals. Implication relations, which are relations of the form `relation` A `=>` B, asserting that if A is present then B must be present, are harder to transform, as they require a different solution for each case. For example:

- If A is a valued signal and its type is numerical while B is pure, then the relation is broken-down to a series of declarations for each range signal representing A, stating that an occurrence of any range signal of A implies that B is present.
- If A is pure and B is valued and numerical, then we add a thread containing a loop, where in every instant we check if A is present, and if so we ensure that if no other range signal representing B is present, then `R1_`B is internally-emitted. The reason that we cannot use implication in this case is that we want to allow any range signal representing B to appear if A is present, and not necessarily one particular range signal.

The full list appears in [1].

We want to verify the following properties in the example:

1) The program never simultaneously emits the signals `AC_ON` and `AC_OFF`. Such a situation "confuses" the external controller of the air conditioner. When an occurrence of both signals in the same instant happens, the observer emits `ErrorACConflict`.
2) The air conditioning is not off when the force exerted by the motor is greater than 270.0 in either direction. In case of violation, the observer emits the signal `ErrorNoAC`.

Hereby is the observer code – expressed in terms of the abstract program:

```
signal AC_isOn, AC_isOff in
   loop
      % no air condition conflict: ON and OFF not emitted
      % at the same instant
      present AC_ON and AC_OFF then
         emit ErrorACConflict
      end present;
      % air condition is never off when motor force is
      % greater than 270 or lower than -270
      present AC_isOff and
              (R1_MotorForce or R9_MotorForce)
        then emit ErrorNoAC
      end present;
      await tick
   end loop ||
   [ abort
         sustain AC_isOff
      when immediate AC_ON;
      loop
         present AC_OFF then
            emit AC_isOff
         else
            present AC_ON then
               emit AC_isOn
            else
               present pre(AC_isOff) then emit AC_isOff
               end present;
               present pre(AC_isOn) then emit AC_isOn
               end present
            end present
         end present;
         await tick
      end loop ]
end signal
```

The declaration of the output signals `ErrorACConflict` and `ErrorNoAC` joins the interface of the module. This observer code consists of one parallel thread in charge of emitting the observer signals in every instant in case of a property violation, and another parallel thread calculating auxiliary signals. The auxiliary signals indicate whether the air conditioning is on or off, depending on the most recent control signal to the air conditioning subsystem emitted by the program. Composing the observer code in parallel with the main program's body, and running XEVE on the resulting program, neither `ErrorACConflict` nor `ErrorNoAC` are ever emitted.

When applying our technique to abstract programs that satisfy the technique's requirements, verification using observers is both sound and complete [1]. We proved it by showing that an abstract program has the same reactive behavior as the concrete program, up to communicating with the environment by means of pure signals instead of the original valued signals and sensors. There is a loss of information and the abstract program allows theoretically more behaviors than the concrete program, in the sense that the abstract program provides ranges of possible values for the outputs given ranges of possible values for the inputs. However, the two programs make the same control decisions under the same circumstances and emit corresponding outputs, in particular with respect to pure signals, including, but not limited to, the observer signals, hence the soundness and completeness of the proof technique as a whole. The proof starts by defining for every event of the concrete program a corresponding event of the abstract system. The proof that the two systems behave in the same way given corresponding timed input sequences is statement-wise. On the one hand, statements that do not handle valued objects remain unchanged by the abstraction. On the other hand, statements that handle them are altered or replaced, hence the proof focuses on them. We identify four groups of statements that concern valued objects: statements that declare local signals and variables; statements that test data, i.e. make control decisions based on the data (namely `if` statements); statements that test signals, i.e. make control decisions based on statuses (presence or absence) of signals; and statements that manipulate data. We investigate, for every statement, the behavior of the construct replacing it in the abstract program. The full details appear in [1].

III. Variable Abstraction

As we do for valued signals, we replace variables with local pure signals representing their current values. Employing pure signals for representing variables in the abstract program has a major drawback. ESTEREL supports multiple assignments to a variable in a single instant, as it does not contradict the synchrony hypothesis [4]; however, unlike variables, a pure signal may have only one status at each instant – it may either be present or absent, not both.

