Extending Lustre with Timeout Automata

Jimin Gao\textsuperscript{a,1,2} Mike Whalen\textsuperscript{b,3} Eric Van Wyk\textsuperscript{a,1,4}

\textsuperscript{a} Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN, USA

\textsuperscript{b} Advanced Technology Center
Rockwell Collins, Inc.
Cedar Rapids, IA, USA

Abstract
This paper describes an extension to Lustre to support the analysis of globally asynchronous, locally synchronous (GALS) architectures. This extension consists of constructs for directly specifying the timeout automata used to describe asynchronous communication between processes represented by Lustre nodes. It is implemented using an extensible language framework based on attribute grammars that allows such extensions to be modularly defined so that they may be more easily composed with other language extensions.

Key words: synchronous languages, extensible languages, attribute grammars, composable language extensions

1 Introduction

Synchronous languages \cite{2} have been successfully used to describe and reason about a wide variety of systems, including hardware design and synthesis \cite{24}, embedded software control \cite{2}, and modeling and analysis of globally asynchronous, locally synchronous (GALS) architectures \cite{15}. These can be seen as domain-specific languages that address the concurrency and synchronization concerns of embedded systems and hardware at a high-level of abstraction.

The Lustre language \cite{14}, in particular, has been used in a wide range of academic and industrial projects. To better suit specific communities, the Lustre language has evolved into different dialects that further specialize the

\textsuperscript{1} This work is partially funded by NSF CAREER Award #0347860, NSF CCF Award #0429640, and the McKnight Foundation.
\textsuperscript{2} Email: jgao@cs.umn.edu
\textsuperscript{3} Email: mwwhalen@rockwellcollins.com
\textsuperscript{4} Email: evw@cs.umn.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
language. These dialects have evolved from a simple “kernel” language that has been fairly stable throughout the development of Lustre. For example, to better support safety-critical software development, activation conditions (condact) and initialized delay (fby) constructs were added to the variant of the language used by the SCADE toolset [11], and a richer type system and modularity constructs have been proposed in Lustre v6. Other examples include with expressions and array slicing and composition operators in Lustre v4, case, TO and FROM expressions and support for generic types in the SCADE textual syntax, and different packages for statecharts-like extensions to the language [8,20]. In recent work [12], we have extended Lustre with condition tables like those found in RSML−e [25], state variables for building simple state machines, and a notion of events.

There are many more domain-specific features that would make Lustre easier to use in new domains. For example, Lustre has been used for the analysis [15] and code generation [5,6] of GALS architectures. Our interests here are in using Lustre to specify and analyze (but not generate code from) the behavior of GALS architectures. Previous explorations of this idea, such as [15], assume that users manually construct a scheduler node and use it to manage the clocks of all of the asynchronous processes in the model. However, a scheduler could be automatically derived using a language extension, given the rates and drift of the asynchronous processes in the model. To support this process, we add to Lustre a timeout condact construct that defines the behavior of an asynchronous process within the architecture as follows:

\[
a, b = \text{timeout\_condact}(\text{rate}, \text{min\_drift}, \text{max\_drift}, \text{channel}(x, y), \text{init\_a}, \text{init\_b});
\]

This construct (defined in Section 2) specifies that node channel representing a periodic process within the architecture is to be executed every \text{rate} milliseconds subject to clock drift in the range \text{min\_drift}..\text{max\_drift}. Like a condact expression, if the node does not evaluate, then the result of the expression is the value from the most recent evaluation, and before the first evaluation, the values \text{init\_a} and \text{init\_b} are used. Using this construct, a scheduler (implemented in the kernel Lustre language) can be automatically derived.

Extending a language using traditional techniques often requires a large development and tooling effort. Thus, there has been much research in programming languages communities on the development of techniques and tools for implementing languages that reduce the costs associated with adding new features to languages. There are (at least) two important criteria for extensions to a language. First, the new language constructs should have the same “look and feel” as the host language constructs. That is, they should support the same type of error-checking, optimization, and translations as do the host language constructs. Second, it should be possible to combine implementations of different extensions to the same host language to create a new language which incorporates the constructs in both. Furthermore, such a composition should require little or no implementation-level knowledge of the language.
extensions. When this second criteria (referred to as the “composability criteria”) is not met, users may be forced to chose between incompatible dialects of Lustre that individually have only some of the desired language constructs.

