Flexible and Extensible Notations for Modeling Languages[⋆]

Jimin Gao, Mats Heimdahl, and Eric Van Wyk

Department of Computer Science and Engineering
University of Minnesota

Abstract. In model-based development, a formal description of the software (the model) is the central artifact that drives other development activities. The availability of a modeling language well-suited for the system under development and appropriate tool support are of utmost importance to practitioners. Considering the diverse needs of different application domains, flexibility in the choice of modeling languages and tools may advance the industrial acceptance of formal methods.
We describe a flexible modeling language framework by which language and tool developers may better meet the special needs of various users groups without incurring prohibitive costs. The framework is based on a modular and extensible implementation of languages features using attribute grammars and *forwarding*. We show a prototype implementation of such a framework by extending the *host language* Mini-Lustre, an example synchronous data-flow language, with a collection of features such as state transitions, condition tables, and events. We also show how new languages can be created in this framework by *feature composition*.

1 Introduction

Model-based development is gaining interest from the software industry, especially in the domain of safety critical systems. The aims are cost reduction and quality improvement through early defect removal through model testing and formal analysis, and automated code generation. There are currently many commercial and research tools that attempt to provide these capabilities [1–5].

In previous work [6] we discussed several factors that hinder the widespread adoption of formal methods and model-based development in practice. We also formulated several conjectures related to this topic, one of which is related to the work presented in this report: *"no modeling language will be universally accepted, nor universally applicable."* If a notation is not liked by the intended users it will simply not be used; a multitude of (domain-specific) languages is needed. Current languages and tools infrastructures are inflexible and make language customization and tool integration difficult and costly. As illustration, consider the following two scenarios.

[⋆] Different aspects of this work are partially funded by NSF CAREER Award #0347860, NSF CCF Award #0429640, and the McKnight Foundation.

First, a development team for an air-transport flight guidance system needs a modeling language. These systems are typically periodical and take actions when certain (generally rather complex) conditions hold. Due to the complexity of the conditions, condition tables, such as those found in RSML^{-e} [5] and SCR [4] would be useful when reviewing the models with domain experts and regulators. The developer's engineers prefer to use a data-flow language specifically designed for control systems, for example, Lustre. Thus, a language that is basically Lustre extended with RSML^{-e}-style tables would be desirable.

Second, a development team for pre-launch checkout software for a launch vehicle seeks a modeling language appropriate for capturing the complex sequences of events that must occur before it is safe to launch the vehicle. The team finds a pure data-flow language like Lustre unsuitable for the task (they would prefer an explicit notion of states and events), but they like the analysis support available for Lustre (numerous model checkers and theorem provers). The team finds the RSML^{-e} syntax more suitable for the task at hand, but does not find a commercial tool supporting the creation nor analysis of RSML^{-e} models. They wonder if the Lustre toolset could be extended with RSML^{-e} features to easily leverage existing analysis and code generation capabilities.

Because customizing commercial toolsets and building tools from the ground up to provide expanded language and analysis support is generally infeasible or too costly we believe a different view of modeling languages and tools is needed. Instead of treating each modeling language as a fixed, monolithic entity, and implementing its tool support based on that view, we adopt the notion of extensible languages—language implementations that are intended and optimized for future (front-end and back-end) additions (and possibly modifications). In this view, the artifacts include a *host language* and a set of *language extensions* that define desired language features not found in the host language.

The *flexible modeling language framework* described in this paper is based on the general idea that language extensions may introduce new constructs to the host language, new semantic analyses that, for example, ensure that the constructs defined in an extension are used correctly, and new translations to different target languages. In this framework, a domain-specific language can be easily created through inclusion of language extensions in the host language. In our domain, synchronous languages, (*e.g.*, Safe State Machines [2], SCADE [2], and SCR [4], to name just a few) are prevalent and all share the same semantic foundations. We believe that most (if not all) features of these languages can be implemented as extensions to a host language in our flexible language framework. Based on our experiences in model-based development, we believe that Lustre [7] is a suitable host language. First, Lustre is expressive enough to capture a large class of interesting behaviors and it has a simple and well-defined semantics suitable for analysis [8]. Second, there are commercial tools for Lustre that interest industrial users [2]. Here we use a reduced version (Mini-Lustre) that espouses the key features of full Lustre, but omits some rather involved and, for the purpose of this paper, less important ones.

In this framework, language features such as RSML^{-e}-style tables, Statecharts like events, and designated state variables do not need to be implemented in the host language but can instead be implemented as a language extension allowing us to address the needs of the developers in our first scenario.

