Checking Properties on the Control of Heterogeneous Systems

Christophe Jacquet Dominique Marcadet

SUPELEC
3 rue Joliot-Curie
91192 Gif-sur-Yvette Cedex
FRANCE

first_name.last_name@supelec.fr

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 system. The only constraint is to specify the controller in a synchrounous 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 it is not possible to generate observers that correspond exactly to the specified properties, our tool is capable of generating approximate observers. Alghough the results given by these are subject to interpretation, they can nevertheless 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 a number of *processing components* which communicate through data flows, and control is modeled by a unique *control component* whose inputs and outputs are pure events. Each processing component may be described in an adapted formalism (Simulink, C code, etc.). The controller must be written in a reactive synchronous language (Esterel, Lustre, Signal, etc.) in order to take advantage of model checking tools. An heterogeneous system is described as a network of components, consisting of any number of processing components and a unique controller.

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

The paper is organized as follows. Section 2 provides a short review of related work, and justifies the choice of purely event-driven control components. 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.

This work has been performed in the context of the Usine Logicielle project of the System@tic Paris-Région Competitiveness Cluster.

2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)
978-0-7695-3388-9/08 $25.00 © 2008 IEEE

2. Related Work and Objectives

Most systems are *hybrid* by nature because they mix continuous behaviors and discrete transitions at some points. For example, when the temperature of an ice cube gradually increases, the laws that determine its physical parameters are locally continuous, but they change, first when ice turns into liquid water, and second when liquid water evaporates. One can think of this system as a state machine capable of switching between states (solid, liquid, gas) within which a continuous description applies. Differential equations work well when describing each state independently of the others, but to describe the system as a whole, qualitative simulation may prove useful [1]. Qualitative simulation discretizes continuous parameters into regions delimited by *landmark values* and offers a homogeneous discrete framework for simulating hybrid systems. Work on this topic has recently been extended to hybrid automata [2, 3].

Manufactured systems, and especially heterogeneous ones, are almost always hybrid systems. Indeed, they often mix traditional continuous automatic control methods with digital supervision that switches the system among a number of different states. This led to the development of *hybrid control*. Hybrid control methods can generally be viewed as hierarchic control, with some kind of automaton at the top level, and continuous control at lower levels [4]. The system thus performs transitions between partitions of the continuous state space where its behavior is purely continuous.

The commonly used languages for control in the industry are generally based on discrete transition systems such as state machines. This is the case of Esterel [5], Lustre [6], Signal [7], StateCharts/SyncCharts [8], Grafcet/Sequential Function Chart [9]. However, these languages generally allow operations on numerical values as source of events. For instance, if a is a numerical input, 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, although theoretical model checking techniques for infinite-state systems exist (see for instance [10] for programs with unbounded integer variables), commercially available model checkers can generally operate on purely event-based controllers only. Thus in practice, one *must* exclude all data operations from the controller in order to use model checking.

As a consequence, 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) a control component 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 natural semantics, for instance *"component C is never active when input E is greater than value V"*. These properties are then translated into the event-based semantics used by the control component, 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 transform 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 [5].

Our purpose is *not* to produce a new formal verification tool, but to design an *analysis tool*, capable of transforming properties expressed on the system as a whole into properties on the control component that can be processed by the verification tools available on the market. These tools are generally specific to the target language (i.e. that of the control component), and often come bundled with the language toolchain. In practice, once the properties on the application have been transformed into properties on controller events, they can easily be translated into *observers* [11] written in the target language. In this way, the actual implementation code is checked, so the WYPIWYE paradigm is applied a second time.

3. Architecture Description Language for Verification

3.1. Description of Embedded Applications

We introduce here a method for describing a heterogeneous application 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

2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)
978-0-7695-3388-9/08 $25.00 © 2008 IEEE