In previous work [28], we raised this issue of incompatible dialects and the traditionally high cost of language development. We proposed an extensible language framework for Lustre based on attribute grammars as a possible alternative approach to language development that satisfies the two criteria mentioned above. This approach is used to implement timeout automata as language constructs in Lustre. The primary contributions of this paper are the specification and implementation of timeout automata as first class language constructs in Lustre. Section 2 describes the GALS approach to development and defines the timeout automata construct. Section 3 describes some aspects of the implementation of the timeout automata as a language extension in our extensible languages approach. Section 4 discusses related work and concludes.

2 Timeout Automata and GALS architectures

2.1 GALS and Flight Guidance Synchronization Example

To illustrate our approach to the analysis of GALS architectures, we describe the synchronization logic in a Flight Guidance System (FGS). The FGS compares the measured state of the aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. The FGS subsystem accepts input about the aircraft’s state from several other subsystems and computes the pitch and roll guidance commands provided to the autopilot.

The FGS system has two physical sides corresponding to the left and right sides of the aircraft. These provide redundant implementations that communicate over a cross-channel bus. Normally, only one FGS instance (the pilot flying side) is active, with the other FGS instance operating as a silent, hot spare. A transfer switch button on the flight control panel (FCP) can be used to toggle the pilot flying side. In some critical flight modes, both sides are active and independently generate guidance values for the autopilot, so that the autopilot can verify that they agree within a predefined tolerance value.\footnote{A more detailed description of the FGS can be found in [21].}

To make the example of this paper tractable, we restrict ourselves to a simplified specification that deals only with the logic determining whether an FGS instance is active. This example captures critical functionality for the FGS, e.g., at least one side is active at all times, and illustrates some of the communication and coordination problems that can occur in GALS systems. In our analysis [21], we prove that this simplified model simulates the behavior of the full FGS w.r.t. synchronization, thereby ensuring that the results proven about the simplified specification also apply to the full specification.
Fig. 1. Two FGS Synchronization Architecture in Simulink

A graphical model of the system architecture is shown in Figure 1. The system inputs are:

- the Transfer Switch input (1), which switches the pilot flying side,
- the Independent Mode inputs (2, 3), which are Boolean signals that determine whether each side believes it is in the independent mode of operation,\(^6\) and
- the Clock inputs (4 – 7). Clocks are Boolean signals that enable the execution of the processes within the architecture. Treating the clock signals as unconstrained inputs allows us to model GALS systems within a synchronous paradigm [15]. By embedding this model inside a model that constrains the clocks, we can model a variety of different physical architectures and reason about their behavior.

The system outputs are:

- the Active outputs, which are Boolean signals that describe whether each side believes itself to be active, i.e., computing pitch and roll commands for the autopilot, and
- the Pilot_Flying outputs, which are Boolean signals that describe whether each side believes itself to be the pilot flying side.

2.2 Timeout Automata

A timeout automaton is a mechanism for constraining the Boolean clocks of processes to match a notion of real (calendar) time. As described in [9], the

\(^6\) As discussed in [21], in an actual system these are not inputs to the FGS but are instead computed. However, the system synchronization properties do not depend on the details of this computation.
automata consists of a set of processes, each of which run at a certain rate. A scheduler (also called an event list) stores the times at which each of the processes will next execute. Evaluation of the system consists of advancing time to the next instant in which a process (or processes) can execute. Given a set of processes $P$, we assume that each process $p \in P$ has an associated rate $r_p$, a time until next execution $t_p$, and a Boolean clock signal $c_p$, and that there is a distinguished variable $ci$ that records the increment of time since the last instant. Then, given a state $\sigma$ mapping identifiers to values, we generate a new state $\sigma'$ as follows:

- $\sigma'(ci)$ equals $\min(\sigma(t_p))$ where $p \in P$
- $\sigma'(c_p)$ is true iff $\sigma'(ci) - \sigma(t_p) = 0$
- $\sigma'(t_p)$ equals $r_p$ if $\sigma'(c_p)$; otherwise $\sigma'(ci) - \sigma(t_p)$.

Each process “fires” (executes) when its clock signal $c_p$ is true. Time always advances by some positive increment described by $ci$. If multiple processes share the same value for $t_p$ and $t_p = ci$, then they execute simultaneously within the step. Clock drift between processes can be introduced to the model by allowing the rate $r_p$ of each process to vary within some specified range.

