URL: http://www.elsevier.nl/locate/entcs/volume62.html 13 pages

Specifying and verifying reactive systems in a multi-language environment [⋆]

Agathe Merceron [a,1], Monica Müllerburg [b,2], G. Michele Pinna [c,3]

[a] *LIAFA - Université Denis Diderot, France*

[b] *GMD - Forschungszentrum Informationstechnik, Germany*

[c] *Dipartimento di Matematica, Università di Siena, Italy*

Abstract

The multi-language environment Synchronie supports the design and formal verification

of synchronous reactive systems. It integrates three synchronous languages and also three ways to specify properties: the temporal logic with future operators CTL, the temporal logic with past operators Past TL, and observers, which are particular synchronous programs. It is argued that this multi-language feature provides an answer to two major issues of formal verification: facility of formalizing properties and facility of verifying large systems. The approach is illustrated with the case study of a time-triggered protocol.

1 Introduction

Reactive systems are real-time systems that continuously react to stimuli from their environment. They consist of software, hardware, and mechatronic components, thus combining discrete and continuous behaviour. Reactive systems are critical components of our everyday lives, be they in control systems like airbag controllers in cars, or in consumer products like washing machines and phones. Synchronous languages have been proposed to design and formally specify reactive real-time systems. The synchronous paradigm is considered as particularly well suited, since properties such as reactivity and satisfaction of timing constraints can be guaranteed. Reactivity means that the system

[⋆] Work partially supported by the project TOSCA.

[1] Email: agathe@liafa.jussieu.fr

[2] Email: muellerburg@gmd.de

[3] Email: pinna@unisi.it

always reacts to an input and the *unique* reaction terminates within a fixed amount of time. The synchronous paradigm views a system run as a sequence of events which individually consist of a computation step and the exchange of signals (possibly with data) between the various components of the system and their environment. Synchronous languages are based on the synchrony hypothesis [2] which stipulates that calculations are carried out quickly enough so that, for the environment, output signals appear to be synchronous with the inputs. Thus, time is divided into non-overlapping instants where reactions take place. It is worth noticing that a synchronous system is deterministic, in spite of the fact that it can be made of several parallel components.

Several synchronous languages have been developed (see [8] for a survey) : Argos, a graphical language [13], and Esterel, an imperative language [3], are both state based and thus well suited for modelling discrete behaviour while Lustre, a data flow language [5], is instead well suited for modelling continuous behaviour. Integrating these three languages enables multi-style design: each system component may be modelled using the language that best captures the kind of behaviour. The workbench Synchronie [21,20,24] presents such an integration. It is extended to include object-oriented aspects in sE [23], Other integration platforms include scade [22], and an extension of Esterel [7].

Temporal logic, in particular CTL (Computation Tree Logic) [6], is well established for specifying properties of reactive systems, and model checkers may be used for their verification [25,14]. Checking that a property specified as a formula written in a suitable temporal logic is satisfied by a system is an automatic process. A model checker searches the reachability space to conclude whether the check passes. There are, however, two problems: (1) specifying properties in a temporal logic may be awkward for some types of properties; (2) the state explosion problem for large reachability spaces makes actual proving not feasible in many cases.

The paper discusses how verification is done in the multi-language environment Synchronie and shows how it copes with these limitations using a case study, namely a time-triggered communication protocol [26,10] (time-triggered protocols are getting particular attention in the transport industry where electronic systems, so-called "by-wire" systems, replace mechanical systems for stearing, braking and such).