The host-language is designed to have specific tool support; here, Mini-Lustre comes with support for semantic analysis such as type checking as well as a translation to a general purpose programming language and a number of translations from Mini-Lustre to the input languages of a number of analysis tools. Thus, to address the second scenario we create language extensions that (1) add the syntax and semantic analysis of the desired new language, (2) specify the translation of the new language constructs to the host language Mini-Lustre, and (3) hide the undesirable (concrete) syntax of Mini-Lustre so that the new language primarily uses the host language as an intermediate representation to take advantage of the translations to various analysis tools. Note that new translations from the host language to new analysis engines will automatically work to languages created by extending the host language.

To be useful, extensible language frameworks need two crucial characteristics. First, language constructs implemented as language extensions must have the same "look and feel" as constructs in the host language. That is, at a minimum, they should perform some semantic analysis to report error messages at the extension level and not rely on their translation to the host language for error checking. Thus, traditional macros are not an acceptable means for implementing language constructs as error checking is done on the constructs to which the macros expand. Second, language extensions must be composable so that various language extensions, implemented independently, can be imported into a host language in a cost-effective way. For some language extensions, the composition can be entirely automatic. The language user may just select language extensions from a list and the framework automatically builds the specification for the new extended language. This composition is possible for the first scenario above in which the host language Lustre is extended with RSML^{-e}-style tables. In other cases, composition may require more involvement from someone skilled in language development; this is the case in our second scenario where Lustre is extended with state variables and the underlying Lustre constructs such as nodes and data-flow equations need to be hidden (at last syntactically). Both forms of composition are demonstrated in Section 3.5.

Our extensible view of modeling languages may be realized using attribute grammars with *forwarding* [9]. The host language is implemented by writing an attribute grammar specification of the language. We have developed an attribute grammar specification language, called Silver for this purpose. The Silver tools automatically generate an attribute grammar evaluator for languages specified in Silver. Language extensions, such as the addition of tables suggested above, are implemented as attribute grammar fragments. The combination of the host language attribute grammar and the selected attribute grammar fragments that implement language extensions provide a specification for the new languages as required in our sample scenarios. Again, the Silver tools provide an automatic implementation for these new languages. We illustrate the feasibility of this approach by implementing solutions to the problems posed in each scenario.

Section 2 presents an implementation of Mini-Lustre as a Silver attribute grammar specification. Section 3 shows how extensions to Lustre can be implemented as attribute grammar fragments and composed to create new (extended) languages. These exercises provide a view of the power of this approach. Section 4 discusses related and future work and concludes.

2 Mini-Lustre: the Host Language

Lustre is a synchronous data flow language designed for programming reactive systems as well as for describing hardware. Lustre is synchronous in that it provides temporal determinism by partitioning physical time into discrete time points, at which computations react instantaneously to external events. This high-level paradigm is specially designed for abstracting the actual computation away from the complex timing constraints involved with control systems. In addition, Lustre specifies its computations using a data flow model, which enables natural parallelism and tractable analysis.

Consider the example in Fig. 1. It specifies partial functionality of an Altitude Switch (ASW), an avionics system that turns the power on for another system when the aircraft descends below a threshold altitude and turns it off when the aircraft ascends above the threshold plus a hysteresis factor. Here we focus on the `AltStatus` variable used to keep track whether the aircraft should be considered above or below the threshold. The initial value of `AltStatus` is undefined (`Unknown ->`) and thereafter assigned by the nested if-expression. We assign `AltStatus` the value `Above` if the altitude readings are reliable (`AltQuality = Good`) and we are either (1) classifying `AltStatus` for the first time (`pre( AltStatus) = Unknown`) and we are above the threshold or (2) AltStatus has been established and we are above the threshold plus the hysteresis. `AltStatus` is `Below` if altitude readings are reliable and the altitude is less than or equal to the threshold. If the altitude readings are not reliable `AltStatus` is `Unknown`.

```
type Status = enum { Unknown, Above, Below } ;

node ASW (AltQuality:Quality, AltThres:int, Hyst:int, Altitude:int)
     returns (AltStatus:Status) ;
 let AltStatus = Unknown ->
      if AltQuality = Good and Altitude > AltThres and
         (pre(AltStatus) = Unknown or Altitude > AltThres + Hyst) then Above
      else if AltQuality = Good and (not Altitude > AltThres) then Below
      else if not AltQuality = Good then Unknown
      else pre(AltStatus) ;     tel;
```