In [9,3,4] this approach has been shown to be amenable to model-checking using SMT-based solvers for interesting GALS problems. We have also used it for system simulation and testing. The primary advantage of timeout automata for analysis that maximal time progress is made on each step (i.e., there are no “stuttering” steps in which the clock ticks but no other changes occur), and each step consumes a varying amount of real time as described by the clock increment $ci$.

In the Dual FGS example, we have used timeout automata to prove the correctness of the synchronization logic between the two FGSs. Properties proved include: (1) at least one FGS is always Active, and (2) at most one FGS is the Pilot Flying side. Other properties of interest are described in [21]. The proofs follow the process described in [9].

2.3 Implementation of Timeout Automata In Lustre

Timeout automata can be described as an extension to Lustre with the addition of a new expression construct:

```
timeout_conduct(rate, min_drift, max_drift, \langle node \rangle, \langle init_vals \rangle);
```

This construct specifies that node node representing a periodic process within the architecture is to be executed every rate milliseconds subject to clock drift in the range min_drift..max_drift. Like a conduct expression, if the node does not evaluate, then the result of the expression is the value from the most recent evaluation, and before the first evaluation of the node, the initial values init_vals are used. It is assumed (and checked by the compiler) that timeout_conduct expressions are not nested within other clocked expressions; this matches the expectation within GALS systems in that the asynchrony
occurs at the global level and synchronous clocking mechanisms are local to
one of the modeled processes.

In the Dual FGS model, the left and right FGSs run every 100 ms with a
+/- 1 ms drift and communications between the two FGSs requires 15 to 25
ms. The expression of this architecture in Lustre is shown in Figure 2.

```luster
type fgs_data = ... ; /* contains PF, Independent, and Ack data */
const lft_fgs_init = ... ; lr_init = ... ;
rht_fgs_init = ... ; rl_init = ... ;

node fgs ( other_fgs_in: fgs_data, ind_mode: bool,
    transfer_switch: bool, init_pilot_flying: bool)
    returns ( fgs_out: fgs_data ) ;
let ... tel ;

node channel ( channel_in: fgs_data)
    returns ( channel_out: fgs_data ) ;
let ... tel ;

node main ( trans: bool, lft_ind_mode: bool, rht_ind_mode: bool )
    returns ( lft_fgs_pilot: bool, lft_fgs_active: bool,
        rht_fgs_pilot: bool, rht_fgs_active: bool ) ;
var
    lft_fgs_out: fgs_data ; lr_chan_out: fgs_data ;
    rht_fgs_out: fgs_data ; rl_chan_out: fgs_data ;
let
(1) lft_fgs_out = timeout_conduct(100.0, -1.0, 1.0,
    fgs(lr_chan_out, trans, lft_ind_mode , true),
    lft_fgs_init) ;
(2) lr_chan_out = timeout_conduct(20.0, -5.0, 5.0,
    channel(lft_fgs_out),lr_init);
(3) rht_fgs_out = timeout_conduct(100.0, -1.0, 1.0,
    fgs(lr_chan_out, trans, rht_ind_mode , false),
    rht_fgs_init) ;
(4) rl_chan_out = timeout_conduct(20.0, -5.0, 5.0,
    channel(rht_fgs_out),rl_init);
tel;
```

Fig. 2. FGS Synchronization Architecture using timeout_conduct.

The semantics of the timeout_conduct specifications match the formaliza-
tion in Section 2.2. Each timeout_conduct expression becomes a process in the
timeout automata model, and a global scheduler is synthesized in Lustre from
the set of these processes. As an example, consider Figure 3, the automatically
generated implementation in Lustre for the FGS timeout_conduct in Figure 2.

The timeout node (line 8 of Figure 3) is used to define the $r_p$, $c_p$, and $t_p$
variables for a process within the model. The `rate` and `drift` inputs set $r_p$,
the `init_time` input sets the initial value of $t_p$, and the `time_decrement` input
corresponds to the global time decrement between steps $ci$. The timeout node
contains an individual count-down timer `time_remaining` that corresponds to
$t_p$, and generates a Boolean signal `fired` that corresponds to $c_p$. 