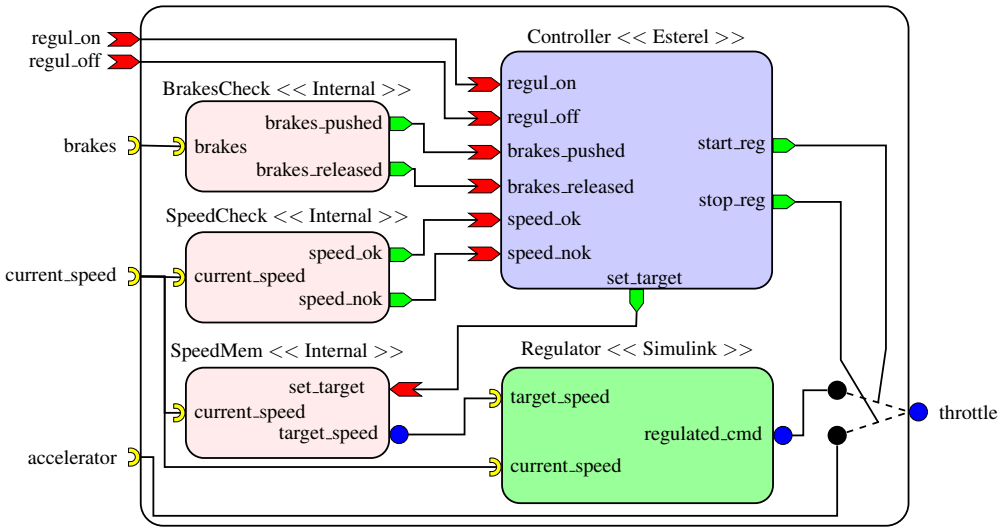

Figure 1. Component-based cruise control described in ADLV.

which extends OMG IDL[1]. 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

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 three 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 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* goes 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;
    }
};
```

[1]OMG IDL is an ISO-standardized Interface Definition Language (ISO 14750), that is part of CORBA. See `http://www.omg.org/technology/documents/formal/corba_2.htm`.

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;
    }
};
```

These two internal components perform an *adaptation* between data flows and the input events of the controller. Thus, internal components play a key role in managing heterogeneity. There is only one control component per application (here, the *Controller* component, which is written is Esterel) which is considered as a black box which consumes and produces events.

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 "regulation 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 "regulation 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.

ADLV descriptions can be used to generate the actual implementation of the system. We will not describe these capabilities 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 [12].

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* [13]. Such formulae offer a good expressive power [14], are simple to use [15, 16] and easy to translate into synchronous reactive languages such as Esterel or Lustre.

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

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

Table 1. List of predicates for safety formulae.

All the predicates can be *negated*, with natural meaning. For instance, $\neg(s < k) = s \geq k$, $\neg(\texttt{active}(c)) = \texttt{inactive}(c)$, 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 appearing in the predicates can be located at the interface between the application and the outside world, as well as internal signals. Safety formulae are part of the ADLV descriptions, alongside the description of processing components and connections.

3.2.2. Example

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 "on" button and he/she releases the brakes*.

The problem with this specification is that it uses a *future* operator: *until*. 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*. This gives 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 */
```

A statement can use only propositional operators:

```
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.

3.3. Checking Safety Formulae

For each safety formula, we must generate an equivalent observer for the controller. However the safety formula can reference signals that are not directly connected to the controller. By analyzing the structure of the application and by looking into the internal blocks, we can build an equivalent temporal formula that only references *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*. Observers are modules written in the same language as the controller (for instance, Esterel or Lustre) which are processed by the 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.

The properties are checked on the synchronous implementation of the controller, which is used to drive the application at run time, thanks to observers directly generated from the application description itself. As stated above, the WYPIWYE is effectively both for the controller and for the internal components.

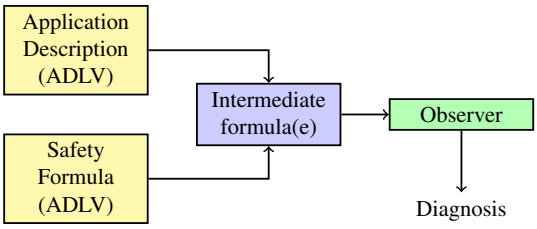

Figure 2. Overview of property verification.

4. Interpreting Temporal Formulae

4.1. Overview

For each formulae 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 [15, 16]. 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* involving 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. Once again, we can identify "start" and "stop" events for the validity of the predicate.

To define truth values that are true between the occurrences of two events, we introduce *interval predicates*, denoted by $[u,v[$. $[u,v[$ is true if u has occurred, but v has not yet occurred. Note that the truth value of an event predicate s is that of the interval predicate $[s,\bar{s}[$. This way, solving sub-problem #1 amounts to replacing any predicate in the *original formula* with interval predicates which involve 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 exactly 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 the interval and proto-interval predicates.