Fig. 1. ASW in Mini-Lustre.

```
grammar lustre ;
nt Root, NodeList, Node, VarDeclList, VarDecl, Locals, EqList, Eq, Expr ;
syn attr pp     :: String occurs on Root, Node, Expr, VarDecl, ... ;
syn attr errors :: String occurs on Root, Node, Expr, ... ;
syn attr ctrans :: String occurs on Root, Node, Expr, ... ;

prod root r::Root ::= nl::NodeList
  { r.errors = nl.errors; r.pp = nl.pp; r.ctrans = ... nl.ctrans ...; }
prod nodeListCons nl::NodeList ::= n::Node nltail::NodeList { ... }
prod nodeListOne  nl::NodeList ::= n::Node  { ... }
prod node n::Node ::= name::Id inputs::VarDeclList outputs::VarDeclList
                      locals::VarDeclList eql::EqList
  { n.pp = "node " ++ name.lexeme ++ " (" ++ inputs.pp ++ ") " ++ ... ;
    n.errors = inputs.errors ++ outputs.errors ++ locals.errors ++ eql.errors ;
    n.ctrans = ... ;  }
prod varDecl vd::VarDecl ::= var::Id type::Type
  { vd.pp = var.lexeme ++ " : " ++ type.lexeme ;   }
prod equation eq::Eq ::= id::Id expr::Expr
  { eq.pp = id.lexeme ++ " = " ++ expr.pp ++ ";\n" ;
    eq.errors = ... ;  /* ensure id and expr have same type */ }
```

Fig. 2. A portion of the Silver specification of Mini-Lustre.

We provide the attribute grammar (AG) specification for Mini-Lustre, which contains the characteristic features of full Lustre, such as node declarations and synchronous computation. The specification is written in Silver and shown in Fig. 2 and Fig. 3. In general, a Silver specification for a language consists of a series of declarations that define its concrete and abstract syntax as well as rules which assign values to attributes associated with nonterminals. To define the syntax, there are declarations for terminals, nonterminals (keyword `nt`), and productions (`prod`). Productions marked as *concrete* are used to construct the parser. They are as expected and thus not shown in Fig. 2 or Fig. 3. The AG portion of the specification consists of declarations for attributes (`attr`), and production-associated equations that define the values of attributes that label nonterminal nodes in a program's abstract syntax tree (AST). An attribute is synthesized (`syn`) if it propagates information up the abstract syntax tree; it is inherited (`inh`) if it propagates information down the AST. Note that the order of Silver declarations does not matter; values can be used before their definition.

The first line of the specification in Fig. 2 provides the name of this grammar. Grammar names are used in following sections in which the Silver `import` statement is used to combine attribute grammar specifications to create the specification for an extended language. Next, the nonterminals in the grammar are declared. 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 `occurs on` attribution clause specifies which nonterminals an attribute decorates. We will elide other nonterminal and attribution (`occurs on`) declarations as they can be inferred from the specification.

```
prod idref expr::Expr ::= id::Id  { ... }
prod and expr::Expr ::= lft::Expr rht::Expr
  { expr.pp = ... ; expr.errors = ... ; expr.ctrans = ... ; }
prod not expr::Expr ::= n::Expr { ... }
prod or expr::Expr ::= lft::Expr rht::Expr
  { expr.pp = "(" ++ lft.pp ++ " || " ++ rht.pp ++ ")" ;
    expr.errors = ... ;  /* check both lft and rht are bool */
    forwards to not( and( not(lft), not(rht) ) ); }
```

Fig. 3. Silver specifications of Mini-Lustre expressions (`Expr`).

A Mini-Lustre program (represented by a nonterminal `Root`) is a series of node definitions (represented by `NodeList`). The nonterminal `Root` on the left hand side of the production `root` is named `r`; the right hand side has a single `NodeList` nonterminal named `nl`. Equations defining the synthesized attributes of `r` are listed in curly brackets. For example, the last equation uses ellipses (`...`) to indicate that the value of the `ctrans` attribute on `r` is computed from the value of `ctrans` on `nl`. A node, defined by production `node`, is composed of a name (`name`), a list of input parameter declarations (`inputs` of type `VarDeclList`), a list of output parameters (`outputs`), a list of local variable declarations (`locals`), and a list of equations (`eql` of type `EqList`). Its attributes are defined as expected. There are several list constructs in Mini-Lustre and we will not show the productions for many of these as they are what one would expect and can be inferred. They will follow the pattern of using a "cons" and "one" production like those defined for `NodeList` in Fig. 2. The production `varDecl` binds identifier names to types. These bindings are stored in a symbol table that is passed to the equations in `eql` as expected. These are not shown since this is a straight-forward and common task in attribute grammars. The production `equation` will check that the identifier `id` and expression `expr` have the same type and generate an error message if they do not. It also defines its `pp` attribute as expected. Further definitions of `pp` are also what one would expect and are thus elided, though each production does have an explicit definition for it.

Of special interest is the production `or` which defines the disjunction of two expressions. It uses an extension to attribute grammars called *forwarding* [9] that is used extensively in defining the extensions to Mini-Lustre in Section 3. To use forwarding a production defines a construct that it is semantically equivalent to. It will *forward* queries for attributes that it does not *explicitly* define with an attribute definition to this "forwards-to" construct. The forwards-to construct will return its value for the queried attribute. In this case of `or`, the production states the semantic equivalent of `or(lft,rht)` is `not( and( not(lft), not(rht)))`. When a construct created by the `or` production is queried for its `errors` or `pp` attributes, it returns the values specified by the explicit definitions. When queried for its `ctrans` attribute, it returns the value of `ctrans` on its semantically equivalent forwards-to construct. This is somewhat similar to macro expansion, where the forwards-to construct corresponds to the body of the macro. Unlike a macro definition of `or` the production with forwarding reports error messages on programmer-written specifications.