6
The expansion of the timeout conducts in `main` creates instances of the timeout node for each process and define constraints that describe the legal values for timers and drift inputs within the model. Line (3) in Figure 3 is the translation of the first timeout conduct in Figure 2 to its implementation as a kernel language conduct construct. The component node call to `fgs` and the initial values are the same; but the rate and drift parameters have been replaced by a clock variable (corresponding to \( c_p \) in the formal model) named `fired_1`. This variable is set on line (6) by a call to the `timeout` node that implements the time keeping operations of the timeout conducts.

On Figure 3 line (7), the model then selects the smallest time-remaining as the amount to advance each component clock (`time_decrement`) and feeds that value back to each individual component timer for use in computing the next clock tick. The definition of the node `min` is not shown but is what one would expect. Since the `time_decrement` value specifies the elapsed global time since the last clock tick, it is also output from the `main` node to allow a model checker to check properties involving global time (for example, the maximum time that some property \( P \) can be false is less than some time \( t \)).

Assert statements are also generated to restrict the new input drift values to be within the originally specified ranges of possible clock drift specified in the original timeout conduct constructs. For the first timeout conduct, the generated `assert` statements are shown in Figure 3 lines (4) and (5). Additional input parameters for the unconstrained input drift values are also added to the interface of `main`. The translation also adds new local variables in the line following the label (1).

The translation of the timeout conduct constructs involves more that just local transformations that are possible with macro processing. The translation needs to generate new equations for each timeout conduct and for defining `time_decrement` based on a global analysis that determines how many timeout conducts were used in the original code and what the generated time-remaining variables for each one are. Note that the original type declarations for `fgs_data`, the four constant `init` values and the declarations of the `fgs` and `channel` nodes are not changed in the translation and appear in the translated code as they did in the original. Thus, they are not repeated in Figure 3.

As there are only four processes in this model, the automata is relatively simple. However, with larger number of processes, it can become unwieldy. It is “boilerplate” code that must be re-written for each GALS system to be analyzed. Also, it is cumbersome to experiment with different architectural configurations (e.g., changing the rates and drift) in the translated model. We wish to encourage this kind of experimentation and formal analysis in the early stages of system design. Finally, there are hints that can be provided to aid analysis based on the structure of the automata (for example, the minimum and maximum possible values that the system clock can advance within a step). To make it easy to analyze these kinds of models, we would prefer to add a language construct to automatically construct the automata.
node main (drift_4: real, init_time_4: real, drift_3: real, init_time_3: real,
            drift_2: real, init_time_2: real, drift_1: real, init_time_1: real,
            trans: bool, lft_ind_mode: bool, rht_ind_mode: bool)
returns (time_decrement: real,
            lft_fgs_pilot: bool, lft_fgs_active: bool,
            rht_fgs_pilot: bool, rht_fgs_active: bool);

(1) var fired_4: real; time_remaining_4: real;
    fired_3: real; time_remaining_3: real;
    fired_2: real; time_remaining_2: real;
    fired_1: real; time_remaining_1: real;

(2) lft_fgs_out: fgs_data; lr_chan_out: fgs_data;
    rht_fgs_out: fgs_data; rl_chan_out: fgs_data;
let
(3) lft_fgs_out = condact(fired_1, fgs(lr_chan_out, trans,lft_ind_mode, true),
                           lft_fgs_init);
    lr_chan_out = condact(fired_2, channel(lft_fgs_out), lr_init);
    rht_fgs_out = condact(fired_3, fgs(lr_chan_out, trans,rht_ind_mode, false),
                           rht_fgs_init);
    rl_chan_out = condact(fired_4, channel(rht_fgs_out), rl_init);

(4) assert(((drift_1 <= 1) && (drift_1 >= -1)));
(5) assert(((init_time_1 >= 0.0) && (init_time_1 <= (100.0 + 1))));
    assert(((drift_2 <= 5) && (drift_2 >= -5)));
    assert(((init_time_2 >= 0.0) && (init_time_2 <= (20.0 + 5))));
    assert(((drift_3 <= 1) && (drift_3 >= -1)));
    assert(((init_time_3 >= 0.0) && (init_time_3 <= (100.0 + 1))));
    assert(((drift_4 <= 5) && (drift_4 >= -5)));
    assert(((init_time_4 >= 0.0) && (init_time_4 <= (20.0 + 5))));

