International Journal on Software Tools Technology Transfer manuscript No.
(will be inserted by the editor)

Eric Van Wyk · Mats Per Erik Heimdahl

Flexibility in Modeling Languages and Tools: A Call to Arms

Abstract In model-based development, the software development effort is centered around a formal description of the proposed software system; a description that can be subjected to various types of analysis and code generation. Based on years of experience with model-based development and formal modeling we believe that the following conjectures describe fundamental obstacles to wide adoption of formal modeling and the potential for automation that comes with it; (1) no single modeling notation will suit all, or even most, modeling needs, (2) no analysis tool will fit all, or even most, analysis tasks, and (3) flexible and stable tools must be made available for realistic evaluations and technology transfer. These conjectures form the basis for the call to arms outlined in this report.

To make automated software engineering techniques more useful for more types of developers and allow us to move forward as a community it is crucial that we develop the foundation for building extensible and flexible modeling language processing tools. New common-infrastructure-based approaches are needed as traditional approaches based on file-based processing of intermediate language representations are not adequate. In this report we outline and illustrate the problem and discuss a possible solution. To initiate the discussions in the community, we hypothesize that languages and tools built using higher-order attribute grammars with forwarding can serve as a basis for such flexible language processing tools; tools that will allow us to unify our efforts and help bring our collective work to a broader audience.

This work was partially funded by NSF CAREER Award #0347860, NSF CCF Award #0429640, and the McKnight Foundation.

Department of Computer Science and Engineering
University of Minnesota
Minneapolis, Minnesota, USA
evw,heimdahl@cs.umn.edu

1 Introduction

Traditionally, software development has been largely a manual endeavor. Informal natural language requirements have been manually captured, a design satisfying the requirements has been manually derived, and code implementing the design has been manually coded. Recently, there has been a move away from such manual techniques to a new paradigm commonly called *model-based development*. In this paradigm, the development effort is centered around a formal description of the proposed software system—the 'model' in model-based development. For validation and verification purposes, this *formal model* can then be subjected to various types of analysis, for example, completeness and consistency analysis, *e.g.*, [36,38], model checking, *e.g.* [13,29], theorem proving, *e.g.* [5,2], and test case generation, *e.g.* [27, 55,59]. There are currently several commercial and research tools that attempt to provide these capabilities– for example, the commercial tools Esterel Studio (with its graphical notation Safe State Machines) and SCADE Studio from Esterel Technologies [22], Statemate from i-Logix [32], Simulink and Stateflow from The Mathworks Inc. [52], and SpecTRM from Safeware Engineering [51]; examples of research tool are SCR[37], Ptolemy [48], and RSML^{-e}[67].

Our goal for the last decade has been to dramatically increase the quality and productivity of software development for critical control systems by centering the development around fully formal models extensively supported by tools. These tools must allow engineers to specify system requirements in an appropriate and familiar notation and to *effectively* analyze the specifications to ensure that safety critical properties are satisfied. In the course of our work we developed an approach to simulation and validation of formal specifications for process control systems called specification-based prototyping [67]. Specification-based prototyping combines the advantages of traditional formal specifications (e.g., preciseness and analyzability) with the advantages of rapid prototyping (e.g., risk management and

early end-user involvement)—goals that are now shared in the move towards model-based development. To enable specification-based prototyping, we developed the fully formal specification language RSML^{-e} (Requirements State Machine Language, without events)[68] and its execution and analysis environment NIMBUS [53,66]. We have successfully evaluated the capabilities on various case studies from the avionics, transportation, and mobile robotics domains [67,68,66], and the environment has been used for several years in industrial research projects [53,41,60,35,11]. Note that our experience is primarily in the critical embedded systems domain, but we believe our observations and proposed solutions are applicable to other domains as well. Based on our experience with model-based development and formal modeling in various critical systems domains we believe that the following conjectures describe the fundamental obstacles that must be overcome for model-based development to have the dramatic real-world impact we, and many others in the community, envision. These conjectures form the basis for our call to arms outlined in this report.

Conjecture 1: No modeling language will be universally accepted, nor universally applicable. Even closely related domains, such as avionics and medical technology, have justifiably different and entrenched views of what notations and features a modeling language should have to be suitable for their domains. Nevertheless, certain *classes* of languages do have wide appeal, for example synchronous languages such as, Safe State Machines [22], SCADE [22], and SCR [37].

Conjecture 2: No verification and validation tool will satisfy all of a user's analysis needs. Analysis tools are quite specialized and new sophisticated analysis tools are constantly emerging. Somehow mating a wide and growing collection of analysis tools with a variety of modeling languages (Conjecture 1) is inevitable.