In the current version of the algorithm, we impose two restrictions on programs supported by the abstraction in addition to those listed in Section IV: (i) each variable is assigned at most once on every instant – after it is read[9] by all statements referencing it in that instant; and (ii) initializing a variable takes an instant (as a result of the previous constraint). Given these assumptions, we have implemented variable simulation[10] in the abstract program as follows:

- A local pure signal is declared for each assignment statement in the concrete program: `A1`, `A2`, `A3`, etc.
- Every assignment statement is replaced by an `emit` statement emitting the corresponding pure signal.
- For every variable *v* a concurrent thread is added to the program in which there is a loop that emits in every instant the pure signal representing the current value of *v*.
- The code in this loop checks the previous status of signals indicating an assignment to *v*. If one of these signals has been emitted during the previous instant, it calculates the pure-signal representation of *v*'s value in the current instant based on the statuses of the pure signals representing the values of valued objects from the previous instant. If no signal indicating an assignment to *v* has been present in the previous instant, the pure signals representing *v*'s previous value are re-emitted in the current instant; since no assignment to *v* means that it retains its previous value.

This implementation with its implied restrictions is suitable for a certain group of programs, such as programs using a variable to manage a state machine: at the beginning of each reaction the current state is checked by reading the state variable, the reaction depends on the state and the input, and at the end of the reaction, if the program changes state or mode of operation, it assigns a new value to the state variable.

For example, in [1] we show a translation to ESTEREL of an *Escape* behavior for autonomous mobile robots based on a program taken from [11]. In this example, once a robot encounters an obstacle it backs off a predefined distance, rotates a predefined angle, and then returns to its previous occupation

[9]Reading a variable refers to an evaluation of a data expression containing an occurrence of that variable.

[10]By variable simulation we refer to the code that the abstraction adds to the program that simulates operations carried-out on a variable, i.e. assigning new values to it, in terms of the pure signals employed by the abstract program to represent the value of that variable.

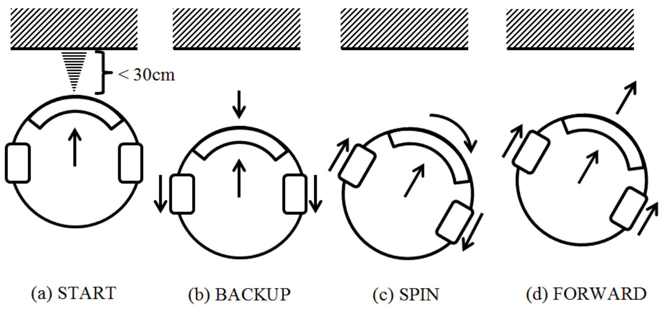

Figure 1: Robot Escape Behavior (with range detection).

(e.g. cruising). We add a range detector such that the robot can escape an obstacle based on proximity without actually having to collide with it. The program uses a state variable, and in every reaction responds according to the current sensor reading and the current state. A reaction ends with assigning a new value to the state variable when transitioning to a new state. Figure 1 illustrates the implementation.

To represent the behavior's state, we use an `integer` variable called `State`. The states are enumerated from 1 to 4. The partition of the integers domain to ranges for `State` is $\{(-\infty, 0], [1, 1], [2, 2], [3, 3], [4, 4], [5, +\infty)\}$ since `State` is assigned the values of 1, 2, 3 and 4, and is also tested to be equal to either one of them.

In the purification process we replace every statement assigning a value to `State` in the concrete program (see the source code in [1]) with a signal emission; e.g., the first assignment `State := 1` is replaced with `emit A1`; the second assignment `State := 2` is replaced with `emit A2`; and so forth. We compose in parallel to the original program a thread containing a loop that emits at every instant the range signal representing `State`'s current value:

```
|| loop
      % for each signal denoting an assignment to State
      % the next range signal for State is emitted
      present pre(A1) then
         emit R2_State
      else present (A2) then
         emit R3_State
      else ...
      else
         % if no assignment has occurred in the previous
         % instant then State's previous range signal
         % is emitted
         present pre (R1_State) then
            emit R1_State
         end present;
         present pre(R2_State) then
            emit R2_State
         end present;
         ...
```

The choice whether to postpone the effect of an assignment to the next reaction or to respond to it immediately affects only the constraint we have to impose, because either way our abstraction allows at most one assignment to a variable in every reaction. In the first option we allow one assignment at the end of the reaction after all statements use the value assigned to it in a previous instant, whereas the second option allows one assignment at the beginning of the reaction and using that value henceforth, until the start of the next reaction where the variable is assigned again with a new value.