(6) fired_1, time_remaining_1 = timeout(100.0, drift_1, init_time_1,
                                       time_decrement);
    fired_2, time_remaining_2 = timeout(20.0, drift_2, init_time_2,
                                        time_decrement);
    fired_3, time_remaining_3 = timeout(100.0, drift_3, init_time_3,
                                        time_decrement);
    fired_4, time_remaining_4 = timeout(20.0, drift_4, init_time_4,
                                        time_decrement);

(7) time_decrement = min(time_remaining_4, min(time_remaining_3,
                                              min(time_remaining_2, time_remaining_1)));
    tel;

(8) node timeout (rate: real, drift: real, init_time: real,
                time_decrement: real)
returns (fired: real, time_remaining: real);
let
    time_remaining = init_time -> if fired then rate + drift
                    else pre(time_remaining) - pre(time_decrement);
    fired = (pre(time_remaining) <= pre(time_decrement));
tel;

Fig. 3. Translated FGS Timeout Automata Model
3 Timeout Automata as a Language Extension

Implementing the timeout automata described in Section 2 by translation to the kernel Lustre language does not, per se, pose any exceptionally difficult challenges. Any solution, including ours, will (i) add a timeout node like the one in Figure 3 to the specification, (ii) add the equations that call to the timeout node and calculate the time_decrement value, and (iii) replace all timeout_construct constructs with the appropriate construct constructs that use the new Boolean fired flag. The main challenges arise in satisfying the look-and-feel and composability criteria described in Section 1. We have built [12] an extensible language framework based on higher-order attribute grammars (AGs) [17, 30] and implemented an AG specification language called Silver [26] that supports the building of languages and extensions that satisfy these criteria. In this approach a host language and language extensions are implemented as individual Silver AG modules. The supporting tools allow the composition of these modules to define new extended languages with little or no implementation-level knowledge of the host or languages extensions [12]. In this section we give a brief overview of how the timeout_construct extension is constructed using this approach. Due to space constraints this is necessarily cursory and a number of simplifications and omissions have been made, but the full Silver specifications can be found at www.melt.cs.umn.edu.

3.1 Mini-Lustre as the Host Language

The specification for Mini-Lustre (a subset of Lustre) is written in Silver, a portion of which is shown in Figure 4. A Silver specification for a language consists of an unordered series of declarations that define its concrete and abstract syntax as well as rules which assign values to attributes associated with non-terminals in the abstract syntax tree (AST). Since concrete syntax is defined as expected for traditional parser and scanner generators we do not show those and only discuss abstract syntax. To define the (abstract) syntax, there are declarations for terminals, non-terminals (keyword nt), and productions (prod), following standard AG terminology [17]. Synthesized attributes (syn) propagate information up the abstract syntax tree; inherited attributes (inh) propagate information down the AST. Equations defining attribute values are used to specify the semantic analyses, such as type checking. 7

The first line in Figure 4 provides the name of this grammar. These are used in import statements to compose attribute grammar specifications to create the specification for an extended language. Next, are declarations for nonterminals. Synthesized attributes pp, errors, and ctrans of type String are declared; these attributes, respectively, define a node’s pretty-print or “unparsed” representation, the errors occurring on the node and its children, and its translation to C. The attribute typerep is used to represent the type

7 This is meant broadly and can include causality and initial-state-definedness checks.
grammar lustre;

nt Root, DclList, Dcl, VarDclList, VarDcl, Locals, EqList, Eq, IdList, Expr;

syn attr pp :: String occurs on Root, Dcl, Expr, VarDcl, ...

syn attr errors :: String occurs on Root, Dcl, Expr, ...

syn attr ctrans :: String occurs on Root, Dcl, Expr, ...

syn attr typerep :: TypeRep occurs on Expr, ExprList;

prod root r::Root ::= dl::DclList
{ r.pp = dl.pp; r.errors = dl.errors; r.ctrans = ... dl.ctran ...; }

prod dclListCons dl::DclList ::= d::Dcl dltail::DclList { ... }

prod dclListOne dl::DclList ::= d::Dcl { ... }

prod nodeDcl n::Dcl ::= name::Id inputs::VarDclList outputs::VarDclList
locals::VarDclList eqs::EqList
{ n.pp = "node " ++ name.lexeme ++ " (" ++ inputs.pp ++ ") " ++ "returns"
++ " (" ++ outputs.pp ++ ") " ++ "\n" ++ locals.pp ++ "\nlet\n"
++ eqs.pp ++ "\nlet\n";

