From Data to Events: Checking Properties on the Control of a System

Christophe Jacquet Frédéric Boulanger Dominique Marcadet

SUPELEC
3 rue Joliot-Curie
91192 Gif-sur-Yvette Cedex, France

Preprint of the article: *From Data to Events: Checking Properties on the Control of a System.* Memocode 2008, the sixth ACM/IEEE International Conference on Formal Methods and Models for Co-Design, Jun 2008, Anaheim, California. pp. 17-26. Published by the IEEE.

This work has been performed in the context of the Usine Logicielle project of the System@tic Paris-Région Competitiveness Cluster (`http://www.systematic-paris-region.org`).

Abstract

We present a component-based description language for heterogeneous systems composed of several data flow processing components and a unique event-based controller. Descriptions are used both for generating and deploying implementation code and for checking safety properties on the systems. The only constraint is to specify the controller in a synchronous reactive language. We propose an analysis tool which transforms temporal logic properties of the system as a whole into properties on the events of the controller, and hence into synchronous reactive observers. If checks succeed, the final system is therefore correct by construction. When properties cannot be translated exactly into observers of the control, our tool is capable of generating approximate observers. In this case, the results are subject to interpretation, but can prove useful and help detect defects or even guarantee the correctness of a system.

1 Introduction

We describe here a method for the design and code generation of heterogeneous software systems, which allows the specification of *safety properties* on the system and their formal verification. We distinguish two aspects in the design of a heterogeneous system: the data processing operations performed to produce the outputs from the inputs, and the control of the system, which determines the schedule and parameters of the operations. Data processing can be described by data-flow models, whereas control is described by state machines or synchronous languages. Designing control independently from data processing results in a separation of concerns, which allows the formal verification, re-use and independent modification of the parts of the system.

We propose a modular approach, in which data processing is modeled by *processing components* which communicate through data flows, and control is modeled by a unique *control component* whose inputs and outputs are pure events. In this context, an application is described by a network of interconnected components. We do not provide automated means of *generating* a controller, but rather tools that allow a system designer to *check* the behavior of the controller he/she has written.

For system designers, meaningful properties are *global* properties, that apply to the inputs and outputs of the application itself. We consider only canonical safety properties, a subclass of Linear Temporal Logic (LTL) formulae, and we propose an automatic method which translates these properties into *local* properties expressed on the control component. The latter properties can be automatically transformed into observers written in the same formalism as used for describing the control component (namely a synchronous reactive language). In this way, formal methods can be used to check the control component directly: what is proved is what gets executed.

The paper is organized as follows. Section 2 provides a short review of related work, and justifies the choice of a purely event-driven control. Section 3 introduces ADLV, our Architecture Description Language for Verification, which is used for both the description of the application and of the safety properties. Then, we explain how safety properties can be translated, first into *intermediate formulae* (section 4), and finally into *observers* on the control component (section 5). Section 6 gives conclusions and perspectives for future work.

2 Related Work and Objectives

Embedded systems are almost always heterogeneous systems. Indeed, they often mix traditional continuous automatic control or signal processing subsystems with digital supervision that switches the system among several different states.

Some researchers have provided tools to analyze the behavior of so-called hybrid systems, physical systems that mix continuous behaviors and discrete transitions at some points [20, 27]. Tools also exist to synthesize *hybrid controllers* [6]. In this article, we do not study analysis of physical systems nor synthesis of hybrid controllers. Rather, we propose methods for *proving properties* on a *wholly-specified* heterogeneous system.

In our view, a system is comprised of several processing components (e.g. a block for computing FFTs, an automatic control loop, etc.), and a unique controller. The languages most commonly used in the industry generally allow one to design a controller as a discrete transition system, generally a state machine. This is the case of Esterel [4], Lustre [13], Signal [22], StateCharts/SyncCharts [1], Grafcet/Sequential Function Chart [9]. These languages are event-based, but they generally allow operations on numerical values as source of events. For instance, if a is a numerical input, then one can specify the event $a < 23$. This convenience has two drawbacks. First, it mixes "pure" control and data operations inside the controller. Indeed, whereas complex operations such as FFTs are naturally thought of as separate blocks, simpler operations are more likely included in the controller. Second, these languages have formal semantics which allow one to prove properties of programs by model checking [17, 25]. However, model checkers have limitations when dealing with float or even integer valued signals. One possible approach allows infinite-state transition systems, and performs abstract interpretation before doing model-checking [8]. However, we think that *explicitly* specifying the transformations on data-flow signals, through the definition of processing components, will help designers achieve a clean design, while avoiding issues.

Therefore, our framework clearly distinguishes two parts in a system: (1) an *operative subsystem*, consisting of components that communicate through data flow and possibly events, (2) control components whose inputs and outputs are only events. This allows formal verification tools to check properties on the control component. However, if these properties were to be specified on controller events, the verification framework would be cumbersome to use. Indeed, the primary vocabulary of the designer consists of the application inputs and outputs, and possibly some internal signals, and cannot be restricted to controller events.