The difference between variable assignment in the concrete program, which is done immediately, and the variable assignment simulation in the abstract program, where the assignment is postponed to the next instant, has no externally-visible implications for programs that satisfy the two constraint mentioned above, since we require the concrete program not to use the new value of an assigned variable until the next instant. However, this approach severely confines the use of variables. One approach that we considered for allowing free use of variables was adding a delay immediately following an assignment, in order to "buy time" for the variable to update, i.e. giving the simulation code time to detect the signal notifying on the change and respond by emitting the new pure signals representing the variable's new value. However, this transformation, which in practice breaks-down seemingly one reaction into a series of reactions, can change the timing and semantics of the program significantly. In particular, it violates the assumption that every signal retains the same status throughout the entire reaction. For example, have a look at the following program segment:

```
emit O;
present O then
   v := 1
end present;
present O then
   v : = 2
end present;
```

Since `O` is present during the instant in which this code segment is executed, the two signal tests are satisfied and `v` is assigned consecutively the values of 1 and 2. Therefore, in the end of this program segment, `v` has the value of 2. Suppose we add a delay after assignment to `v` to comply with the assumptions above, e.g. by waiting for an arrival of some signal `DELAY` after each assignment statement. We get the following code:

```
emit O;
present O then
   v := 1;
   await DELAY
end present;
present O then
   v : = 2;
   await DELAY
end present;
```

In this program segment `v` is assigned the value of 1 since `O` is present before the first arrival of the `DELAY` signal. If we assume that `O` is not emitted by any parallel thread, then `v` is not assigned the value of 2 since `O` is absent after the arrival of `DELAY`.

We suggest an improvement to our algorithm, a workaround that enables us to assume that every variable is assigned at most once in every instant. This requires implementing a pre-processing step, combined with an alternative implementation of variable simulation that responds immediately to signals notifying on assignments to the original variable.

The pre-processing step will transform the concrete program such that each variable is assigned at most once in every reaction. We call the target form SINGLE ASSIGNMENT PER REACTION (SAPR). It is inspired by *Static Single Assignment* (SSA) form. An SSA form is a representation of a program involving separating each variable *v* in the program into several variables v_i such that each variable is assigned only once [12]. In this work we refer to the variables v_i as *instances* of *v*. The SAPR form is distinguished from SSA form in the sense that every variable is assigned at most once in every instant but not necessarily once in the entire program code. An additional difference is that SSA form employs Φ-functions: in a node in the control flow graphs having incoming edges from two other nodes where there are two different valid instances of some variable (for example, such node could be a statement following an `if` statement having two branches – `then` and `else`, each of which manipulates a certain variable `L` from the original program code, such that new instances are created in each branch for `L` – `L1` and `L2` respectively), the SSA form defines a new variable (e.g. `L3`) and assigns to it a Φ-functions standing for the value of valid variable instance at the entrance to that node (i.e. `L3` $\leftarrow$ Φ(`L1`, `L2`)). In the SAPR transformation we need an equivalent of a Φ-function where the control converges back at terminations of branching statements and traps. We use special variable instances which are assigned the value of the valid variable instances at the ends of branches and before `exit` statements escaping traps.

For example, suppose that in the program investigated in Section II we wanted to limit the value of control signal to a maximum of 50 in either direction, and wanted to break-down the calculation of the control signal to several steps using variables, we could have used the following code:

```
var Error, CtrlSigVal : double in
   Error := 100.00 - ?Speed;
   CtrlSigVal := Error * 3.0;
   if CtrlSigVal > 50.0 then
      CtrlSigVal := 50.0
   end if;
   if CtrlSigVal < -50.0 then
      CtrlSigVal := -50.0
   end if;
   emit MotorForce(CtrlSigVal)
end var
```