3 Mini-Lustre Extensions

In this section we define four language features that can be added to Mini-Lustre as modular language extensions. These features, RSML^{-e}-style tables, equals clauses, and state variables, and Statechart-like events, are all features that some but not all users of synchronous languages find useful. We also show how a simple "module" extension can hide host language syntax to in essence create a new language that is not an extension of the host. The goal of this section is to show in some detail how feature-rich modeling languages, tailored to specific domains or to user preferences, can be easily created by simply composing the host-language with the desired set of language features. Thus, the extensions, just like Mini-Lustre, capture the important characteristics of the features and not a full realization of them. This high-degree of modularity is achieved through the *forwarding* extension to attribute grammars.

3.1 Tables

```
b = table
    (c1 && c2) : T F ;
    ! (c2)     : T * ;
    (c3 || c2) : F T ;
   end;
```

Fig. 4. RSML^{-e} table.

Tables are used for specifying complicated boolean expressions, available in both RSML^{-e} and SCR. They have been shown to be useful when presenting specifications to domain experts, such as pilots and air traffic controllers [10–12]. An example of such a table is shown in Fig. 4 in which `b`, `c1`, `c2`, and `c3` are boolean variables. In each row of the table there are "truth value" entries `T` (true), `F` (false), or `*` (don't-care) indicating the desired truthfulness of the preceding Boolean expression, *e.g.* `c1 && c2` in the first row. A table is an alternative form of the Boolean expression that can be obtained by taking the conjunction of the expressions generated for each entry in a column and then taking the disjunction of these expressions generated for the columns. Therefore, the equation in Fig. 4 is semantically equivalent to the pure-Mini-Lustre code shown below:

```
b = ((c1&&c2) && !c2 && !(c3||c2)) || (!(c1&&c2) && true && (c3||c2));
```

The table extension is implemented as a Silver attribute grammar fragment, portions of which are shown in Fig. 5. This specification shows the abstract syntax productions and attribute definitions for error-checking and computing the pure-Mini-Lustre code to which the table construct translates (forwards to). A table is an alternative form of expression (nonterminal `Expr`) as defined by the production `table`. It consists of a number of rows (`ExprRowList`); each row (`ExprRow`) in turn consists of a (Boolean) expression and a list of truth-values (`TruthValueList`). Truth values (`TruthValue`) consist of the terminal `TrueTV` (marker `T`), `FalseTV` (marker `F`), or `Star` (marker `*`).

Several attributes are used to compute the pure-Mini-Lustre expression shown above that the table construct will forward to. The inherited attribute `rowexpr` is used to pass the Boolean expression in each row down to the truth values where a boolean expression in the host language is constructed (according to the truth value) and is passed up the AST in the attribute `texpr`. For example,

```
grammar lustre_tables ;
import  lustre ;

prod table t::Expr ::= erows::ExprRowList
{ t.errors = erows.errors ;
  forwards to disjunction(mapConjunction(transpose(erows.texprss ))) ; }

nt ExprRowList, ExprRow, TruthValueList, TruthValue ;

syn attr texprss :: [[Expr]] occurs on ExprRowList ;
syn attr texprs :: [Expr] occurs on ExprRow, TruthValueList ;
syn attr rlen :: Integer occurs on ExprRowList, ExprRow, TruthValueList ;
inh attr rowexpr :: Expr occurs on TruthValueList, TruthValue;
syn attr texpr   :: Expr occurs on TruthValue ;

prod exprRowCons erows::ExprRowList ::=
         erow::ExprRow erowstail::ExprRowList
{ erows.rlen = erow.rlen ;
  erows.errors = erow.errors ++ erowstail.errors ++
                   if erow.rlen == erowstail.rlen then "" else
                   "Error: rows need same num of cols";
  erows.texprss = cons(erow.texprs, erowstail.texprss ) ; }

prod exprRowOne erows::ExprRowList ::= erow::ExprRow
{ erows.errors = erow.errors ; erows.rlen = erow.rlen ;
  erows.texprss = [erow.texprs] ;   }

prod exprRow erow::ExprRow ::= e::Expr tvl::TruthValueList
{ erow.rlen = tvl.rlen ; erow.texprs = tvl.texprs ; tvl.rowexpr = e ;  }

func disjunction Expr ::= es::[Expr]
{ return if leng(es) == 1 then head(es)
    else or(head(es), disjunction(tail(es))) ; }

prod tvlistCons tvl::TruthValueList ::=
     tv::TruthValue tvltail::TruthValueList
{ tvl.rlen = 1 + tvltail.rlen ;
  tvl.texprs = cons (tv.texpr, tvltail.texprs );
  tv.rowexpr = tvl.rowexpr ; tvltail.rowexpr = tvl.rowexpr ;  }

prod tvlistOne tvl::TruthValueList ::= tv::TruthValue
{ tvl.rlen = 1; tvl.texprs = [tv.texpr]; tv.rowexpr = tvl.rowexpr ; }

prod tvTrue tv::TruthValue ::= t::TrueTV { tv.texpr = tv.rowexpr ; }
prod tvFalse tv::TruthValue ::= f::FalseTV { tv.texpr = not(tv.rowexpr) ; }
prod tvStar  tv::TruthValue ::= s::Star { tv.texpr = true() ; }
```