The approach presented here is original in that it permits the specification of properties on the system using a vocabulary that is close to the designer's concerns, for instance *"component C is never active when input E is greater than value V"*. These global properties are then translated into the event-based semantics used by the control component at a local level, for instance *"event Y is never emitted between an occurrence of X and an occurrence of Z"*, using the description which is also used for generating the implementation. This guarantees the consistency of the transformation applied to the safety property and of the way implementation code is generated: this code is *correct by construction*. The verification method thus follows the WYPIWYE (*what you prove is what you execute*) principle [4].

We do *not* intend to produce a new formal verification tool, but to design an *analysis tool*, capable of transforming *global* properties expressed on the system as a whole into *local* properties expressed on the control component. The latter can easily be translated into *observers* [15] written in the target language. Observers can then be processed by the verification tools available on the market. In this way, the actual implementation code is checked, so the WYPIWYE paradigm is applied a second time.

Generators have been proposed to translate temporal logic properties expressed on signal values into run-time monitors [23, 2]. Our framework has the same expressive power as these, in that it allows one to reference and compare signal values. However, whereas these monitors only ensure that properties are verified on a *limited number* of *finite* traces, model checking allows us to *prove* the correctness of a system on *all* possible behaviors, including finite and infinite ones.

3 Architecture Description Language for Verification

3.1 Description of Embedded Applications

We introduce here a method for describing embedded applications as a set of processing components that are activated and connected at the request of a *control component*. This description is used both to generate the actual implementation of the application and to transform properties expressed on the application into properties that can be checked on the control component. We have designed the ADLV language (*Architecture Description Language for Verification*) which has both an abstract syntax and a textual concrete syntax which extends OMG IDL3. ADLV focuses on the description of component interfaces and connections between components; it does not intend to be as general-purpose as AADL [11]. This section is an informal introduction to ADLV through the example of a cruise control system depicted on figure 1. In this diagram, the following symbols are used for input and output ports:

🟥	event sink	◗	data flow input
🟩	event source	🔵	data flow output

An event is a *message* that bears no value, and that is either present or absent at any point in time. A data flow has a numeric or boolean value at each instant. The semantics of the system is given by the Kahn process network model of computation [19].

Processing components can either be "black boxes" described in specific formalisms such as Simulink (e.g. the *Regulator* component) or "internal" components directly described in ADLV (e.g. the 3 leftmost components). In the former case, only the *interface* of the component is described in ADLV, while internal components are completely described in ADLV, and their behavior is known to the ADLV analysis and code generation tools. Internal components serve as adaptors between components of heterogeneous nature: for instance, they can have dataflow inputs, and generate events based on a condition given by the designer. For example, the internal component *BrakesCheck* has a boolean data-flow *brakes* input, and produces an event each time the value of *brakes* changes. It produces *brakes_pushed* when *brakes* changes from *false* to *true*, and *brakes_released* for the other transition. This leads to the following definition for *BrakesCheck* in ADLV concrete syntax:

```
internal component BrakesCheck {
    sink BoolFlow brakes_on;

    publishes PureEvent brakes_pushed {
        when   brakes_on;
    }
    publishes PureEvent brakes_released {
        when ! brakes_on;
```

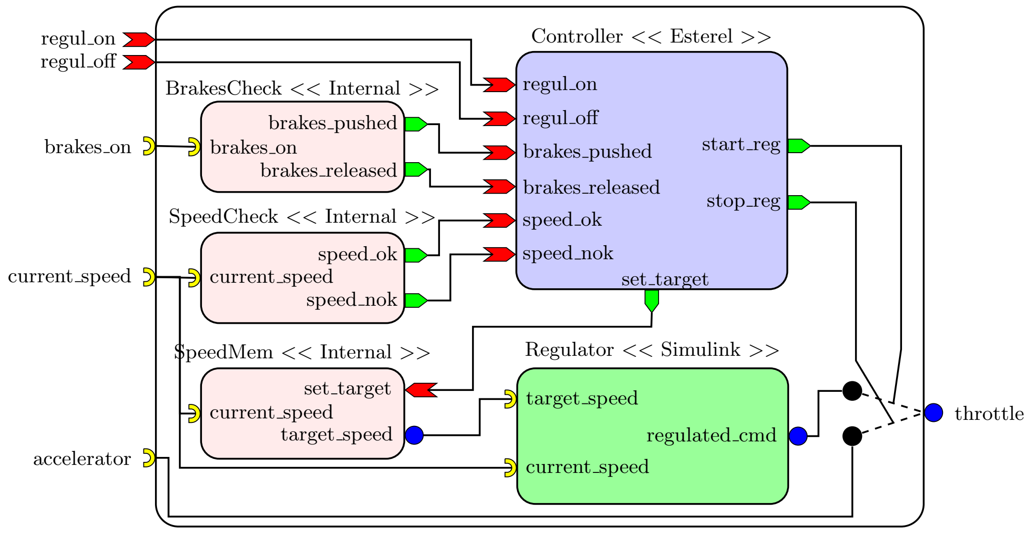

Figure 1: Component-based cruise control described in ADLV.

```
    }
};
```

Likewise, *SpeedCheck* emits a *speed_ok* event when *current_speed* enters the range of admissible values for regulation (40..130 km/h), and emits a *speed_nok* event when it exits this range. Hence the following definition:

```
internal component SpeedCheck {
    sink FloatFlow current_speed;

    publishes PureEvent speed_ok {
        when current_speed >= 40
          && current_speed <= 130;
    }
    publishes PureEvent speed_nok {
        when current_speed < 40
          || current_speed > 130;
    }
};
```

There is only one control component per application (here, the *Controller* component, which is written is Esterel); it is considered as black box that consumes and produces events. Connections between components can either be static or dynamic. In the latter case, an initial configuration is given, that can change over time.