In Synchronie (swb for short) the integration of languages is achieved via a common semantic model, called *Synchronous Component* [21]. swb compiles modules written in any of the above mentioned language into this model. *Boolean Automata* constitute the core part of *Synchronous Components*. A *Boolean Automaton* (ba for short) is a kind of finite state machine encoded by a set of Boolean equations. It is used to represent the calculation of the instantaneous reaction: given a set of input signals provided by the environment, it calculates the set of output signals to be fed back to the environment. All the tools of the workbench, like the reactivity checker (it checks whether or not a program is *reactive*, i.e. given an input, there is a unique reaction), the simulator and the code generator work on this common model.

ba's can be easily transformed into the input format of existing model checkers such as SMV [14] or VIS [25]. Since swb supports CTL, the traditional temporal logic based verification is possible. However, swb provides additional ways to prove properties. One way is to use a logic with past temporal operators, that we call Past TL, particularly suited to specify safety properties [12]. Another, original and, for many practitioners, simpler way of specifying safety properties is through observers [9]. An observer is another synchronous program which observes the signals received and emitted by a synchronous system. The observer emits a special error signal as soon as the synchronous system violates the property it observes. Because observers are synchronous programs, the same language which is used for modeling the system may also be used to formally specify safety properties. Since swb provides more than one synchronous language, any of them may be used. Thus, swb provides not only multi-style *design* but also multi-style *verification*, which helps to make the specification of formal properties a more natural and handy process.

One solution to the state explosion problem is modular verification [11]. It makes the whole automatic verification process easier and, for large systems, feasible at all. The multi-language environment swb provides full support for modular design and modular verification [17].

The paper is structured as follows: in the next section we briefly discuss the role of boolean automata in a multi-language environment, then, in section 3, we illustrate how verification is done. Finally we illustrate, in section 4, the case study.

2 The role of boolean automata

Coherence in swb is achieved via the translation of all components in a common format called ba's. More detailed discussions on ba's may be found in [20,21] where additional insight into synchronous languages and their integration is provided. ba's have two kinds of statements, one for defining *signals* to represent transient information

$s \Leftarrow \phi$: the signal s is emitted if the condition ϕ is satisfied,

and another for defining *registers* to represent persistent information

$h \leftarrow \phi$: the control register h is true (or 'set') in the next instant if ϕ is satisfied.

In the statements above, ϕ is a boolean expression defined on signals and registers.

The operational semantics is defined by two successive phases: given a valuation v, that assigns a truth value (true or false) to each register, and inputs (intuitively the *free* variables), a reaction is the solution of the system equations $\{s \Leftarrow \phi\}$; this solution extends v to cover all the signals, and this is used to compute the assignments $\{h \leftarrow \phi\}$ to yield the next state of the machine. A solution to the signal equations for all input patterns and reachable states must be proved to exist at compilation time to guarantee that the program is *reactive* in that it may respond to any input stimulus. Moreover the solution must be unique to guarantee that the program is *deterministic*.

The following BA represents the switcher of `application`, see Fig. 2: it has two states (`a_off` and `a_on`) and it switches from a state to the other upon reception of the signals `on` and `stop`. Depending on the state and the signal received it emits `start`:

$$\texttt{start} \Leftarrow \texttt{a\_off} \wedge \texttt{on}$$

$$\texttt{a\_off} \leftarrow \alpha \vee (\texttt{a\_off} \wedge \neg\texttt{on}) \vee (\texttt{a\_on} \wedge \texttt{stop})$$

$$\texttt{a\_on} \leftarrow (\texttt{a\_off} \wedge \texttt{on}) \vee (\texttt{a\_on} \wedge \neg\texttt{stop})$$

Signal `start` is emitted if the automaton is in state `a_off` and signal `on` is present. Register `a_off` is set in the initial reaction, represented by the special signal α, or if the automaton is in state `a_off` and signal `on` is not present, or if the automaton is in state `a_on` and signal `stop` is present. Register `a_on` is set if the automaton is in state `a_off` and signal `on` is present or it is in state `a_on` and `stop` is not present.

Each component of the system is translated into a BA and these are composed using (among others) two relevant operations: parallel composition and refinement. Putting BA's in parallel is obtained simply by putting side by side all the statements of the automata, provided that they have a disjoint set of registers. Refinement needs something more: the initial signal α has to be replaced by the starting condition of the refined state, and the negation of its preemption condition has to be added [24]. The translation into BA's and the composition of modules on this intermediate format has several advantages. First, it allows the composition of modules written in various synchronous languages, every language that translates into BA's. Second, it makes multi-style verification possible, as we see next.