Conjecture 3: To make progress in model-based development, practicing engineers must evaluate proposed solutions on practical problems; if proposed theories, methods, and tools do not solve real problems they are of little or no value. Therefore, the methods and tools must be flexible enough to easily adapt and be improved based on what is learned from using the tools on real-world problems.

The goal with this report is to generate awareness of what we perceive to be a serious problem and attempt to build a community around an effort to develop the foundation for building extensible and flexible modeling language processing tools that can satisfy the following three goals derived from the above conjectures—successful modeling tools must

1. allow easy extension and modification of formal modeling notations to meet the domain specific needs of a class of users,
2. allow easy construction of high-quality translations from the modeling notations to a wide variety of analysis tools, and
3. enable controlled reuse of tool infrastructure to make tool extensions cost effective.

Currently, we are aware of no tools that support the easy customization (or complete redefinition) of a modeling language necessary to accommodate the needs and likes of stake-holders in different domains. Some modeling notations, notably UML [61], attempt to appeal to different domains by incorporating a wide variety of rich constructs. The richness of such languages, however, makes them unsuitable for most formal analysis. In another approach, an intermediate model notation is used to accommodate the integration of a wide variety of analysis tools—various modeling formalisms are translated to the intermediate notation that can in turn be mapped into suitable analysis tools. Nevertheless, with an intermediate notation valuable information from the source language, for example, model structure, may be lost in the translation to the intermediate notation and there is a risk for exponential growth in model size during translation. We believe other techniques are needed to satisfactorily solve this problem.

2 Background and Related Work

There has been an immense amount of research done in formal modeling tools, mappings from modeling notations to analysis tools, and translation technology. The coverage in this section is by necessity cursory, but we will discuss our previous experiences and the core problem, and then present a short overview of efforts related to our research agenda.

2.1 The Problem

As stated above, our conjectures are based on extensive experience in this domain and they have a large impact on how languages and tools are adopted in industry (or not adopted as is often the case). In any larger project there will be a need to use several different modeling notations. For example, notations such as Simulink [52] or SCADE [22] are suitable for the control oriented aspect of a system design whereas Safe State Machines [22] or Statecharts [32] are more suited for the discrete aspect of a system. Other notations, for example, SCR [39], SpecTRM-RL [51], and RSML^{-e} [67] are considered better suited for high-level models used in the requirements domain. Thus, there must be the flexibility to select the right notation for the right task without having to "retool" the entire organization. In addition, after working on numerous projects in collaboration with industry, we have noticed that the adoption of new modeling tools

is often hampered for nontechnical reasons. Typical obstacles to tools adoption are seen in comments such as *"We like what you are doing, but you do not have internal events like Esterel-we find events essential"*, *"We like SCR, can you work with that?"*, and *"We think the guard-conditions in Statecharts are messy, could you provide the nice tables we saw in RSML?"*. These obstacles apply to both academic and commercial tools, and attempting to accommodate these wishes is typically economically infeasible—modifying tools and languages is simply too costly. Furthermore, as formal modeling techniques are slowly being adopted in industry, the need for powerful analysis tools becomes acute. As would be expected, no analysis tool is suitable for all tasks. Given the rapid change in verification and analysis technology, we will most likely see a steady stream of new exciting analysis tools that need to be incorporated into tools in the future.

To summarize, there is a critical need to accommodate numerous notations (and to modify these notations to fit the customer's individual problems and taste) and a multitude of analysis tools. Because tool vendors and research groups cannot currently support these needs, promising tools and techniques are not adopted for superficial reasons and useful new analysis techniques are not adopted because of cost and technical difficulties. Therefore, we have come to the conclusion that we need a catalytic infrastructure for the design, development, and deployment of formal modeling tools that will serve two purposes; (1) it will allow the research community to evaluate and quickly deploy new ideas in a stable environment and (2) provide a blueprint for tool vendors for how to build tools that can be customized to meet the customer's needs. We hope this paper will serve as a catalyst and start of a community-wide initiative that will dramatically improve the penetration of formal modeling and recent analysis research results in industry.

2.2 From Models to Analysis Tools

The most common current solution to get from modeling tools to analysis tools is to develop separate translators from each modeling notation to each analysis tool. As the numbers of notations and analysis tools grow, this is an unsustainable solution since we may need $n \cdot m$ translators to map n modeling notations to m analysis tools. Providing such a collection of high-quality translators is not economically feasible.