The behavior of the cruise control is as follows: the Simulink component calculates a "regulated command" for the throttle. At startup, the accelerator pedal is connected to the throttle. If the current speed is in the range of admissible values, and the driver presses a "regulator on" button, the output of the *Regulator* component is connected to the throttle. As soon as the driver brakes, the throttle is connected back to the accelerator. To re-enable the cruise control, the driver must both stop braking and press the "regulator on" button. There is therefore a dynamic connection between the throttle and either the "regulated command" from the *Regulator* component or the *accelerator* input. It is depicted as a "switch" on figure 1.

We have implemented a tool that generates the actual implementation of the system from an ADLV description [5]. We will not describe this here, but rather focus on the checking of safety formulae.

3.2 Safety Formulae

3.2.1 Canonical Safety Formulae

Temporal logic is often used to express properties of reactive systems. In particular, Linear Temporal Logic (LTL) is widespread and well understood, especially for the verification of programs [10].

It is often critical to check that some property is "always true" or "never true". In LTL, for some property f, "f is always true" is denoted by $\Box f$, "f is never true" is denoted by $\Box\neg f$. If we restrict f to the class of *past formulae*, $\Box f$ is a *canonical safety formula* [7]. Such formulae offer good expressive power [21], are simple to use [16, 18] and easy to translate into synchronous reactive languages [3] such as Esterel or Lustre.

In our framework, past formulae are built from classical propositional operators ($\vee$, $\wedge$, $\rightarrow$, $\neg$), past temporal operators ($\mathcal{S}$ [since], $\mathcal{B}$ [back to], $\boxdot$ [always], $\Diamond$ [once], $\odot$ [previous]), and predicates. The predicates that system designers can use are given in table 1.

Type	True when...
s	event s is present
s `op` k	data flow s satisfies a comparison to constant k. `op` is a comparison operator among $\{<, \leq, =, \geq, >\}$
(`in`)`active`(c)	component c is (in)active
$c_i.p_j \ll c_k.p_\ell$	port p_ℓ of component c_k is connected to port p_j of component c_i
`true`, `false`	*boolean literals*

Table 1: List of predicates for safety formulae.

All the predicates can be *negated*, with natural meaning. For instance, $\neg(s < k) = s \geq k$, $\neg$(`active`(c)) = `inactive`(c), etc. These transformations are *purely syntactic rewrite rules*; there is *no* meaning associated with comparison operators, and signals or values are merely uninterpreted character strings. The signals used in the predicates can belong to the interface of the application, or be internal signals. Safety formulae are part of the ADLV descriptions, alongside the description of processing components and connections.

3.3 Non-Canonical Safety Formulae

Let us consider the following property that must always be satisfied: *when the driver brakes, the regulator is not connected until the driver presses the "regulator on" button and he/she releases the brakes.*

This property uses a *future* operator: *until.* Therefore, it is not a canonical safety property. However, it can easily be rewritten in the past tense: *since the brakes were pushed, if the driver has not both pressed the "on" button and released the brakes, then the regulator is not connected.* Hence the following ADLV statement (`a << b` means "output port b is connected to input port a"):

```
always {
  (!(regul_on && !brakes_on)) since brakes_on =>
      !(throttle << regulator.regulated_cmd);
}  /* Statement S1 */
```

At the moment, the designer has to rewrite safety formulae that are not in a canonical form. In section 6 we will discuss possible improvements to handle non-canonical safety formulae.

3.4 Purely Propositional Example

A statement can possibly use propositional operators only:

```
never {
  set_target &&
    (current_speed < 40 || current_speed > 140);
}  /* Statement S2 */
```

The statements S_1 and S_2 will be used as examples in the remainder of this paper. *never* and *always* statements being equivalent, we will only consider *never statements* from this point on, without loss of generality.

3.5 Checking the Controller against Safety Formulae

For each safety formula, we must generate an equivalent observer in order to use it to check the controller. However the safety formula can involve signals that are not directly connected to the controller. By analyzing the structure of the application and by looking into the internal blocks, we are able to build an equivalent temporal formula that only references some of the *controller* events. This method is described in section 4.

We are then able (see section 5) to translate this formula (called an *intermediate formula*) into one or two *observers*. These are modules written in the same language as the controller (for instance, Esterel or Lustre). One can then use language-specific *checking tools* in order to *prove* that the safety properties are satisfied by the controller.

The steps towards the generation of the observer are summarized on figure 2.