3 Specification and verification in the multi-language environment

In SWB properties may be specified in the temporal logics CTL and Past TL , or as observers in any of the supported synchronous languages. CTL is a branching time temporal logic with future operators, while Past TL is a linear time temporal logic using only past operators. Observers are synchronous programs.

CTL is widely used for automatic verification as it is available on model checkers. It uses temporal operators like G, *generally*, which is interpreted as *in each instant later*; the operator X, *next*, interpreted as *in the next instant*; and F, *future*, interpreted as *in some instant later*. It uses also two paths quantifiers: A for all and E for exists.

Take the property '`start` is always followed sometime later by `stop`'. In CTL it is written : AG (`start` $\rightarrow$ F `stop`). This property concerns something that will happen in the future, (liveness property), and CTL is very well suited to express this kind of properties. However CTL may be awkward to express safety properties. For instance consider '`stop` is always preceded by `start`' which is a property verified by `application`. To express it with CTL one has to cast it into a form using future operators, something like 'if there is no `start`, then later there is no `stop`'. By contrast it is easily expressed using a temporal logic with past operators like Past TL. Past TL provides the operator *Once* which is interpreted as *there is some preceding instant*. Reformulating '`stop` is always preceded by `start`' with 'at every instant where `stop` is present, there is a preceding instant where `start` was present', which remains quite intuitive, gives the Past TL formula: AG (`stop` $\rightarrow$ *Once* `start`).

Properties involving particular patterns in signal sequences are difficult to express in temporal logic, either CTL or Past TL. An illustrative example is '`start` and `stop` alternate and `start` begins'. By contrast, such a property is easily formulated as an observer, e.g. written in ARGOS as shown in Fig. 1.

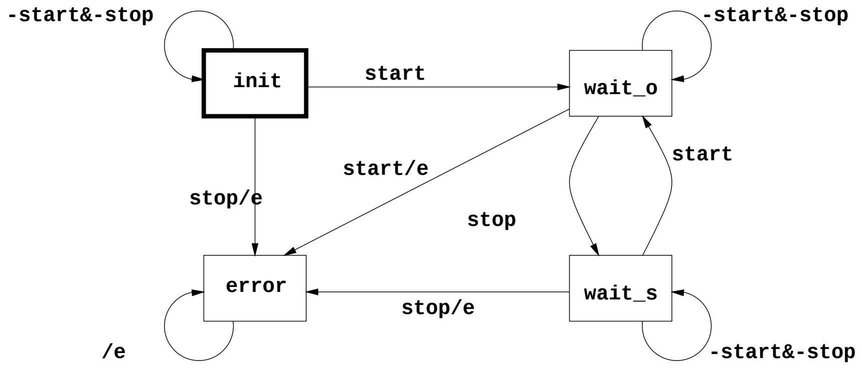


Fig. 1. The automaton for `start` and `stop` alternate and `start` begins.

Expressing invariants like '`start` and `stop` are exclusive' is easy in any of these formalisms. This property can be expressed as an observer in LUSTRE:

```
node  MutExcl(start, stop : bool) returns (e : bool);
let
  e = start and stop;
tel
```

Verification of properties expressed in CTL, Past TL or as observers using a synchronous language is automatic in SWB. BA can be transformed into inputs for available model checkers. Thus CTL formulas are verified using a model checker. One can associate observers expressed directly in BA to Past TL properties as shown in [4]. Verifying a property expressed as an observer simply consists of checking the simple safety property *AG not* `e`, where `e` is the special error signal, on the state space of the program composed in parallel with the observer, using again a model checker.

Though formal verification through model checking is very attractive it has a serious drawback: it is not feasible when the number of states increases dramatically, which is the case for large systems. Modular verification is one answer to the state explosion problem, and it makes the whole verification easier and, for large systems, feasible at all.