This code segment employs two variables: `Error`, which contains the error, i.e. the difference between the set point (the target speed – 100) and the current speed; and `CtrlSigVal`, which contains the value of the control signal, obtained by multiplying the error (the value of `Error`) by the gain (3). If the product of the error and the gain is smaller than –50, then it is set to –50, and if it is larger than 50 it is set to 50. Below is the SAPR form, guaranteeing that in each instant each variable is assigned at most once:

```
var Error_0, Error_1,
    CtrlSigVal_0, CtrlSigVal_1,
    CtrlSigVal_2, CtrlSigVal_3,
    CtrlSigVal_4, CtrlSigVal_5 : double in
   Error_1 := 100.00 - ?Speed;
   CtrlSigVal_1 := Error_1 * 3.0;
   if CtrlSigVal_1 > 50.0 then
      [ CtrlSigVal_2 := 50.0 ] ;
      CtrlSigVal_3 := CtrlSigVal_2
   else
      CtrlSigVal_3 := CtrlSigVal_1
   end if;
   if CtrlSigVal_3 < -50.0 then
      [ CtrlSigVal_4 := -50.0 ] ;
      CtrlSigVal_5 := CtrlSigVal_4
    else
      CtrlSigVal_5 := CtrlSigVal_3
    end if;
    emit MotorCtrlSig(CtrlSigVal_5)
end var
```

In the transformed code segment we create a variable instance for every assignment statement. Once assigned, a variable instance substitutes the variable from the original program wherever it appears in data expressions from that point on, until the next assignment to the original variable, where another variable instance is used. `if` statements, which have no `else`-branch in the original program, are added with an `else`-branch. A new variable instance is created to hold the valid value upon leaving the `if` statement, and it is assigned at the end of each branch. This exemplifies the reason why `else`- and `then`-branches are introduced when missing – if a variable is manipulated in one branch and not in another then still at the end of the `if` statement there is one variable instance replacing the original variable and holding the right value. In [1] we elaborate on the transformation and provide the full example, including simulation results.

IV. Discussion

Our method supports only programs that compile and execute without errors and that comply with several assumptions and constraints, which are described in this section. These constraints are required for maintaining relationships between valued objects and for replacing numerical and Boolean calculations with pure signal operations. However, our method still is useful for a large set of control and robotic programs. Various robot behaviors appearing in [11] can be implemented in ESTEREL, processing pure signals or requiring sufficiently simple calculations, thus our method can be applied to them.

In [1] we list assumptions about the input program that exist mainly for the simplicity of the solution. These assumptions usually concern ESTEREL syntactic sugaring instructions. However, they do not reduce the expressive power of ESTEREL, as each assumption can be attained by replacing the original construct by a semantically-equivalent construct, either manually or automatically. For example, we require that the program will consist of only one module. When the program consists of several modules, one module calls another using the `run` statement. The `run` statement instantiates one module within another module by syntactically replacing the `run` statement with the body of the instantiated sub-module, exporting the data declarations of the instantiated sub-module to the parent module, and discarding the interface declaration of the instantiated sub-module [4]. By replacing any `run` statement in a program having multiple modules with the instantiated sub-module's code, we can comply with this assumption.

Unfortunately, we have some real constraints that limit the family of programs to which our abstraction is applicable.

a) We do not support external code, i.e., code written in the host language: The ESTEREL v5 compiler translates the program into a program or circuit written in a host language chosen by the user, for example C. The programmer can declare abstractly and use various function, procedures, tasks, constants and data types to be implemented in the host language and linked to the code generated by the ESTEREL compiler [4]. Being outside the scope of the ESTEREL program, its behavior cannot be taken into consideration, as our method verifies the ESTEREL code.

b) We do not support cyclic dependencies between values of numerical valued objects: In order to maintain relationships between targets and sources of assignments (as well as valued signal emissions) of numerical valued objects, we restrict the numerical data expressions used to affine transformations of values of numerical valued objects (see constraint 3 in the list below in this section). We refine a valued object x's partition of $(-\infty, +\infty)$ to ranges for every assignment of x's value to a valued object y, according to the data expression, defining an equation with x and y which is satisfied once that assignment is executed.

Since the order of partitioning is derived from the dependencies between valued objects, if there exist cyclic dependencies[11] between valued objects then the order of partitioning cannot be determined, since no strict ordering of the partitions calculated can reflect the dependencies between the objects.