Fig. 5. Silver table extension specification.

for the first row of table in Fig. 4, the `texpr` attributes for the `T` and `F` markers have values of `(c1 && c2)` and `! (c1 && c2)`, respectively. A `true` constant is always created for a `*` entry. The synthesized attributes `texpr`, `texprs`, and `texprss` collect these Boolean expressions into a list of lists of `Exprs` that is passed up to the top `ExprRowList` node of the complete table. Both list types and list expressions are denoted using square brackets (`[ ]`). After a transposition of the list, the pure-Mini-Lustre translated table is formed. This construct is the forwards-to construct of the production `table`. Several utility functions, `transpose`, `disjunction`, `conjunction`, and `mapConjunction`, are defined for building the translated table. Only `disjunction` is shown, but the others are similar.

The other critical function computed by the attributes is to perform error checking. We need to check that all rows in the table have the same number of columns. This is a semantic analysis that *must* be performed on the extension constructs, as an incorrect number of columns in a row will *not* be detected on the translation to pure-Mini-Lustre. To accomplish this, that `table` production explicitly defines its `errors` attribute to be the errors reported on its child `erows`. To detect such errors, a row length attribute, `rlen`, is computed on truth value lists and expression row lists and compared on the `exprRowCons` production to detect any rows whose length differs from other rows. This analysis highlights the critical role played by forwarding. It allows us to define some attributes, such as `ctrans` *implicitly* via the translation to the host language and to define other attributes, such as `errors`, *explicitly* on the extension constructs. This ensures that semantic analyses can be carried out at the right level of abstraction.

The `lustre_tables` grammar specification provides a definition of Lustre extended with the table construct since it imports the grammar `lustre`. The Silver tools can take this specification and build an attribute evaluator that performs error checking and translations to C of the extended Mini-Lustre + tables language. In Section 3.5, we will see how several independently developed extensions such as those described in the following sections can be combined in creating an extended language.

```
grammar lustre_equals ;  import lustre ;
nt Cases ;  syn attr ifexpr :: Expr occurs on Cases;
prod equals e::Expr ::= cs::Cases
  { e.errors = cs.errors ++ ...; /* ensure e and vals in cs have same type */
    forwards to cs.ifexpr ; }
prod casesCons cs::Cases ::= val::Expr cond::Expr cs1::Cases
  { cs.errors = ... ; /* ensure cond is boolean type */
    cs.ifexpr = ifthenelse(cond, val, cs1.ifexpr) ; }
prod casesOtherwise cs::Cases ::= val::Expr
  { cs.errors = ... ; cs.ifexpr = val ; }
```

Fig. 6. The Silver specification of the equals clause extension.

3.2 Equals Clauses

```
b = equals e1 if (c1 && c2)
    equals e2 if ! c1
    equals e3 if ! c2
    otherwise pre(b);
```

Fig. 7. Equals clauses

Specification in the state machine transition style is a popular approach in many domains and is the basic paradigm of languages such as Statechart, SCR, and RSML^{-e}. The *equals* clause construct implemented here as a language extension is one way to describe the transition choices of a state machine. An example of the equals clause is shown in Fig. 7, where `c1`, `c2`, and `c3` are Boolean variables, and `e1`, `e2`, and `e3` are expressions of the same type as variable `b`. This equals clause is evaluated as follows: if the condition `(c1 && c2)` evaluates to `true`, the value of variable `b` is taken to be that of `e1`; otherwise if `c1` is `false`, the value of `b` is `e2`; otherwise if `c2` is false, the value of `b` is `e3`; and if none of the condition holds, `b` retains its original value (`pre(b)`). This equals clause can be translated to the pure-Mini-Lustre nested if-then-else expression shown below.

```
b = if (c1 && c2) then e1 else if (!c1) then e2
    else if (!c2) then e3 else pre(b);
```

Part of the AG specification of the equals clause extension is shown in Fig. 6. The complete equals clause (`Cases`) is defined as an expression (`Expr`) in production `equals`. The `ifexpr` attribute on nonterminal `Cases` holds the equivalent if-then-else expression, which is constructed in production `caseCons`. It is used as the forwards-to construct in the `equals` production. The `errors` attribute is defined explicitly, as in the table extension.