In SWB, modules are given by the design: they are the parallel, the refined and the refining components that all together constitute the program . Modular verification makes sense only if it is conservative, i.e. if properties proved for a component do also hold for the whole program. Fortunately a number of nice results have been proved for parallel composition and can be applied in SWB. Among these results, we are interested in those concerning properties expressed as ∀CTL formulas. ∀CTL formulas are CTL formulas using only the quantifier ∀ (namely *A*). In our setting, i.e. boolean automata and SWB, it is possible to cast the results as follows: provided that the output signal sets of the modules are disjoint, a property expressed as a ∀CTL formula proved on a module, is also true for the complete program obtained by refining the module with another one or by putting it in parallel with another one (as shown in [15,17]). It it worth noticing that the result does not apply to a *refining* module. Other results concern the use of other temporal logics or observers [9]. While observers are more limited than a logic like CTL in that they can only express safety properties, a stronger result is obtained for modular verification. Provided that the output signal sets of the modules are disjoint, a property expressed as an observer or as a Past TL formula proved on a module holds also for the complete program; it does not matter whether the module is a refined one, a refining one or a parallel one [17,16].

Having different formalisms to express properties gives flexibility in many ways: on *how* to formally specify properties (the most appropriate formalism may be used depending on the particular property as shown above); on *who* formally specifies properties (not only temporal logic specialists may specify properties, designers used to synchronous languages can specify properties as well via observers); and finally on *when* are properties to be formally specified (core properties to verify are part of the informal system requirements and may be formalized at a very early stage, possibly before the design begins; however, as the designer proceeds, more properties worthwhile to verify may be identified). Hence formal specification of properties may continue in parallel with the design and programming of the system, an approach similar to *extreme programming* put forward in [1] for testing.

4 The case study: a time-triggered protocol

The protocol for automotive applications presented in [26] is a time-triggered protocol [10] which allows a fixed number of stations to communicate via a shared bus. Messages are broadcasted to all stations via the bus. Each station that participates in the communication sends a message when it is the *right* time to do so. Therefore, access to the bus is determined by a time division multiple access (TDMA) schema controlled by the global time generated by the protocol. A TDMA cycle is divided into *time slices*. The stations are ordered and time slices are allocated to the stations according

to their order. During its time slice, a station has exclusive message sending rights. Focusing on control, we consider a time slice as being composed of two parts: a *frame*, which includes the identification of the station, and an *acknowledgement window*.

The protocol is fault-tolerant. If a message is not properly received, the station(s) that notice(s) the transmission error send(s) a veto in the acknowledgement window and all the stations change from normal mode to

error mode. If the faulty sending station is not able to recover it will be excluded. For the fault-tolerance algorithm to work properly, each station maintains a *membership service*, which indicates which stations are in normal mode.

To participate in the communication, a station starts in the initialization mode. It enters the normal mode as soon as it succeeds synchronizing with other stations. Suppose $station_i$ wants to communicate. It has to find out when it is its time to access the bus. Therefore, it goes first in the initialization mode. Upon entering it, it sets its timer to a time `T1`. During this time, it listens to the traffic on the bus. As soon as it recognizes

a valid frame, i.e., a frame whose 'ack window' does not have any veto, it knows which station is sending. Consequently, it can deduce when it will have to send, since time slices are statically allocated. It synchronizes and enters the normal mode, updating its membership service for any valid frame. If $station_i$ is the first to require a communication, it won't hear any valid frame during `T1`. When time `T1` elapses, it sends a frame and sets its timer to $\mathtt{T2}_i$, a time proper to $station_i$ and different from any $\mathtt{T2}_j$ for any $station_j$ with $i \neq j$. Communication needs at least two partners, therefore $station_i$ will continue listening to the traffic during time $\mathtt{T2}_i$, sending a message when $\mathtt{T2}_i$ elapses and resetting its timer to $\mathtt{T2}_i$. If a second station, say $station_j$, needs also to communicate, it enters the initialization mode too. Because $\mathtt{T2}_i$ and $\mathtt{T2}_j$ are different, eventually one of the two stations will recognize a valid frame and synchronize, thus switching to the normal mode. It sends a message in its own time slice, allowing the second station to synchronize as well. Notice that during the set up of the protocol, collisions may occur, for instance if two stations start in the same instant, since `T1` is the same for everybody.

Besides its time-triggered aspect, a relevant feature of the protocol is to allow a quick change of mode and a membership service with minimal overhead both in message length and in the number of messages.