Even when an object's value is transformed and set to itself directly (e.g. `v:=3*v+1` for a variable v), the partitioning process should theoretically refine the partition of that object's domain to ranges over and over again, infinitely many times. One solution we considered was to allow self-assignments, in the following form: Let $P = \{R_1, R_2, ..., R_n\}$ be a partition of $(-\infty, +\infty)$ for some valued object x. Suppose we do not take self-assignments into consideration when partitioning, but rather leave the current partition, and define assignment as "transitioning" x from its previous value's bounding range(s) to a union of ranges bounding x's value following the assignment. If x occurs in a formula F testing x's value, and an assignment causes the bound of x's value to contain both ranges satisfying F and ranges not satisfying F, we cannot decide if F is satisfied. For example, suppose we compute for an integer variable v the ranges [1, 8], [9, 9] and [10, 12] (among others) and we have a conditional testing the expression $v = 9$. Consider the assignment $v := v + 1$. If v is in [1, 8], and the assignment is executed, then in the next instant it can be in either [1, 8] or [9, 9], thus we cannot decide if the condition $v = 9$ is satisfied.

To determine the order by which the partitioning process inspects valued objects when partitioning by dependency, a directed graph of valued object dependencies $G = \langle V, E \rangle$ is created in the following manner. Our inputs include the set VALOBJ of valued objects in the source program, and the set of all variable assignments and valued signal emissions

[11]The term *cyclic dependency* in ESTEREL usually refers to a situation where there exists an instantaneous circular dependency between a signal and itself [4] (also known as a *causality cycle*). ESTEREL programs that contain an instantaneous dependency cycle, for which the number of solutions is not exactly one, are considered invalid. In this section we discuss a different kind of dependency, however: a dependency between the current value of an object and its previous one, from which it is calculated. We use the terms *causality cycle* and *causality problem* to refer explicitly to a causality cycle.

which appear in the original program. V and E are defined as follows. For each numerical valued object define a vertex: $V = \{x \mid x \in$ VALOBJ and x is of type `integer`, `float`, or `double`$\}$. The set E of edges in G is calculated using the following procedure.

```
1.    $E \leftarrow \emptyset$
2.    For each two numerical valued objects $v$ and $u$ in $V$:
      2.1.  If the current or previous value of a numerical valued object $u$ is assigned to $v$ (i.e., if $v$ is a variable and there exists an assignment $v$ := $exp$ or if $v$ is a signal and there exists an emission emit $v(exp)$, where $exp$ is a data expression with an occurrence of $u$), stretch a directed edge from $u$ to $v$; that is, add $(u, v)$ to $E$.
      2.2.  If $G$ contains cycles, halt the process with an error message.
```

The order by which the process partitions the domains of valued objects is a post-order of this graph. That is, a valued object v's partition is computed after the partition is computed for all numerical objects to which v is assigned.

Hereby is a summary of the constraints on the programs to which our method is applicable. These compromises are basically due to two limitations:

1) One way to allow automatic methods to check a non-trivial property is to reduce the power of the language or, equivalently, reduce the class of program verifiable using the method; and
2) Technical issues with aspects of our abstraction process conflicting with the synchrony hypothesis.

The constraints are:

1) All valued objects are of type `boolean`, `integer`, or floating-point real (i.e. strings and user-defined types are not allowed);
2) All numerical formulas used in Boolean data expressions are of the form $(a{\star}D{+}b)\ R\ k$ where R is an relational operator, D is the current or previous value of a numerical valued object (sensor, valued signal or variable of type `integer`, `float` or `double`) and a, b and k are some literal constants;[12]
3) All numerical data expressions used in assignments and valued signal emissions are of the form $a{\star}x{+}b$, where x is an occurrence of a valued object (in its current or previous value), and a and b are constant literals, all of which of the same domain;
4) All delayed expressions in temporal statements may not be count delays;
5) No repeat loops;[13]
6) No combined signals and no valued traps;
7) No cyclic dependencies between numerical valued objects or self-assignments (except in loop-free program segments); and
8) During one instant, the program does not access more than one incarnation of a local valued signal or a variable.

Currently the algorithm and tool support accessing the values of valued signals only during reactions in which they are present; however, this can be handled by adding signals to represent the latest value of a signal from the last time it occurs. This solution will be implemented in future versions.

The set of programs to which our method is not applicable includes, but not limited to, programs employing counters (due to a dependency of the counter by its previous value); programs performing calculations which are more complicated than allowed by constraints 2 and 3 (e.g., containing operations such as dividing by the value of some object or comparing one numerical valued object to another); and programs requiring extension in the host language. By design, ESTEREL's data definition facilities are minimal, since data-handling is not the primary concern in control-dominated reactive programming [4]. A major advantage of this approach is high portability of the code. To program complex systems, the programmer has to put a significant effort in implementing data handling capabilities in the host language. Without supporting host language extensions, our technique is limited to simpler, control-centric programs.