3.3 State Variables

Like the equals clause extension, state variables, representing communicating state machines, are an important element in the state transition style of software specifications. Here we show how the RSML^{-e} state variable construct that captures this notion can be implemented as a language extension built not only on the host language Mini-Lustre but also on the two previous extensions `lustre_tables` and `lustre_equals`.

Fig. 8 shows the same ASW specification from Fig. 1 rewritten with extended Mini-Lustre, complete with tables, equals clauses, and state variables extensions. The meaning of the state variable declaration is easy to infer from this example. What is different from the original equals clauses is that the *otherwise* clause is implied here. Although the mixture of Lustre node and state variable declarations may seem strange—for example, the variable `AltStatus` with its type is declared twice—an additional extension that defines *modules*, which forwards to the node declaration, can fix the syntax and then provide a complete package of features for the descriptions of state machine models.

Part of the attribute grammar specification of the state variable extension is shown in Fig. 9. The state variable production `stateVar` forwards to the semantically equivalent Mini-Lustre equation that uses the same `ifexpr` attribute defined on `Cases` used in the previous example. The inherited attribute

```
node ASW (AltQuality:Quality, AltThres:int, Hyst:int, Altitude:int)
        returns (AltStatus:Status) ;
let  state variable AltStatus : Status
       initial value : Unknown
       equals Above if table pre(AltStatus) = Unknown   : T * ;
                             AltQuality = Good          : T T ;
                             Altitude > AltThres        : T T ;
                             Altitude > AltThres + Hyst : * T ;
                       end table
       equals Below if table AltQuality = Good           : T ;
                             Altitude > AltThres         : F ;
                        end table
       equals Unknown if AltQuality = Good
     end state variable      tel;
```

Fig. 8. ASW in Mini-Lustre extended with state variables, equals clauses and tables.

defaultExpr passes the expression to be used in the equation if none of the conditions in the equals clauses are true. An aspect production is used to add new attribute rules for attributes inStateVar and defaultExpr to productions casesCons and equals imported from the grammar lustre_equals. The inherited attribute inStateVar is true on the Cases enclosed in a stateVar and false otherwise. This is used on the casesOne production to raise an error if it, instead of the casesOtherwise production, is used in the original equals clause which requires an otherwise.

3.4 Events

Events are an extension to Mini-Lustre quite different from the previous ones. Below is a fragment of an enhanced definition of AltStatus from Fig 1 written using Mini-Lustre extended with events. The complex conditions from the original definition are abbreviated as C1, C2, and C3 here.

```
event AltClassEvt, AltLostEvt;
AltStatus = Unknown ->
         if catch(AltRcvEvt,C1) then throw(AltClassEvt,Above)
    else if catch(AltRcvEvt,C2) then throw(AltClassEvt,Below)
    else if catch(AltRcvEvt,C3) then throw(AltLostEvt,Unknown)    ...
```

A new declaration construct event is added to the language and used in the declaration of two events. Here, an *altitude classified* event is thrown if AltStatus is defined to be either Above or Below. If not, the AltLostEvt event is thrown. In the assignment equation for AltStatus, two new constructs, throw and catch, are used for the generation and consumption of events. The evaluation of throw(evt,e) produces the value e, and causes event evt to be generated in the next time step. An event remains active for only a single step and catch(evt,e) returns true if the event evt is active at the current step and e evaluates to true. Therefore throw is an expression with side-effects, clearly a conceptual departure from the data flow model of Lustre. The event/action

```
grammar lustre_statevar ;
import lustre, lustre_tables, lustre_equals ;
inh attr inStateVar  :: Boolean occurs on Cases ;
inh attr defaultExpr :: Expr occurs on Cases ;

prod stateVar eq::Equation ::= id::Id type::Type init::Expr cs::Cases
{ cs.defaultExpr = pre(idref(id)) ; cs.inStateVar = true ;
  cs.errors = cs.errors ++ ... ; /* ensure init has correct type */
  forwards to equation(id, follow(init, cs.ifexpr)); }
prod casesOne cs::Cases ::= val::Expr cond::Expr
 { cs.errors = if cs.inStateVar then "" else "Error: Missing OTHERWISE clause";
   cs.ifexpr = ifthenelse(cond, val, cs.defaultExpr) ; }
aspect prod equals e::Expr ::= cs::Cases { cs.inStateVar = false; }
aspect prod casesCons cs::Cases ::= val::Expr cond::Expr cs1::Cases
 { cs1.defaultExpr = cs.defaultExpr ; cs1.inStateVar = cs.inStateVar ; }
```

Fig. 9. The Silver specification of the state variable extension.

specification style can simplify some specifications and is an important feature in languages like Statechart.