4.1 Formal description of the protocol

We illustrate briefly the overall structure and some parts of the realization of the protocol. Some details are given for the parts involved in the formal proofs. Further details can be found in [18]. The synchronous approach gives the global clock for free. A time unit for the protocol is the reaction time of its synchronous realization.

The protocol is modeled by a number of stations, three in the actual implementation ($1 \leq i \leq 3$), running in parallel. The bus is modeled implicitly as the set of global messages, i.e. those messages that are not local to any station. A frame sent by $station_i$ is modeled by two signals, `bframe_`i and `eframe_`i: the first one represents that the station begins to send a frame, and the second

the end of this operation. In the actual implementation they come in two consecutive instants. A time slice has a length of three time units (instants): the first two are for the frame and the third one is for the acknowledgement window. Taking advantage of the graphical style of Argos and of its refinement operator

for states, the top level of the protocol consists of parallel single state automata: one for each station and one for the collision controller. The collision controller checks whether signals on the bus collide. It emits the signal `rframe_`i in the third instant of the time slice if the frame sent by $station_i$ does not collide with any other frame and it is accepted by the other stations. All states are further refined. The states for the stations are refined by the same Argos automaton, called `station` and shown in Fig. 2. Only the parameters, which are signals, differ. Within `station`, `station_f` denotes the first follower station, `station_s` the second, and signals `x` specific to `station_f` ar e called `x_f`.

The Argos automaton for `station` (Fig. 2) consists of three parallel components. Each component is again an Argos automaton. The automaton `application` models the status of the station which is either off (state `a_off`) or on

(state `a_on`). The automaton `engine` represents the station's protocol engine, which is either `idle` or `working`. The state `working` is refined by the Argos automaton shown partly in Fig. 3. The automaton `timer` is responsible for counting the instants in the initialization mode.

Starting an automaton with parallel components, all parallel initial states are entered, hence starting the automaton in Fig, 2, the three parallel initial states `a_off`, `idle`, `t0` are entered. Further, because of the synchrony hypothesis and broadcasting of signals, the following happens when `station` needs its protocol engine. Upon a request from the environment (issued via