A *Proportional-Integral-Derivative (PID) controller* [11] is an example of a controller to which our method is not applicable. This controller has three terms: proportional (proportional to the error signal), derivative (proportional to the derivative of the control signal, i.e. the rate of change in the error signal over time), and integral (proportional to the integral of the error signal, i.e. the summation of error over time). To calculate the integral term, the system needs to sum of the error over time, essentially creating a cyclic dependency. However, in the high level at which robot behaviors are programmed in behavior-based robotics, proportional controllers, which our method supports, are nearly always sufficient [11]. Moreover, PID controllers are not the usual application of ESTEREL. ESTEREL, as a *state-based formalism*[14], is better suited for problems where control flow is prevalent, e.g. systems that jump between different functioning modes [13]. Another style of synchronous programming, where the system's behavior is represented as a set of recurrent equations, characterizing languages such as SIGNAL and LUSTRE, is well-adapted to problems where data-flow is prevalent [13], hence being more natural for implementing PID controllers where the transfer function is repeatedly calculated.

We currently investigate means to lift some of these constraints. Some research directions appear in Section VI.

[12]Currently we support only linear equations and inequalities in one variable, for future work see Section VI.

[13]ESTEREL v5 offers several loop constructs: `loop`, `loop-each` and `every` (temporal loops) and `repeat`. A `repeat` loop executes a finite number of times, unlike the others, which can loop forever [4]. We do not support only the `repeat` statement.

[14]A state-based formalism uses a state transition diagram where arrows are labeled with communication actions to represent the system's behavior. The diagram can be explicit in visual formalisms such as in STATECHARTS or implicit in imperative formalisms such as ESTEREL and CSML [13].

State-Space Complexity Evaluation

The complexity of the number of pure signals that the abstraction adds to the program is $O\left(N_{Bool} + N_{num} \cdot (B_{atomic} + A_{const}) + A + E + B\right)$ when there is at most one relation between every two valued objects; N_{Bool} is the number of Boolean valued objects, N_{num} is the number of numerical valued objects, B_{atomic} is the number of atomic Boolean data expressions having occurrences of numerical valued objects, A_{const} is the number of assignments of constant values to numerical valued objects, and *A*, *E*, and *B* are the numbers of variable assignments, valued signal emissions, and Boolean data expressions respectively. For more complicated situations in which there can be two or more relations between two numerical valued objects we provide a recurrence relation in [1].

In general, an automaton generated from an ESTEREL program may suffer from size explosion [14]. Increasing the number of input and output signals may significantly enlarge the sets of states and transitions of the automaton produced from the purified program compared to the one generated from the original program. In two of the examples presented in [1, Chapter 5] the number of states remains the same following the purification and for one example the number of states increases from 6 to 16. However, one must remember that the abstraction is done for verification purposes only. Moreover, a typical ESTEREL application yields a fairly small number of states, usually between 10 and 100 [14].

V. CASE STUDIES

The constraints described in the previous section provide an accurate characterization of the family of programs to which the method can be applied. The example in Section II demonstrates the applicability of the method to two classes of control systems within this set: bang-bang controllers and proportional controllers. Another example demonstrating the application of the method to a bang-bang controller is a temperature controller system maintaining a constant temperature in a chamber using a thermometer, a timer, and two boilers [1]. We present there also the application of our method various robot programs:

- Border-following robots using one or two light sensors.
- Bang-bang implementation of the *Home* robot behavior: a robot homes on a destination marked by a beacon using differential sensing. It is based on a program appearing in [11]. We use our technique to verify that the robot does not run over the beacon, assuming that when the robot is too close then the light intensity picked by the sensor is greater than a given threshold.
- *Escape* behavior: once colliding with an obstacle, a robot goes back a predefined distance, rotates a predefined angle and then continues moving forward. It is also based on a program that is described in [11] (though modified to use a range detector and escape once detecting a close obstacle). We verify that the robot never reaches a non-reactive state, in which it fails to find a reaction and emits an error signal.

The last two examples (Home and Escape behaviors) demonstrate the application of the method to mobile robot programming following the *reactive control paradigm*. This paradigm, based on animal models of intelligence, decomposes the overall action of the robot by behavior, allowing handling multiple goals and multiple sensors, increasing robustness and extensibility [15]. By combining various behaviors and control algorithms, complicated control systems to which our method is applicable can be derived. The full code for the examples, including original program code, abstract program code, and observer code are provided in [1].