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

Table 2. Predicates for internal use.

4.2. Building Proto-Intermediate Formulae

The algorithm for translating an original formula into a proto-intermediate formula is based on the `pif` ("proto-intermediate formula") function. $\texttt{pif}(f)$:

1. if f is of type `(in)active`(c), look for the activation conditions of component c. These conditions are events produced by the controller and processed 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 a *publishes ... when ...* statement[2] of an internal component. If the corresponding internal ports are connected to the controller, let p_f and $p_{\neg f}$ be the controller's 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 $a' = \texttt{pif}(a)$ and $b' = \texttt{pif}(b)$. If a' and b' are defined and are exact intermediate formulae for a and b, return $a' \star b'$. 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 $a' = \texttt{pif}(a)$. 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 privilege an exact formula possibly found by decomposing f (steps 5 and 6). We return an approximate solution only as a last resort.

Remark 2

In steps 5 and 6, formulae are decomposed according to 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) it takes advantage of the common patterns in safety formulae and when clauses which are written by the *same* person[3].

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

[3]This is similar to what is done in *Global Common Subexpression Elimination* (GCSE) in compiler theory [17, 18]: subexpressions are identified, but the structure is not reorganized.

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:

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

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. to find 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 $f \Rightarrow a_i$[4]. 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 ($I = [I^+, I^-[$).

There is a partial order relation $\cdot \Leftarrow \cdot$ on the set of formulae, and an associated equivalence relation $\cdot = \cdot$. By structure morphism, these relations respectively induce a partial order relation $\cdot \preceq \cdot$ and an equivalence relation $\cdot \leftrightarrow \cdot$ 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

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

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).

Among the signals that happen "before" I^+, only the *maximum* ones, the "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 I^-, and S_O^- the set of *minimum* elements located after I^-. S_I^+, S_O^+, S_I^- and S_O^- are depicted on figure 3.

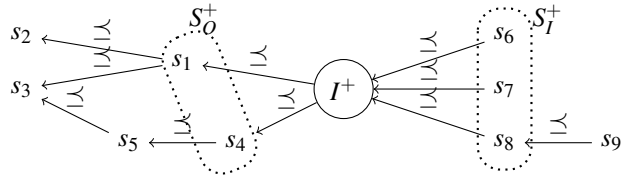

Figure 3. Hasse diagram, with sets S_I^+ & S_O^+.

Let us define $s_O^+ \in S_O^+$, $s_I^+ \in S_I^+$, $s_I^- \in S_I^-$ and $s_O^- \in S_O^-$. The relative positions of these signals are shown on figure 4. This defines a "bracketing" of I by an outer approximate $[s_O^+, s_O^-[$, and an inner approximate, $[s_I^+, s_I^-[$.

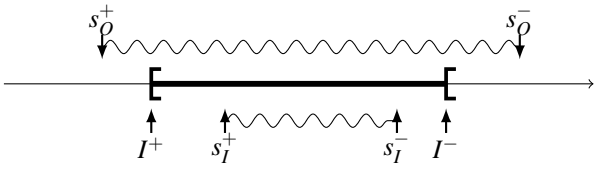

Figure 4. "Bracketing" of I.

This "bracketing" is valid whatever the signals s_O^+, s_I^+, s_I^- and s_O^-:

$$\left\{ \begin{array}{ccccc} s_O^+ & \preceq & I^+ & \preceq & s_I^+ \\ s_I^- & \preceq & I^- & \preceq & s_O^- \end{array} \right.$$

We wish to define the *best possible approximation* for I^+ and I^-. Hence, within leftmost members, we consider the *last* signal to occurr, 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 $[s^+, s^-[$,
- else, try to define intervals A_I and/or A_O. Return the pair (A_I, A_O), which is called a *proto-interval*,
- 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_incorrect, and 2) speed_correct $\Rightarrow I_e^-$.

The interval [speed_incorrect, speed_correct[is thus an *outer* interval for e. There is no inner interval for e, so the proto-intermediate formula for S_2 is $PIF_2 =$ set_target $\wedge (\emptyset, [\text{speed\_incorrect}, \text{speed\_correct}[)$.