n.errors = inputs.errors ++ outputs.errors ++ ... ; n.ctrans = ... ; }

prod varDcl vd::VarDcl ::= var::Id type::Type { ... }

prod equation eq::Eq ::= ids::IdList expr::Expr
{ eq.pp = ids.pp ++ " = " ++ expr.pp ++ ";\n";

eq.errors = ... ; /* ensure ids and expr have same type(s) */ }

prod idExpr e::Expr ::= id::Id
{ ...; e.ctrans=...; e.errors = ... ; /* ensure id is declared */ }

prod conductExpr e::Expr ::= f::Expr call::Expr init_vals::ExprList { ... }

Fig. 4. A portion of the Silver specification of Mini-Lustre.

of an expression or expression/id list. The occurs on clause specifies which
nonterminals an attribute decorates. We will elide other nonterminal and
attribution declarations as they can be inferred from the specification.

A Mini-Lustre program (represented by Root) is a series of declarations
(DclList). The nonterminal Root on the left hand side of production root is
named r ("::" reads as "has type"); the right hand side has a single DclList
nonterminal named dl. Equations defining the synthesized attributes of r are
listed in curly brackets. For example, the first equation defines the pp attribute
on r to be the value of pp on dl. A node, defined by nodeDcl, has a name
(name), a list of input parameter declarations (inputs:: VarDclList), a list
of output parameters (outputs), a list of local variable declarations (locals),
and a list of equations (eql:: EqList). The production varDcl binds iden-
tifier names to types. These bindings are stored in a symbol table that is
passed to the equations in eqs and used for type-checking the expressions and
equations following rules specified by the errors attributes. For example, the
production equation checks that the identifier id and expression expr have
the same type and generates an error message if they do not.

3.2 Timeout Automata as a Language Extension

To add the timeout_conduct construct to the extensible Lustre framework we
must write a Silver attribute grammar specification that will specify the con-
crete and abstract syntax of the new construct, perform error checking and other analyses on the timeout \_\_condact, specify its translation to a condact construct, and for each use of a timeout \_\_condact in a node add additional equations and parameters to that node. Further we must add the definitions for the timeout and min node to the Lustre specification. Using language features provided by Silver, all these tasks can be specified in a single grammar module, thus making this extension a stand-alone unit that can be optionally composed with other similarly-defined language extensions.

Fig. 5 shows the Silver production tmoConcepExpr that specifies the abstract syntax of the timeout \_\_condact construct. To maintain the native look-and-feel, the pp, errors, and typerep attributes are defined explicitly in this production. Explicitly defining errors ensures that type errors are detected and reported on the timeout \_\_condact, not its kernel Lustre translation. Though elided, the definition of errors checks that the types of values returned by the node call call match the types of the initial values init \_\_vals.

Although tmoConcepExpr explicitly defines some attributes, it does not do so for attributes such as ctrans (or attributes for translating to the input languages of different model checkers). These attributes are implicitly defined using forwarding \[27\] through translation to a condact (condactExpr) in the host language by using the forwards to clause. When a tmoConcepExpr node in the AST is queried for an attribute that is not explicitly defined by an attribute definition, it forwards that query to the forwards-to construct. The value defined there is returned as the value of that attribute for the timeout \_\_condact. Thus, the value of ctrans on a timeout \_\_condact is the value of the ctrans attribute on the generated (translated-to) condact construct. Therefore, all back-end tools only see the generated condact calls while Lustre programmers see the timeout \_\_condact calls they write.

In addition, the Silver specification assigns a unique integer identifier (attribute num) to each timeout \_\_condact call. The identifier for each call is used in generated local variable names such as fired \_\_1 and fired \_\_2 as seen in Figure 3. Furthermore, relevant information regarding this timeout \_\_condact call is gathered and propagated up the AST to the enclosing node declaration using the synthesized attribute tmoCallInfoList. This information is used for generating the added equations for variables such as time \_\_remaining \_\_1 and time \_\_decrement also seen in Figure 3. The additional attribute definitions of tmoCallInfoList on existing host language productions can be all specified in the grammar module of the timeout \_\_condact extension by using the Silver language feature aspect productions, and no changes to the host language specification need to be made [13]. Once the information of all timeout \_\_condact calls in a node is gathered to the level of node declaration (production nodeDecl), it is used to generate additional equations and parameters to be inserted into the node. This step is a global transformation that is simplified and modularly defined by using the Silver language feature collections. Its mechanism is not further elaborated here and interested readers
grammar timeout; import lustre;
prod tmoConductExpr e::Expr ::= rate::Expr min_drift::Expr max_drift::Expr
call::Expr init_vals::ExprList
{ e.pp = "timeout_conduct(" ++ rate.pp ++ ", " ++ min_drift.pp ++ ...;
e.typerep = call.typerep;
e.errors = ...; /* check that call and init_vals have the same type */
forwards to conductExpr(idExpr(mkTerminal(Id, "fired_" ++ toString(e.num))),
call, init_vals);
e.num = gen_unique_int ( ) ;
/* gathering information of timeout_conduct calls */
e.tmoCallInfoList = [tmoInfo(rate, min_drift, max_drift, e.num)]; }