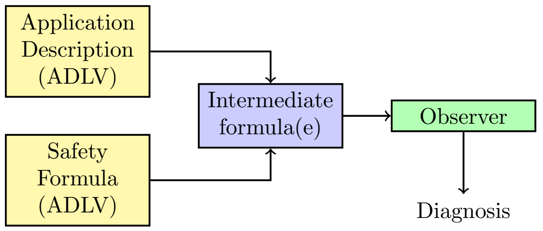

Figure 2: Overview of property verification.

These properties are checked on the synchronous implementation of the controller, which is destined to drive the application at run time, thanks to observers directly generated from the application description itself. As stated above, the WYPIWYE principle is effectively applied in these two respects.

4 Interpretation of Temporal Formulae

4.1 Overview

For each formula in a *never* statement, the general idea is to compute a signal in the synchronous language, which is emitted at each instant when the formula is satisfied. A special signal *failure* is emitted when the top-level formula of a never statement is satisfied. Model-checking tools will either prove that *failure* can never be emitted, or exhibit a counterexample.

The problem is stated as follows: *Given f a temporal logic formula involving application signals, build a corresponding signal s, based uniquely on the controller's input and output events, using constructs of the controller implementation language.* We can decompose this problem into two sub-problems:

1. transform predicates involving application signals into predicates involving controller events only. This section studies this sub-problem, which represents the major part of our work,

2. generate an observer in the target language, from a temporal logic formula. This has been proved to be relatively easy [16, 18]. More details regarding our own framework are given in section 5.

The first issue boils down to translating predicates:

- *event* predicates must be transformed into controller event predicates, what can be achieved by following the connections,

- *comparisons* involve data flow values. The data flows are generally inputs of internal components that emit events when the comparison becomes true or false. Such a predicate is therefore true between the occurrences of a "start" and a "stop" event,

- *activations* and *connections* are either performed at startup, or modified at runtime by internal components. As above, we can identify "start" and "stop" events for the validity of the predicate.

To define truth values that are true between the occurrence of two events, we introduce *interval predicates*, denoted by $[u, v)$. $[u, v)$ is true if u has occurred, but v has not occurred yet. Note that the truth value of an event predicate s is that of the interval predicate $[s, \bar{s})$, with $\bar{s}$ denoting the *absence* of the signal s. This way, solving sub-problem #1 amounts to replacing any predicate in the *original formula* with interval predicates involving controller events only. This yields a new temporal logic formula which is called an *intermediate formula*. Both types of formulae share the same propositional and temporal operators; they differ by the types of acceptable predicates: those of table 1 for original formulae, intervals for intermediate formulae.

However, it is not always possible to find an intermediate formula that is *strictly equivalent* to the original formula. Nevertheless, it is sometimes possible to approximate the original formula by two intermediate formulae, one too strict, one too loose, as will be seen in section 4.3. More precisely, when a formula cannot be translated into an interval, but can be "bracketed" by two intervals, the analysis algorithm creates a *proto-interval* that consists of the pair of bracketing intervals. As a result, the most general algorithm doesn't directly produce intermediate formulae built from intervals, but rather formulae built from proto-intervals, that are called *proto-intermediate formulae.*

In cases where it is possible to produce an intermediate formula equivalent to the original formula, proto-intermediate formulae are simply equal to the intermediate formulae. Otherwise, proto-intermediate formulae are the best approximations for the original formula. More details are given in section 4.4, including an algorithm to produce one or two intermediate formulae from a proto-intermediate formula. In this case, we call these *approximate* intermediate formulae, in contrast to otherwise *exact* intermediate formulae. Table 2 gives the definitions for interval and proto-interval predicates.

Type	Meaning
$[s_i, s_j)$	true when event s_i has occurred, but event s_j has not yet occurred
(A_I, A_O)	approximation of the truth value of a formula by too strict an interval (A_I, inner) and too loose an interval (A_O, outer)

Table 2: Predicates for internal use.

4.2 Building Proto-Intermediate Formulae

The algorithm for translating an original formula to a proto-intermediate formula is based on the following "proto-intermediate formula" `pif`(f) function:

1. if f is of type `(in)`**`active`**(c), look for activation conditions of component c. These conditions are events produced by the controller and processed by an internal component. Return an interval, whose bounds are the controller events that activate and deactivate component c.

2. if f is a connection predicate, proceed as above: look for controller events that are processed by internal components to connect ports, and return an interval whose bounds are the controller events that cause the connection and disconnection.

3. if f is of type *signal* (s), s may either be one of the controller ports (let $p = s$), or s may be connected to such a port p. If p is found, return $[p, \bar{p})$.

4. then, look for f and $\neg f$ in the *when* clause of the *publishes ... when ...* statements[1] of the internal components. If the corresponding internal ports are connected to the controller, let p_f and $p_{\neg f}$ be the controller ports. Then, return $[p_f, p_{\neg f})$. If only an approximate result is found, store it in the *approx* variable, and proceed to the next step.

5. if f is of type $a \star b$ where $\star$ is a binary operator, let $a' =$ `pif`(a) and $b' =$ `pif`(b). If a' and b' are defined and are exact intermediate formulae for a and b, return $a' \star b'$. Or else, if *approx* is defined, return *approx*. If *approx* is not defined, return $a' \star b'$.