Intermediate formulae can easily be translated into observers (see section 5). A proto-interval, as a pair of intervals, thus a pair of intermediate formulae, can thus 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 for the proofs

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

$$\left\{ \begin{array}{ccccc} f'_O & \Leftarrow & f & \Leftarrow & f'_I \\ \neg f'_I & \Leftarrow & \neg f & \Leftarrow & \neg f'_O \end{array} \right. \begin{array}{l} \text{(start event)} \\ \text{(stop event)} \end{array}$$

Conversely, let f be an original formula. 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 with 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:

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

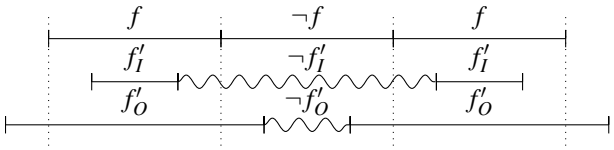

Figure 5. Structure of the intermediate formulae for proto-intermediate formula $g' = \neg f'$.

Conjunction, disjunction and implication

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

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`[5], 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 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 $g'_I \Leftarrow g \Leftarrow g'_O$. Let Ψ be a unary temporal operator. Manna et al [19] 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: $\Psi(f',g') \rightsquigarrow (\Psi(f'_I, g'_I), \Psi(f'_O, g'_O))$.

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

Conclusion

All operators permit a "natural" transformation of proto-intermediate formulae into pairs of intermediate formulae. In the end, either one *exact* or possibly two (one inner and/or one outer) *approximate* intermediate formulae are generated.

Example

The proto-intermediate formula PIF_2 is naturally rewritten as an outer intermediate formula:

$$IF_2 = \text{set\_target} \wedge [\text{speed\_incorrect}, \text{speed\_correct}[$$

5. From Intermediate Formulae to Observers

5.1. Observers and Use Thereof

As stated above, an intermediate formula is a logic formula that must never be true. We have to translate it into an *observer* in the target language, which 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 is never emitted (and hence that the safety property holds), 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 corresponds really 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 omit any failure case, but it is prone to detecting *false counterexamples*, 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 for 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.

5.2. Esterel Observers

Esterel is a synchronous reactive language with a significant user base in the industry, especially for designing safety-critical systems.

It is quite straightforward to associate an Esterel module to every sub-formula f of an intermediate formula. This module emits a signal $\mathscr{S}_f$ whenever f is true. A simple interval $I = [s_1, s_2[$ can be translated into a module which maintains an output signal $\mathscr{S}_I$ *between* the occurrences of s_1 and s_2:

```
every immediate s1 do
    abort sustain S_I when s2
end every
```

For a propositional operator, say $e = a \wedge b$, we first generate the modules corresponding to a and b, respectively `C_a` (output signal $\mathscr{S}_a$) and `C_b` (output signal $\mathscr{S}_b$). Then the module for the *and* operator is:

```
run C_a || run C_b || [
  every immediate [S_a and S_b] do
    emit S_e
  end every
]
```

Modules for temporal operators are implemented in the same way. Examples can be found in other papers [16]. For instance, $e = a \mathscr{S} b$ is translated as:

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

There is also a top-level module, responsible for collecting the signals associated with every individual intermediate formula, and generating the *failure* signal when necessary.

The program containing the observers needs to be compiled alongside the controller code, and then Esterel's standard *checkblif* tool can perform verification.

5.3. Lustre Observers

Regarding its expressive power and use, Lustre is close to Esterel. However, while Esterel is based on modules programmed in an imperative way, Lustre is based on *nodes* programmed in a functional way and through which data flows. A very practical option is to build a library of nodes, corresponding to each of the propositional and temporal operators [20]. For instance, the following nodes calculates $b \mathscr{S} a$:

```
node since(B, A: bool) returns (B_snc_A: bool);
let
  A_occurd = if A then true
             else false -> pre(A_occurred);
  B_snc_A  = if A_occurred then B
             else false; -- true in weak since
tel
```

It is thus possible to build *Lustre expressions* that correspond directly to intermediate formulae. For instance, the intermediate formula IF_1 is translated into:

```
never_3 = not(
  implies(since(not(
        interval(regul_on, not(regul_on))
      and
        interval(brakes_released, brakes_pushed)
      ), interval(brakes_pushed, brakes_released)),
  not(interval(start_reg, stop_reg))));
```

A top-level node emits an output signal *ok* to indicate whether or not all formulae are satisfied. This is used by the standard Lustre verifier called *lesar*.