The specification using events above can be translated to pure-Mini-Lustre in which events are translated into Boolean variables. A `throw` forwards to its second argument and a `catch` forwards to the conjunction of the Boolean variable of the named event and its second argument. An equation for each new Boolean variable is also generated by combining the conditions of the if-then-else constructs that enclose all `throw` constructs of the corresponding event. Below is the equation generated for `AltClassEvt`.

```
  AltClassEvt = false -> (pre(AltRcvEvt) && pre(C1)) ||
      (!(pre(AltRcvEvt) && pre(C1)) && (pre(AltRcvEvt) && pre(C2)) ;
```

The attribute grammar specification to create the above equation is not trivial but is more verbose than it is complex. Essentially an inherited attribute of type `Expr` is used to pass down the conditions of enclosing if-then-else constructs to each `throw` construct. A synthesized attribute is used to compute the disjunction of these expressions. Thus, the two `throw` constructs in the example correspond to the disjunction of two expressions in its translation. The generation of these equations requires a global transformation beyond the capabilities of macro-based approaches. Due to space constraints the Silver implementation of events is not shown.

3.5 Scenario Implementations

Silver has a flexible module system based on the `grammar` declarations seen in the specifications above. Silver `import` statements can be used to easily compose new and extended languages from these named grammars.

For Scenario 1, the desired language is created by composing the host language module `lustre` and language extension modules `lustre_tables` and `lustre_equals` by the Silver specification in Fig. 10. The Silver tools read this to build the attribute grammar for the specified language from the imported host and extensions. The `import ... including syntax` statement performs two functions. First it imports all the definitions of the attribute grammar constructs (productions, attributes, etc.) from the named module. These are used in the attribute grammar evaluation phase to perform error checking and translations as specified by the attributes. Second, the `including syntax` clause also add the concrete syntax specifications defined in the imported module. The specifications of the concrete syntax are given to a parser and scanner generator to build the parser and scanner for the extended language. In the specifications above we have not shown these as they are done in a traditional manner.

Many language extensions, including the tables, equals, state variable, and event extensions presented here are such that they can be automatically composed with other extensions to create, for example, the `scenario1` language above. This means that no attribute grammar "glue" code needs to be written to compose the host language and the language extensions. In this case, the Silver specification above can be automatically generated from the list of extensions selected by the user.

```
grammar scenario1 ;
import lustre
 including syntax ;
import lustre_tables
 including syntax ;
import lustre_equals
 including syntax ;
```

Fig. 10. Scenario 1.

For the second scenario, we create a new language that does not use the node construct of Mini-Lustre but replaces it with a simple module system for collecting state variables. This is similar to RSML^{-e} but is a smaller language meant to only demonstrate how new languages can be created in the framework. This is easily accomplished by a Silver specification that imports the `lustre`, `lustre_statevar`, `lustre_tables`, and `lustre_equals` modules, but uses a `syntax hiding` clause to block the importation of the concrete productions for nodes and equations from the host specification `lustre`. This specification also defines concrete syntax for a module construct that consists of a sequence of state variables. The abstract production for this module construct forwards to the expected translation in the host language Mini-Lustre. Space limitations prevent showing this specification, but the key point is that a new language is defined by hiding aspects of the host language and replacing them with the desired new ones.

4 Discussion

4.1 Related Work

Tools and techniques for language extensibility and modularity have been studied extensively in the area of programming languages and thus the description here is necessarily cursory. In the framework we have described, language extensions can define new language constructs, new semantic analyses on the extension-defined and host language-defined constructs, and translations to new target languages. There are existing tools and techniques that support each of these types of language extension, but no single approach supports them all. Closely related are various macro approaches, such as syntactic, hygienic, and programmable [13] macro systems. These allow new constructs to be defined but do not support semantic analysis of the new constructs.

There has also been a considerable amount of work on language modularity, *e.g.*, [14,15], from the perspective of attribute grammars. Higher-order functions as attributes provide the inspiration for some in seeking modular specifications, *e.g.*, [16], while object-oriented concepts of inheritance and objects motivate others, *e.g.*, [17]. Silver builds on much of this work and incorporates, for example, higher-order attributes [18]. Also of interest are Hedin's re-writable reference attribute grammars [19] in which a mechanism for rewriting the abstract syntax tree based on rewrite-rules is used. There, attributes are only retrieved from the rewritten tree; this differs from forwarding, which allows attributes to be retrieved from the original tree and the forwarded-to tree. This is critical for extensions like tables where we must do error checking on the original tree but want to get attributes for translations to target languages from the forwarded-to tree. Microsoft's Intentional Programming system (IP) [20] is the most closely related system to the extensible language framework used in this paper.