6. if f is of type $\star\ a$ where $\star$ is a unary operator, let $a' =$ `pif`(a). As above, return $\star\ a'$ or *approx*.

7. if nothing has been returned so far, `pif` fails.

Remark 1 Proto-intermediate formulae only appear when looking for f in *when* clauses (step 4). If this happens, we do not return the intermediate formula right away, but rather try to give priority to an exact formula possibly found by decomposing f

[1]Identification of formulae can be achieved by using a canonical form, derived from a *normal form*.

(steps 5 and 6). We return an approximate solution only as a last resort.

Remark 2 In steps 5 and 6, formulae are decomposed following the tree structure resulting from the way they were written by the user. We do not consider reorganizing the formulae because: 1) it limits the complexity of the algorithm, 2) safety formulae and *when* clauses are written by the *same* person (the system designer), thus they are likely to share common patterns. Therefore preserving the structure helps identify common sub-formulae.

Example The predicates of statement S_1 can be found directly in *when* clauses, and translated into intervals:

- a look at the textual description of component *BrakesCheck* shows that `brakes_on` corresponds to $[\text{brakes\_pushed}, \text{brakes\_released})$,
- `regul_on` is directly a controller input event, so it corresponds to $[\text{regul\_on}, \overline{\text{regul\_on}})$,
- `throttle << regulator.regulated_cmd` corresponds to $[\text{start\_reg}, \text{stop\_reg})$ (the dynamic connection on the right of figure 1 is treated as an internal component).

This yields IF_1, an *exact* intermediate form:

$$\neg\{(\neg([\text{regul\_on}, \overline{\text{regul\_on}}) \wedge [\text{brakes\_released}, \text{brakes\_pushed})))\ \mathcal{S}\ [\text{brakes\_pushed}, \text{brakes\_released})\} \rightarrow \neg[\text{start\_reg}, \text{stop\_reg}))$$

4.3 General Case: Dealing With *when* Clauses

This section deals with the matching of an original formula f in *when* clauses. The match may be exact as in the example above, but this section gives details about how approximate matches are found.

Let f be an original formula, and let us suppose that there are a number of *publishes... when...* statements, of the form `publishes` s_i `when` a_i. The goal is to find an interval I equivalent to f, i.e. a *start signal*, at which f becomes true, and a *stop signal*, at which f becomes false. To determine the start signal, we look for signals s_i in *when* statements where $a_i = f$, $a_i \Rightarrow f$, or $f \Rightarrow a_i$[2]. The same applies to the stop signal, with f being replaced with $\neg f$. From now on, we only consider the start signal; finding the stop signal is similar.

Each formula a_i is associated to a signal s_i (which is emitted when a_i becomes true). Let's call I^+ and I^- the start and stop signals of f (thus $I = [I^+, I^-)$).

There is a partial order relation $\Leftarrow$ on the set of formulae, and an associated equivalence relation $=$. By structure morphism, these relations respectively induce a partial order relation $\preceq$ and an equivalence relation $\leftrightarrow$ on the set of signals. $a \Leftarrow b$ means that a must be satisfied for b to be satisfied ($b \Rightarrow a$). This means that the event s_b associated to b *cannot happen before* s_a, the event associated to a. Therefore, s_b *must* happen after s_a. The $\preceq$ relation is therefore a temporal order on the occurrence of signals. $s_a \preceq s_b$ means that s_a is always emitted before s_b. Likewise, $\leftrightarrow$ corresponds to the simultaneity of signals. As a consequence, it is possible to build a *Hasse diagram* involving I^+ and the signals s_i that compare to I^+ (see figure 3).

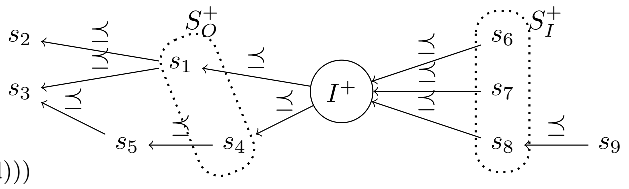

Figure 3: Hasse diagram, with sets S_I^+ & S_O^+.

Among the signals that happen "before" I^+, only the maximum ones, those "closest to I^+", matter. On the example, these are s_1 and s_4. These signals provide the *best possible approximation of* I^+*, while preceding it.* Let S_O^+ be the set of these signals. Formally, it is defined as: $S_O^+ = \{s \in \mathbb{S} | s \preceq I^+ \wedge \neg(\exists s' \in \mathbb{S}, s \preceq s' \preceq I^+)\}$. Likewise, let S_I^+ be the set of *minimum* elements located after I^+, S_I^- the set of *maximum* elements located before

[2]Determining the implication relationships is straightforward using the aforementioned *canonical forms*.

I^-, and S_O^- the set of *minimum* elements located after I^-. S_I^+ and S_O^+ are depicted on figure 3.