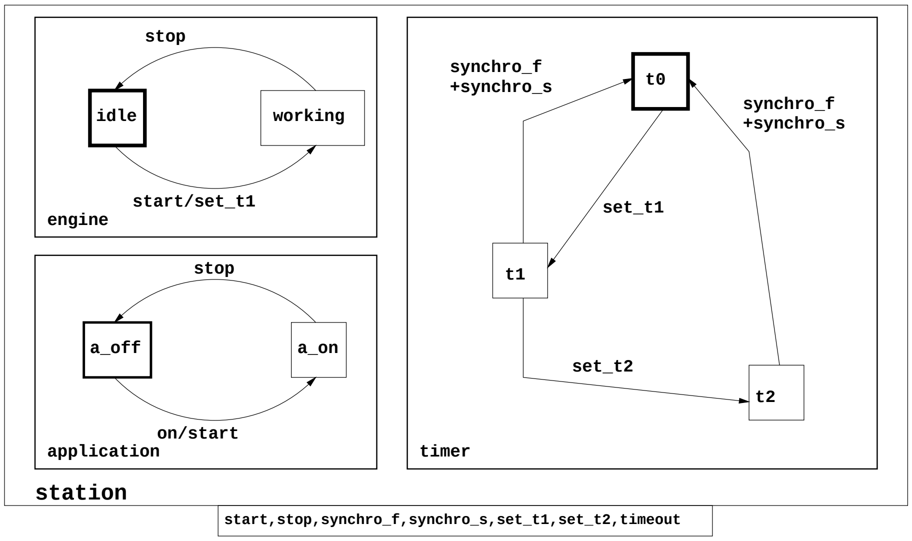


Fig. 2. ARGOS automaton for *station*

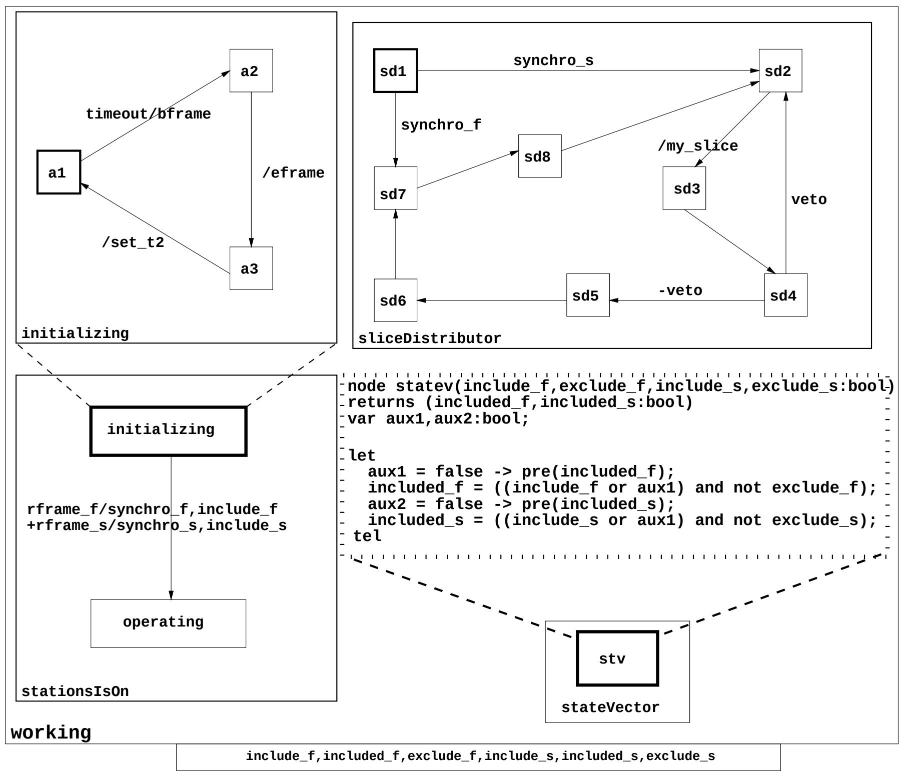
the input signal `on`), the automaton `application` emits the signal `start` and enters the state `a_on`. Signal `start` is immediately received by `engine` which

changes to state `working`, emitting `set_t1` which is in turn received by the `timer` causing it to change to state `t1` for counting the first time (`T1`). As soon as `timer` receives `synchro_f` or `synchro_s` (representing the success in synchronization with one of the two other stations), it goes back in its initial state. If the station does not succeed in synchronization within time `T1` the counter issues the signal `timeout` at the end of the counting, the timer starts to count (in state `T2`) and another synchronization attempt is performed. The counter is periodically reset to `T2` until the synchronization succeed.

The `working` state of `engine` is refined by three automata running in parallel: `stationIsOn` models the status of the protocol engine which is either initializing or operating, `stateVector` models the membership service, and `sliceDistributor` decides whether it is the station's turn to send its frame (see Fig. 3).

The protocol engine is either in the initialization mode or in the operating mode, where operating mode covers normal and error mode. Hence, the automaton `stationIsOn` consists of the two states, `initializing` and `operating`. Both states are further refined by ARGOS automata. Fig. 3 shows how the state `initializing` is refined.

The automaton `stateVector`, modelling the membership service, consists of just one state which is refined by a LUSTRE node, as shown in Fig. 3. The equations for `aux1`, `aux2`, `included_f`, and `included_s` are evaluated in parallel. Variable `aux1`, for instance, has the value `false` in the first instant, and in all further instants it has the value that `included_f` had in the previous instant. The state vector represents the state of the system as it is perceived by the station: each station knows which stations are participating to the game, i.e. which are `included`. Those stations which are not included are



```
node statev(include_f,exclude_f,include_s,exclude_s:bool)
returns (included_f,included_s:bool)
var aux1,aux2:bool;

let
   aux1 = false -> pre(included_f);
   included_f = ((include_f or aux1) and not exclude_f);
   aux2 = false -> pre(included_s);
   included_s = ((include_s or aux1) and not exclude_s);
 tel
```

Fig. 3. Argos automaton refining state `working`

ignored. The `sliceDistributor` decides whether it is the station's turn to send its frame. Therefore, it keeps track of the time slices and emits the signal `my_slice` when the time slice to send a frame has arrived. In addition, it notices whether the frame sent by another station, say `station_f`, was rejected. In this case it emits the signal `exclude_f` causing the `stateVector` to exclude `station_f`. This is done by a Lustre program which refines states `sd6` and `sd8` not further discussed here.