The abstraction we propose has an advantage over complete data abstraction performed when providing the `-soft` flag to the ESTEREL compiler, since it avoids adding behaviors not displayed by the original program. For example, consider the following code portion switching on and off an actuator based on a numerical input through a sensor `S`:

```
if ?S < 90.0 then emit On end if;
if ?S > 110.0 then emit Off end if
```

The ESTEREL compiler generates with the `-soft` flag a circuit in which both conditions can be satisfied at the same time, therefore it can emit both `On` and `Off` at the same reaction. However, using our abstraction, in the purified program no two range signals representing `S` can occur simultaneously, therefore both conditions cannot be satisfied at the same time.

VI. CONCLUSION

Combining the verification power of XEVE with the transformation of non-Pure ESTEREL programs into Pure ESTEREL programs, we can verify safety properties of a larger family of programs. We provided examples for various categories of programs that can be verified.

There still is a large set of programs to which we cannot apply our method, e.g. those involving complicated calculations and counters. By giving up completeness, the full potential of the technique developed is realized. Applying the technique to parts of the program that fulfill the constraints while letting the ESTEREL compiler remove the rest of the data when compiling the program into a circuit can produce a more precise over-approximation than total control-based abstraction. This is especially useful when the system consists of several sub-systems, some of which fulfilling the constraints while others not. In [1] we provide an example of a system comprised of a PID controller and a limit switch. The limit switch shuts the process down by cutting off the PID controller's output once an undesired limit is reached. After the measured value drops back to the safe zone, the switch can be manually reset in order to reactivate the control system. An observer helps to verify safety properties of the system by emitting a special signal whenever the program emits the output signal while the measured value is higher than some critical threshold. Not only that the calculations performed by the PID controller are not supported by our technique, but also the program takes the set point, clock interval, and gains from constants defined in the host language. The abstraction performed by the ESTEREL compiler alone produces a circuit which XEVE reports to possibly emit the observer signal. However, using our technique to abstract the safety limit switch and the observer, which comply with the requirements of our techniques, and letting the compiler abstract the rest of the program creates a circuit which never emits the observer signal, as XEVE guarantees.

The current version of the algorithm supports only linear equations and inequalities in formulas occurring in Boolean data expressions. We plan to extend the class of supported formulas, for example to handle also univariate polynomial equations and inequalities whose the roots can found analytically, i.e. polynomials of second, third, and forth degrees; the roots can be used to partition the domain of the valued object (which serves as the variable of the formula) to intervals.

One alternative approach to perform the analysis and the abstraction is based on the constructive semantics of ESTEREL [16]. ESTEREL statements are divided into two groups: *kernel statements*, forming a primitive core of the language, and *derived statements*, which are definable as combinations of kernel statements, whose purpose is to make programming more convenient. The semantics of an ESTEREL program is obtained by structural induction on the statements which it consists of. Considering the constructive semantics of ESTEREL, it is essentially enough to define the abstraction process and prove its correctness for kernel statements to obtain a process and a proof that applies to the entire ESTEREL syntax. Yet it would require to implement a much more complicated abstraction technique, that not only finds and transforms items of interest, but also breaks-down the program to kernel statements. Our approach focuses on the occurrences of valued objects in the program, leaving the rest of the program untouched.

The TEMPEST [17] toolset provides a compiler for linear temporal logic formulas representing safety properties to observers in ESTEREL language. The automaton compiled from the observer-augmented program can be verified using other tools from the TEMPEST package. When the control flow of the program to be verified is independent of valued objects (i.e., there are no conditionals testing run-time values of valued objects), the control structure can be fully determined at compile-time, therefore the verification is both sound and complete. However, when the program's control structure depends on run-time values, the technique is sound but not complete, since the ESTEREL compiler considers all paths, including those that are unreachable due to data values [18, Section 4].