4.2 Conclusion

Silver was developed for building extensible languages based on attribute grammars with forwarding. It is a full-featured attribute grammar specification language with higher-order attributes [18], forwarding [9], a module system, polymorphic lists, and pattern-matching; it is freely available on the internet at `www.melt.cs.umn.edu`. We have used it to build an extensible versions of Java 1.4 called the Java Language Extender [21] and several modular extensions. One embeds the domain-specific language SQL into Java for static syntax and type checking of SQL queries; another adds general-purpose features such as algebraic data-types and pattern matching.

For users of synchronous languages, we can provide a flexible modeling language framework that allows a rich variety of modeling language features to be used. In the work presented here, we have showed how constructs such as condition tables, state variables, and events can be easily added to a host modeling language in a modular way. Note that these are not just lightweight syntactic extensions that do some error checking. Both the tables and the events extensions do a considerable amount of code transformation and manipulation to generate the host language constructs that they translate to via forwarding. The analysis and manipulation rely heavily on the expressive nature of attribute grammars.

We are currently building a much more complete implementation of Lustre and RSML^{-e} in this framework and exploring the feasibility of building higher-level abstractions as language extensions. It is our belief that a well-developed extensible language framework can be built that allows researchers and practitioners to more freely explore the wide range of possible language features that will help to more effectively specify software systems and ultimately make formal methods more appealing to a wider audience.

References

1. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Comp. Prog. **19**(2) (1992) 87–152
2. Esterel-Technologies: Corporate web page. www.esterel-technologies.com (2004)
3. Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A., Trakhtenbrot, M.: Statemate: A working environment for the development of complex reactive systems. IEEE Trans. on Soft. Engin. **16**(4) (1990)
4. Heitmeyer, C., Bull, A., Gasarch, C., Labaw, B.: SCR*: A toolset for specifying and analyzing requirements. In: Proceedings of the Tenth Annual Conference on Computer Assurance, COMPASS 95. (1995)
5. Thompson, J.M., Heimdahl, M.P., Miller, S.P.: Specification based prototyping for embedded systems. In: Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering. Volume 1687 of LNCS. (1999)
6. Van Wyk, E., Heimdahl, M.: Flexibility in modeling languages and tools: A call to arms. In: Proc. of IEEE ISoLA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation. (2005)
7. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language Lustre. Proc. of the IEEE **79**(9) (1991) 1305–1320
8. Halbwachs, N., Lagnier, F., Ratel, C.: Programming and verifying real-time systems by means of the synchronous data-flow language lustre. IEEE Transactions on Software Engineering (1992) 785–793
9. Van Wyk, E., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in attribute grammars for modular language design. In: Proc. 11th Intl. Conf. on Compiler Construction. Volume 2304 of LNCS. (2002) 128–142
10. Heninger, K.: Specifying software requirements for complex systems: New techniques and their application. IEEE Trans. on Software Engin. **6**(1) (1980) 2–13
11. Leveson, N., Heimdahl, M., Hildreth, H., Reese, J.: Requirements Specification for Process-Control Systems. IEEE Trans. on Software Engin. **20**(9) (1994) 684–706
12. Zimmerman, M.K., Lundqvist, K., Leveson, N.: Investigating the readability of state-based formal requirements specification languages. In: Proc. 24th Intl. Conf. on Software Engineering, ACM Press (2002) 33 – 43
13. Weise, D., Crew, R.: Programmable syntax macros. ACM SIGPLAN Notices **28**(6) (1993)
14. Ganzinger, H.: Increasing modularity and language-independency in automatically generated compilers. Science of Computer Programing **3**(3) (1983) 223–278
15. Kastens, U., Waite, W.M.: Modularity and reusability in attribute grammars. Acta Informatica **31** (1994) 601–627
16. Le Bellec, C., Jourdan, M., Parigot, D., Roussel, G.: Specification and implementation of grammar coupling using attribute grammars. In: Prog. Lang. Impl. and Logic Prog. (PLILP '93). Volume 714 of LNCS. (1993) 123–136
17. Hedin, G.: An object-oriented notation for attribute grammars. In: Proc. of European Conf. on Object-Oriented Prog., ECOOP'89, Cambridge Univ. Press (1989)
18. Vogt, H., Swierstra, S.D., Kuiper, M.F.: Higher-order attribute grammars. In: ACM PLDI Conf. (1990) 131–145
19. Ekman, T., Hedin, G.: Rewritable reference attributed grammars. In: Euro. Conf. on Object-Oriented Prog., ECOOP'04. Volume 3086 of LNCS. (2004) 144–169
20. Simonyi, C.: The future is intentional. IEEE Computer **32**(5) (1999) 56–57
21. Van Wyk, E., Krishnan, L., Bodin, D., Johnson, E.: Adding domain-specific and general purpose language features to Java with the Java language extender. In: Companion to the Proc. OOPSLA. (2006)