Let us define $s_O^+ \in S_O^+$, $s_I^+ \in S_I^+$, $s_I^- \in S_I^-$ and $s_O^- \in S_O^-$. Relative positions of this signals are shown on figure 4. This provides us with a "bracketing" of I: an outer approximate, $[s_O^+, s_O^-)$, and an inner approximate, $[s_I^+, s_I^-)$. This "bracketing" is valid whatever the signals s_O^+, s_I^+, s_I^- and s_O^-:

$$\begin{cases} s_O^+ & \preceq & I^+ & \preceq & s_I^+ \\ s_I^- & \preceq & I^- & \preceq & s_O^- \end{cases}$$

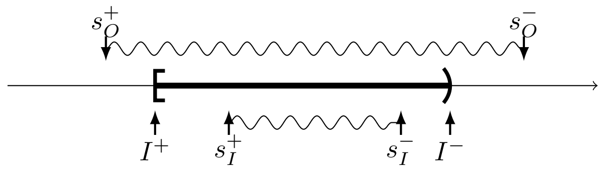

Figure 4: "Bracketing" of I.

We wish to define the *best possible approximation* for I^+ and I^-. Hence, within leftmost members, we consider the *last* signal to occur, and within rightmost members, we consider the *first* signal to occur. This enables us to define intervals A_I (best inner approximation) and A_O (best outer approximation):

$$\underbrace{[\text{first}(S_I^+), \text{last}(S_I^-))}_{A_I} \subset I \subset \underbrace{[\text{last}(S_O^+), \text{first}(S_O^-))}_{A_O}$$

This finishes the complete description of the step #4 in the algorithm of section 4.2:

- if there exist signals s^+ and s^- such that $s^+ \leftrightarrow I^+$ and $s^- \leftrightarrow I^-$, then return the interval $[s^+, s^-)$,
- or else, try to define intervals A_I and/or A_O. Return the pair (A_I, A_O), which is called a *proto-interval*,
- or else, go to step #5.

4.4 Proto-Intermediate Formulae

4.4.1 Definitions

A *proto-interval* is a pair (A_I, A_O) of intervals that constitutes a "bracketing" of the interval I corresponding to a formula f. If $A_I = A_O$, it means that $I = A_I = A_O$ and this is an *exact* match (the set of intervals is trivially embedded into the set of proto-intervals). From now on we will assume that $A_I \neq A_O$. A *proto-intermediate formula* is a formula whose predicates are proto-intervals.

Example In statement S_2, the expression

$e =$ `current_speed < 40 || current_speed > 140`

cannot be found *exactly*. However, if we examine the *when* clauses of component *SpeedCheck*, we see that: 1) $I_e^+ \Rightarrow$ speed_nok, and 2) speed_ok $\Rightarrow I_e^-$.

The interval [speed_nok, speed_ok) is thus an *outer* interval for e. There is no inner interval for e, so the proto-intermediate formula for S_2 is

$$PIF_2 = \text{set\_target} \wedge (\emptyset, [\text{speed\_nok}, \text{speed\_ok}))$$

Intermediate formulae can easily be translated into observers (see section 5). A proto-interval, as a pair of intervals, is thus a pair of intermediate formulae and can therefore be translated into *two* observers: one too loose, one too strict. However, a non-trivial proto-intermediate formula cannot directly be used as such and needs to be *rewritten into a pair of intermediate formulae.*

4.4.2 Transforming Proto-Intermediate Formulae into Pairs of Intermediate Formulae

A proto-intermediate formula f' is rewritten as (f'_I, f'_O), where f'_I is an inner ("strict") intermediate formula, and f'_O is an outer ("loose") intermediate formula. We denote this by $f' \rightsquigarrow (f'_I, f'_O)$. Thus for a proto-interval, we have the trivial rule: $(A_I, A_O) \rightsquigarrow (A_I, A_O)$.

Methodology used Suppose that an original formula f has a proto-intermediate formula f' which is rewritten as (f'_I, f'_O). Then the order relations on the start and stop events give:

$$\begin{cases} f'_O & \Leftarrow & f & \Leftarrow & f'_I & \text{(start event)} \\ \neg f'_I & \Leftarrow & \neg f & \Leftarrow & \neg f'_O & \text{(stop event)} \end{cases}$$

Both relations are equivalent, so we can retain only the first one. Conversely, let f be an original formula.

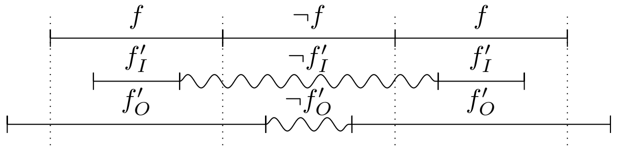

Figure 5: Structure of the intermediate formulae for proto-intermediate formula $g' = \neg f'$.

If there are intermediate formulae g and h such that $g \Leftarrow f \Leftarrow h$, then g and h are respectively inner and outer intermediate formulae for f. In short, $f' \rightsquigarrow (g, h)$.

Negation Let f' be a proto-intermediate formula, associated to an original formula f, with $f' \rightsquigarrow (f'_I, f'_O)$. Let $g' = \neg f'$. The situation is depicted on figure 5.