4.2 Specification and verification of protocol properties

The following three properties were identified as particularly important:

Property 1: stations will synchronize.

Property 2: there is no collision on the bus.

Property 3: the membership service does function correctly.

To cut down complexity, properties have been verified modularly as much as possible.

Property 1: stations will synchronize
This property refers to something good that will happen in the future (liveness property). It is therefore best specified using a temporal logic with future operator, hence CTL.

A station will synchronize if (1) it is able to send a frame and (2) if there is

another station that wants to synchronize as well or has already synchronized.

The property was proved in two parts. If `station` is in the state `a_off` and `on` is present, then it will always send its frame. For the second part, it will synchronize, `rframe_f` (or `rframe_s`) must be emitted sufficiently often. This will be the case if `station_f` and `station_s` are not both in state `a_off`. Thus the second property is proved with the fairness constraint `rframe_f` *or* `rframe_s`. For both parts of the proof it is assumed that `off` remains absent, which is safe, as `off` is emitted only in the error mode, not in the initialization. The CTL formulae are: $AG((\texttt{a\_off}\ \textit{and}\ \texttt{on}) \Rightarrow AF\ \texttt{bframe})$ and $AG((\texttt{a\_off}\ \textit{and}\ \texttt{on}) \Rightarrow AF\ (\texttt{synchro\_f}\ \textit{or}\ \texttt{synchro\_s}))$. The temporal operator AF stands for always in the future. Putting these properties together, one can conclude that a station will synchronize. This property has been proved on the module `station` dropping the refining components for `sliceDistributor`, `stateVector`, and `operating`, since those are not concerned with the signals involved in the property.

Property 2: there is no collision on the bus
This property requires that any two stations never send a frame in the same instant. This is the case if their slice distributors do not emit `my_slice` in the same instant. This is quite easy to formalize as a LUSTRE observer. The following equation ensures the property for any two stations $station_1$ and $station_2$: $\texttt{e} = \texttt{my\_slice\_1}\ \textit{and}\ \texttt{my\_slice\_2}$. The property $AG\ \textit{not}\ \texttt{e}$ was proved as true.

Property 3: the membership service does function correctly
For proving this property it was shown that the LUSTRE node for `StateVector` is correct. The proof was done in two steps. First, if $station_f$ is included in the state vector of *station*, then it has always been included since the signal `include_f` has been emitted. The property is best formalized using a logic with past operators: $AG\ (\texttt{included\_f} \Rightarrow (\texttt{included\_f}\ \textit{Since}\ \texttt{include\_f}))$. Further, if a station is not included, then either it has never been included or it has not been included since an exclude has been emitted. Again, this is best formalized by a Past TL formula: $AG\ (\textit{not}\ \texttt{included\_f} \Rightarrow (P\ \textit{not}\ \texttt{included\_f})\ \textit{or}\ (\textit{not}\ \texttt{included\_f}\ \textit{Since}\ \texttt{exclude\_f}))$ where P should be red as *always in the Past.* Property 3 was proved on the module `stateVector` only.

References

[1] Beck, K., "Extreme Programming explained: embrace change" Addison Wesley Longman, 2000.

[2] Benveniste, A. and G. Berry, *The synchronous approach to reactive and real-*

time systems, Proceedings of the IEEE **79** (1991).

[3] Berry, G. and G. Gonthier, *The Esterel synchronous programming language: Design, semantics, implementation*, Science of Computer Programming **19** (1992), pp. 87–152.