6. Conclusions and Perspectives

This paper has introduced both a formalism for describing systems made of heterogeneous parts, as well as an analysis method that allows the designer to express properties on the systems in a natural way, using temporal logic to specify relations among internal or external signals. Our "black box" approach as regards the internal specification of the components allows the designer to use any formalism of his/her liking.

The 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 a system.

We have an implementation of the analysis tool in Java which reads the textual ADLV description of an application, analyses the safety formulae, and builds proto-intermediate formulae and intermediate formulae. It can then produce observers 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 [21] 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. The tool is available at `http://wwwdi.supelec.fr/logiciels/adlv/`.

2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)
978-0-7695-3388-9/08 $25.00 © 2008 IEEE

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

References

[1] B. Kuipers, "Qualitative Simulation," *Artificial Intelligence*, vol. 29, no. 3, pp. 289–338, 1986.

[2] A. Tiwari and G. Khanna, "Series of abstractions for hybrid automata," in *Hybrid Systems: Computation and Control HSCC* (C. J. Tomlin and M. R. Greenstreet, eds.), vol. 2289 of *LNCS*, pp. 465–478, Springer, Mar. 2002.

[3] A. Tiwari, N. Shankar, and J. Rushby, "Invisible formal methods for embedded control systems," *Proc. of the IEEE*, vol. 91, pp. 29–39, Jan. 2003.

[4] M. Branicky, V. Borkar, and S. Mitter, "A Unified Framework for Hybrid Control: Model and Optimal Control Theory," *IEEE Transactions on Automatic Control*, vol. 43, no. 1, p. 31, 1998.

[5] G. Berry and G. Gonthier, "The ESTEREL synchronous programming language: design, semantics, implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, 1992.

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming langage Lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[7] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming real-time applications with SIGNAL," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1336, 1991.

[8] C. Andre, "SyncCharts: A visual representation of reactive behaviors," Tech. Rep. TR95-52, Université de Nice-Sophia Antipolis, 1995.

[9] R. David, "Grafcet: a powerful tool for specification of logic controllers," *IEEE Trans. on Control Syst. Techn.*, vol. 3, no. 3, pp. 253–268, 1995.

[10] T. Bultan, R. Gerber, and W. Pugh, "Model-Checking Concurrent Systems with Unbounded Integer Variables: Symbolic Representations, Approximations, and Experimental Results," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 4, pp. 747–789, 1999.

[11] 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*, pp. 1–12, Springer, 1999.

[12] E. Emerson, "Temporal and modal logic," *Handbook of Theoretical Computer Science*, vol. 8, pp. 995–1072, 1990.

[13] E. Y. Chang, Z. Manna, and A. Pnueli, "Characterization of Temporal Property Classes," in *Proceedings of ICALP '92*, pp. 474–486, Springer, 1992.

[14] F. Laroussinie and P. Schnoebelen, "A hierarchy of temporal logics with past," *Theor. Comput. Sci.*, vol. 148, no. 2, pp. 303–324, 1995.

[15] N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE," *IEEE Trans. Softw. Eng.*, vol. 18, no. 9, pp. 785–793, 1992.

[16] L. J. Jagadeesan, C. Puchol, and J. V. Olnhausen, "Safety Property Verification of Esterel Programs and Applications to Telecommunications Software," in *Proceedings of CAV'95*, pp. 127–140, Springer, 1995.

[17] J. Cocke, "Global common subexpression elimination," *Proceedings of a symposium on Compiler optimization (SIGPLAN)*, pp. 20–24, 1970.

[18] A. V. Aho, R. Sethi, and J. D. Ullman, "Machine-Independent Optimizations," in *Compilers: Principles, Techniques, and Tools*, ch. 9, Addison-Wesley, 2006.

[19] Z. Manna and A. Pnueli, "Basic Properties of the Temporal Operators — Monotonicity," in *The Temporal Logic of Reactive and Concurrent Systems Specification*, ch. 3, pp. 202–203, Springer, 1992.

[20] N. Halbwachs, J. 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.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

978-0-7695-3388-9/08 $25.00 © 2008 IEEE