Fig. 5. Silver specification for the timeout_conduct construct.

may refer to [13] for detailed explanations.

4 Conclusion

4.1 Discussion

In this paper we have defined a timeout conduct construct useful in specifying and analyzing GALS architectures. It has been implemented as a language extension in an extensible Lustre framework. Timeout automata is one of several approaches for modeling asynchrony within synchronous languages. It has been used successfully on several protocol examples (e.g. [9,3,4]) and allows a natural expression of interesting safety and bounded liveness properties over GALS architectures. However, in the simplistic translation described in this paper, it adds a significant amount of additional state into the model, which makes formal analysis more expensive. Abstractions of the possible real-time evolutions of the architecture, such as those described by [15] may yield more tractable analysis. The use of extensible languages opens up several possible directions for future research. First, we plan to investigate whether abstractions can be performed as part of the compilation step to “kernel” lustre. Second, we plan on investigating techniques for describing clock relations (such as in [15]) directly through language extensions.

Our initial efforts in extensible languages were in the domain of programming languages. We have built an extensible specification of Java 1.4 and specified a number of non-trivial language extensions [29]. One extension embeds the database query language SQL into Java so that queries can be written naturally and syntax and type errors in SQL queries can be detected at compile-time, instead of run-time, as is the case in library-based approaches.

4.2 Related Work

There have been many other efforts to extend Lustre with new language features. Many of these features can also be implemented by translation to the a kernel Lustre language. For example, recent work to add state machines to
Lustre [8] translates the state machine constructs into a kernel Lustre language and the addition of modules and generics proposed for Lustre v6.

Extensions for synthesizing Lustre logical clocks from Simulink models with “real-time” rates for blocks are proposed in [5,6]. This work is similar in that it moves from a notion of real-time to logical time. Unlike timeout automata, it imposes a fixed real-time value on the base rate of the model; this allows for code generation but makes it more difficult to analyze processes with non-harmonic periods or arbitrarily small amounts of process drift.

Embedded domain specific languages [16], higher-order extensions to Lustre [23], and reactive extensions to ML [19] can be used to build extensible language frameworks for synchronous languages [7]. But composition of language features typically requires some implementation level understanding of the language extension and thus various extensions cannot be as freely composed as in our approach [12].

More generally, several approaches have been described for extending languages with new features. Macros systems (lexical, syntactic, hygienic [18], etc) do allow new languages constructs to be specified but they lack an effective means for performing the static analysis used to, for example, generate domain specific error messages. Note that some modern macro systems (e.g. [1] however do a some limited facilities for error processing. Object-oriented frameworks, such as Polyglot [22], have also been proposed for building extensible languages, but they do not support the automatic composition of language extensions that is provided by the attribute grammar-based approach.

Modular language definition and extensibility has received a significant amount of attention from the AG community. Other attribute grammar approaches lack forwarding and the default definition of attributes that it provides - thus the reuse of language features specified as AG fragments is achieved only by writing attribute definitions that “glue” new fragments into the host language AG. However, a particularly interesting approach is the rewritable reference attribute grammars [10] in the JastAddII system. New constructs are translated to host language constructs by destructive rewrites on the syntax tree. Although forwarding is similar to rewriting, it is non-destructive; the original tree and the forwards-to tree exist simultaneously. This allows both the explicit and implicit (via forwarding) specification of semantics, a capability that we have found to be crucial in the highly modular language specifications required for extensible languages and composable language extensions. Some modularity is lost when the rewrites are destructive.

References


Gao, Whalen, Van Wyk