[4] Budde, R. and A. Merceron, *A generator of boolean acceptors for safety properties*, in: *5th Australasian Conf. on Parallel and Real-Time Systems (Part'98)*, Springer-Verlag, 1998, pp. 45–56

[5] Caspi, P., D. Pilaud, N. Halbwachs and J. Plaice, *Lustre: a declarative language for programming synchronous systems*, in: *14th ACM Conf. on Principles of Programming Languages*, ACM Press, 1987, pp. 178–188.

[6] Clarke, E. M., E. Allen Emerson and A. Prasad Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Transactions on Programming Languages and Systems **8** (1986), pp. 244–263.

[7] Esterel homepage, *http://www.esterel.org*.

[8] Halbwachs, N., "Synchronous Programming of Reactive Systems," Kluwer Academic Publishers, 1993.

[9] Halbwachs, N., F. Lagnier and P. Raymond, *Synchronous observers and the verification of reactive systems*, in: *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Workshops in Computing, Springer-Verlag, 1993.

[10] Kopetz, H. and G. Grünsteidl, *A time triggered protocol for automotive applications*, Research report 16/92, Institut für Technische Informatik, Technische Universität Wien, Vienna, Austria (1992).

[11] Kupferman, O. and M.Y. Vardi, *Modular model checking*, in: *Proc. Com positionality Workshop*, Lecture Notes in Computer Science **1536**, Springer-Verlag, 1998.

[12] Manna, Z. and A. Pnueli, "The Temporal Logic of Reactive and Concurrent Systems - Specification," Springer Verlag, 1992.

[13] Maraninchi, F., *The* ARGOS *language: Graphical representation of auto mata and description of reactive systems*, in: *IEEE Workshop on Visual Languages*, IEEE Society Press, 1991.

[14] McMillan, K. L., "Symbolic Model Checking," Kluwer Academic Publishers, 1993.

[15] Merceron, A. and G. Michele Pinna, *Modular verification of* ARGOS *programs*, in: *6th Australasian Conf. on Parallel and Real-Time Systems(Part'99)*, Springer-Verlag, 1999, pp. 367–376.

[16] Merceron, A. and G. Michele Pinna, *Refinement and modular verification with observers*, in: *1st Asia-Pacific Conference on Quality Software (APAQS 2000)*, IEEE Society Press, 2000, pp. 104–114.

[17] Merceron, A. and G. Michele Pinna, *Component-based verification is a synchronous setting*, International Journal of Software Engineering & Knowledge Engineering **11** (2001), pp. 181–203.

[18] Merceron, A., M. Müllerburg and G. Michele Pinna, *Verifying a time triggered protocol in a multi language environment*, in: *Proceedings of Safecomp '98*, Lecture Notes in Computer Sciences **1516**, Springer-Verlag, 1998, pp. 185–195.

[19] Poigné, A. and L. Holenderski, *Boolean automata for implementing* ESTEREL, Arbeitspapiere der GMD 964, GMD (1995).

[20] Poigné, A. and L. Holenderski, *On the combination of synchronous languages*, in: *Compositionality: The Significant Difference*, Lecture Notes in Computer Science **1536**, Springer-Verlag, 1999, pp 490 –514.

[21] Poigné, A., M. Morley, O. Maffeïs, L. Holenderski and R. Budde, *The synchronous approach to designing reactive systems*, Formal Methods in System Design **12** (1998), pp. 163–187.

[22] SCADE homepage, *http://www.telelogic.com/products/scade/.*

[23] sE homepage, *http://set.gmd.de/ees/synchronie/doc/lang.html.*

[24] SYNCHRONIE homepage, *http://set.gmd.de/ees/synchronie/swb.html.*

[25] The VIS Group, *VIS: A system for verification and synthesis*, in: *Proc. 8th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science **1102**, Springer-Verlag, 1996, pp. 428–432.

[26] v. Hanxleden, R., J. Bohne, L. Lavagno and A. Sangiovanni-Vincentelli, *Hardware/software co-design of a fault-tolerant communication protocol*, in: *Proc. of the IEEE International Workshop on Embedded Fault-Tolerant Systems*, Dallas, 1996.