We plan to incorporate the power of TEMPEST with our method to extend both the class of verifiable properties and the class of programs to which those techniques can be applied. TEMPEST should be able to successfully verify programs where the control structure is independent on the run-time values of valued objects; therefore, TEMPEST should be directly applied to programs not complying with the constraints which our method requires, while actually not querying values of objects in conditionals. Hybrid versions may be suggested, such as purifying only valued objects on which the control-flow depends directly or indirectly, while leaving the rest of the valued objects untouched (as long as the valued objects on which the control flow depends satisfy our constraints). This, however, cannot be verified using XEVE. An additional possible research direction is extending the language used in TEMPEST to express safety properties querying the run-time values of valued objects, such as "variable V can never be greater than 20". In certain cases, this might be obtained by purifying together the parallel composition of the main program and the observer component.

REFERENCES

[1] N. Koblenc, "Purification of Esterel Programs," Master's thesis, Dept. Mathematics and Computer Science, Open Univ. of Israel, Ra'anana, Israel, June 2015, the thesis and prototype tool are available at http://www.cs.tau.ac.il/~tyshbe/NIR/nirThesis.html.

[2] N. Koblenc and S. Tyszberowicz, "Purification of Esterel Programs," in *Preproceedings of the Brazilian Symp. on Formal Methods (SBMF)*, C. Braga and N. Martì-Oliet, Eds., 2014, pp. 183–188, available at: http://www2.ic.uff.br/~cbraga/sbmf14/sbmf14-preproceedings.pdf (visited September 2015).

[3] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Stankovic, J. A. (Consulting Editor), Ed. Kluwer, 1993. [Online]. Available: http://dx.doi.org//10.1007/978-1-4757-2231-4

[4] G. Berry, "The Esterel v5 Language Primer, Version v5_91," Centre de Mathématiques Appliquées – Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000.

[5] L. J. Jagadeesan, C. Puchol, and J. E. von Olnhausen, "A formal approach to reactive systems software: a telecommunications application in Esterel," *Formal Methods in System Design*, vol. 8, no. 2, pp. 123–151, 1996. [Online]. Available: http://dx.doi.org/10.1007/BF00122418

[6] I. Sommerville, *Software Engineering*, 8th ed. Addison-Wesley, 2007.

[7] A. Bouali, "XEVE, an Esterel verification environment," in *Computer Aided Verification (CAV)*, ser. LNCS, A. J. Hu and M. Y. Vardi, Eds., vol. 1427. Springer-Verlag, 1998, pp. 500–504. [Online]. Available: http://dx.doi.org/10.1007/BFb0028770

[8] "The Esterel v7 Reference Manual, Version v7_30 – initial IEEE standardization proposal," Esterel Technologies, 2005.

[9] G. Berry and the Esterel Team, *The Esterel v5_91 System Manual*, Centre de Mathématiques Appliquées – Ecole des Mines de Paris / INRIA, Sophia-Antipolis, 2000.

[10] S. Das, D. L. Dill, and S. Park, "Experience with predicate abstraction," in *Computer Aided Verification (CAV)*, ser. LNCS, N. Halbwachs and D. Peled, Eds., vol. 1633. Springer, 1999, pp. 160–171. [Online]. Available: http://dx.doi.org/10.1007/3-540-48683-6_16

[11] J. L. Jones and D. Roth, *Robot Programming: A Practical Guide to Behavior-Based Robotics*. McGraw-Hill, 2003.

[12] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in progress," in *Principles of Programming Languages (POPL 1988)*. ACM, 1988, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1145/73560.73561

[13] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270–1282, September 1991. [Online]. Available: http://dx.doi.org/10.1109/5.97297

[14] G. Berry and G. Gonthier, "The Esterel synchronous programming language: design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992. [Online]. Available: http://dx.doi.org/10.1016/0167-6423(92)90005-V

[15] G. Dudek and M. Jenkin, *Computational Principles of Mobile Robotics*. Cambridge University Press, 2000.

[16] G. Berry, *The Constructive Semantics of Pure Esterel*, 2002, draft book, version 3, available at: http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf (visited September 2015).

[17] C. Puchol, *The TempEst Program Verification Toolset*, AT&T Bell Laboratories and the University of Texas at Austin, product documentation.

[18] L. J. Jagadeesan, C. Puchol, and J. E. Von Olnhausen, "Safety property verification of Esterel programs and applications to telecommunications software," in *Computer Aided Verification (CAV)*, ser. LNCS, P. Wolper, Ed., vol. 939. Springer-Verlag, 1995, pp. 127–140. [Online]. Available: http://dx.doi.org/10.1007/3-540-60045-0_45