Formally, one can write $f'_I \Leftarrow f \Leftarrow f'_O$. By taking the contraposition: $\neg f'_O \Leftarrow \neg f \Leftarrow \neg f'_I$. Hence the conclusion, that shows that the negation *inverts* the inner and outer formulae:

$$\text{if} \quad f' \rightsquigarrow (f'_I, f'_O) \quad \text{then} \quad \neg f' \rightsquigarrow (\neg f'_O, \neg f'_I)$$

Conjunction, disjunction and implication Let f' and g' be two proto-intermediate formulae, associated respectively with original formulae f and g. Let us suppose that $f' \rightsquigarrow (f'_I, f'_O)$ and $g' \rightsquigarrow (g'_I, g'_O)$.

We have: $f'_I \Leftarrow f \Leftarrow f'_O$ and $g'_I \Leftarrow g \Leftarrow g'_O$. The relation $\Leftarrow$ is *compatible* with logical `and`[3], thus we have:

$$f'_I \wedge g'_I \Leftarrow f \wedge g \Leftarrow f'_O \wedge g'_O$$

Finally: $f' \wedge g' \rightsquigarrow (f'_I \wedge g'_I, f'_O \wedge g'_O)$. Likewise, $\Leftarrow$ is compatible with $\vee$, thus $f' \vee g' \rightsquigarrow (f'_I \vee g'_I, f'_O \vee g'_O)$.

Implication is dealt with by rewriting $f' \rightarrow g'$ as $\neg f' \vee g'$. By applying the rules seen above:

$$f' \rightarrow g' \rightsquigarrow (f'_O \rightarrow g'_I, f'_I \rightarrow g'_O)$$

Temporal operators We still are under the assumption that $f'_I \Leftarrow f \Leftarrow f'_O$ and $g'_I \Leftarrow g \Leftarrow g'_O$.

[3]The formula $[(a \rightarrow b) \wedge (c \rightarrow d)] \rightarrow [(a \wedge c) \rightarrow (b \wedge d)]$ is a tautology, which can easily be verified.

Let Ψ be a unary temporal operator. Manna et al [24] state that temporal operators are monotonic, hence the relation: $\Psi(f'_I) \Leftarrow \Psi(f) \Leftarrow \Psi(f'_O)$. We thus conclude: $\Psi(f') \rightsquigarrow (\Psi(f'_I), \Psi(f'_O))$.

The same applies to binary temporal operators. If Ψ is a binary temporal operator, we have likewise: $\Psi(f', g') \rightsquigarrow (\Psi(f'_I, g'_I), \Psi(f'_O, g'_O))$.

Conclusion Apart from negation and implication, almost all operators permit a "natural" transformation of proto-intermediate formulae into pairs of intermediate formulae. When doing so, there are two cases:

1. the proto-intermediate formula contains *exact* proto-intervals only. In this case, we finally get only one intermediate formula, which is *exact* too,
2. the proto-intermediate formula contains at least an *approximate* proto-interval. In this case, we finally get one inner and/or one outer intermediate formula(e).

Example The proto-intermediate formula PIF_2 is naturally rewritten as only an outer intermediate formula $IF_2 = \text{set\_target} \wedge [\text{speed\_nok}, \text{speed\_ok})$.

5 From Intermediate Formulae to Observers

As stated above, an intermediate formula is a logic formula that must never be true. Therefore, we have to translate it into an *observer* in the target language. The observer runs in parallel with the controller and emits an error signal in the states where the formula is true. One can then use a *verification tool* either to *prove* that the error signal can never be emitted (and hence that the safety properties are satisfied), or conversely, to exhibit a *counterexample*. The verification tool is generally provided with the target development environment; examples include *checkblif* for Esterel and *lesar* for Lustre.

When the analysis of safety formulae produces *exact* observers, the results of the checking tools directly correspond to the satisfaction or non-satisfaction of the formulae. However, when the analysis produces *approximate* observers, the results are subject to interpretation, and the analysis tool must state it clearly.

Indeed, an observer based on an inner intermediate formula can miss some failure cases because it is *too loose*. However, if the checking tool finds a counterexample, it really corresponds to a case of non-satisfaction of the safety formulae. The checking toolchain thus performs an *under-verification* of the system.

Conversely, an observer based on an outer intermediate formula doesn't miss any real failure case, but it is prone to detecting *false positives*, because it is *too strict*. The checking toolchain thus performs an *over-verification* of the system.

Example An observer based on IF_2 is too strict compared to the statement S_2. This statement ensures that the event `set_target` never occurs when the speed is below 40 or above 140 km/h. However, an observer based on IF_2 will ensure that `set_target` never occurs when the speed is below 40 or above *130* km/h. Thus it may detect “counterexamples” for speeds in the interval 130-140 km/h that are not contradictory with the safety property S_2.

Generation of Observers Generating observers from intermediate formulae is straightforward, and has already been studied both for Esterel [18] and Lustre [14]. Each formula and sub-formula f can be translated into some *module*, *node* or *expression* which emits a signal S_f whenever f is true. For instance in Esterel, if modules `C_a` and `C_b` calculate respectively a and b, the following process calculates $a \mathcal{S} b$ (denoted by a signal S_e):

```
run C_a || run C_b ||
[
    every immediate S_b do
        do
            sustain S_e
        watching immediate [not S_a]
    end every
]
```

6 Conclusions and Perspectives

The analysis method presented here allows the designer to express properties on a system in a natural way, using temporal logic to specify relations among internal or external signals. These properties can then be automatically translated into temporal logic properties on the controller events. From this, observers can be generated in the language used for specifying the controller, and used to prove the properties by model-checking.

In cases in which specified properties do not exactly match controller events, approximate observers can be generated. Although their results are subject to interpretation, they can help system designers detect certain defects and validate part of the behavior of their systems.

We have implemented the analysis tool in Java. This tool reads the textual ADLV description of an application, analyses the safety formulae and the behavior of the internal components, and builds proto-intermediate formulae and intermediate formulae. It can then produce observers for the controller in various languages thanks to a modular structure which requires the definition of just one class for each supported language. This class implements a visitor pattern [12] that traverses intermediate formulae and generates programs in the target language. We provide visitors for Esterel and Lustre, but support for other languages can be added very easily. This tool is available at `http://wwwdi.supelec.fr/logiciels/adlv/`.

Perspectives include handling a wider range of system descriptions. For instance, the current method cannot deal with the cases in which connections between processing components and the controller change at run-time. This extension makes the analysis more complex since the mapping between formulae and controller signals becomes dynamic.

The set of safety properties currently accepted by our tool is limited to *canonical* safety formulae. However, recent work has shown that non-canonical safety formulae can be translated into Büchi automata, i.e. deterministic observers [26]. We could certainly leverage these results to extend the range of acceptable safety formulae.

References

[1] C. Andre. SyncCharts: A visual representation of reactive behaviors. Technical Report TR95-52, Université de Nice-Sophia Antipolis, 1995.

[2] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In *FSTTCS*, volume 4337 of *LNCS*, pages 260–272. Springer, 2006.

[3] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. In *Readings in hardware/software co-design*, pages 147–159. Kluwer Academic Publishers, 2002.

[4] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[5] Ahcene Bouzoualegh, Dominique Marcadet, Frédéric Boulanger, and Christophe Jacquet. An Architecture Description Language for Verification in Component-based Software. In *IEEE Computer Software and Applications Conference (COMPSAC 2008)*, July 2008. Accepted for publication, to appear.

[6] M.S. Branicky, V.S. Borkar, and S.K. Mitter. A Unified Framework for Hybrid Control: Model and Optimal Control Theory. *IEEE Transactions on Automatic Control*, 43(1):31, 1998.

[7] Edward Y. Chang, Zohar Manna, and Amir Pnueli. Characterization of Temporal Property Classes. In *Proceedings of ICALP '92*, pages 474–486. Springer, 1992.

[8] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):253–291, 1997.

[9] R. David. Grafcet: a powerful tool for specification of logic controllers. *IEEE Trans. on Control Syst. Techn.*, 3(3):253–268, 1995.

[10] E.A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science*, 8:995–1072, 1990.

[11] P.H. Feiler, B. Lewis, and S. Vestal. The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering. In *RTAS 2003 Workshop on Model-Driven Embedded Systems*, 2003.

[12] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming langage Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[14] N. Halbwachs, J.C. Fernandez, and A. Bouajjanni. An executable temporal logic to express safety properties and its connection with the language Lustre. In *Sixth International Symposium on Lucid and Intensional Programming*, 1993.

[15] N. Halbwachs and P. Raymond. Validation of Synchronous Reactive Systems: From Formal Verification to Automatic Testing. In *Proc. of the 5th Asian Computing Science Conference on Advances in Computing Science*, pages 1–12. Springer, 1999.

[16] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE. *IEEE Trans. Softw. Eng.*, 18(9):785–793, 1992.

[17] TA Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proceedings of IEEE LICS'92*, pages 394–406, 1992.

[18] Lalita Jategaonkar Jagadeesan, Carlos Puchol, and James Von Olnhausen. Safety Property Verification of Esterel Programs and Applications to

Telecommunications Software. In *Proceedings of CAV'95*, pages 127–140. Springer, 1995.

[19] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74:471–475, 1974.

[20] B. Kuipers. Qualitative Simulation. *Artificial Intelligence*, 29(3):289–338, 1986.

[21] F. Laroussinie and Ph. Schnoebelen. A hierarchy of temporal logics with past. *Theor. Comput. Sci.*, 148(2):303–324, 1995.

[22] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[23] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *Proceedings of FORMATS/FTRTFT*, volume 3253 of *LNCS*, pages 152–166. Springer, 2004.

[24] Zohar Manna and Amir Pnueli. Basic Properties of the Temporal Operators — Monotonicity. In *The Temporal Logic of Reactive and Concurrent Systems Specification*, chapter 3, pages 202–203. Springer, 1992.

[25] S. Merz. Model checking: a tutorial overview. *LNCS*, 2067:3–38, 2001.

[26] A. Morgenstern and K. Schneider. From LTL to symbolically represented deterministic automata. In F. Logozzo, editor, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 4905 of *LNCS*, pages 279–293. Springer, 2008.

[27] A. Tiwari, N. Shankar, and J. Rushby. Invisible formal methods for embedded control systems. *Proc. of the IEEE*, 91(1):29–39, January 2003.