To address this problem, an intermediate notation can be used as an interchange format between modeling notations and the analysis tools; for example, DC for synchronous languages [58], IF for the exchange of models between model checkers [8], and SAL for general purpose applications [6]. An intermediate language sitting between the modeling notations and the analysis tools reduces the number of translators needed to map n notations to m analysis tools to $n + m$ translators.

There are, however, problems with this approach. For example, it is quite difficult to choose the *right level of abstraction for the intermediate language*, even when one knows all of the modeling-languages and target-languages. Consider choosing a low-level intermediate language, for example, Lustre [30], that does not have template types. If we have a modeling-language that has template types, such as SCADE, the translator must instantiate the template type for each of its uses. Since these template types can be immensely useful to model, for example, generic communications channels in a systems architecture, we may have dozens of occurrences to instantiate (possibly one for each connector in the architecture). This may be appropriate when translating to an analysis tool such as NuSMV [12] which does not allow template types, but it may be wholly inappropriate when targeting an analysis tool such as PVS [56] which does allow template types. Because of this information loss, the translation through an intermediate language to PVS cannot take advantage of all of the capabilities of the target language thus making verification in PVS more difficult and time consuming than strictly needed. This provides a *low-quality* translation of the source language to PVS. On the other hand, if the intermediate language has many high-level constructs, such as template types, the translations to the different target languages become much more difficult. Since we believe that new notations will constantly evolve and new analysis tools will become available in a constant stream (as an example, consider the dramatic evolution of SAT based model checkers in the last few years), the selection of an appropriate intermediate notation will become impossibly difficult and we do not see this as a feasible long term solution.

Intermediate languages used in compilers for programming languages also suffer similar drawbacks. As an example, consider SUIF (the Stanford Unified Intermediate Form) [64], an intermediate representation that allows different researchers in compiler optimizations to develop and add their own optimizing pass to a compiler built around SUIF. In some respects SUIF lacks important source language information that is useful in optimizations; for example, Fortran EQUIVALENCES are not represented directly in SUIF. On the other hand, many, for example [63], have criticized SUIF as being too complex and thus taking too long to learn how to use. These observations reinforce our opinion that selecting a fixed intermediate language based on incomplete knowledge of both source and target notations will be an exercise in futility.

Another approach has been adopted by groups attempting to automatically generate analyzers and translators from the definition of a notation's semantics—an approach in principle similar to the way in which parsers are generated from grammar definitions. Dillon and Stirewalt [17] have developed an approach where the operational semantics for process-algebra and temporal-logic notations can be described and semi-automatically

translated to inference graphs expressing the specification and its possible next states. Cleaveland and Sims [14] have developed a similar semantics-based compiler for translating process algebra notations into the input language of the Concurrency Workbench. These approaches, however, cannot represent data variables and and are not intended for the breadth of support in terms of extendibility, semantic analysis, execution, and translation we envision.

Finally, and most closely related to the solution we seek, Niu *et al.* [54] propose a template-based approach to structure the operational semantics of model-based specification notations. In their work, a parameterized template pre-defines behavior that is common among notations. Template parameters instantiate the parameterized definitions with notation-specific semantics. In this way they can accommodate a surprisingly wide variety of notations, for example, CSP, Statecharts, SCR, etc. The goal of this work is, however, very different from ours. Their work is based around one notation where the goal is on *exploration* and *comparison* of various semantic choices for this notation. Any extensions or deviations from their notation, for example, in terms of syntax, scoping rules, naming conventions, etc., must be accommodated outside their tools framework. In addition, their template approach does not lend itself to the flexible translation, model execution, and debugging capabilities we are aiming for.

2.3 Extensible Language Techniques

Our stated goal of allowing domain specific language features to be added to a modeling language has been studied in the area of programming languages and there are many tools and techniques that attempt to solve this problem. Besides adding new syntactic forms to a modeling language, we also require that these new constructs be able to specify some semantic analysis so that they can generate domain specific error messages, debugging behavior, as well as specify their direct translations to target languages when appropriate. Although each of these requirements are addressed by some of the many tools and techniques for language extensibility in the literature, no single approach addresses all of them. Traditional syntactic [9,47], hygienic [45,15], and programmable [74] macros systems and embedded domain specific languages [40,49] do allow new language constructs to be added to a language. However, they lack an effective way to perform the necessary static analysis. On the other hand, meta-object protocol systems, *e.g.* [10, 43] provide limited opportunities to add new language constructs but can perform the static analyses needed to, for example, check for domain specific errors. Some modern macro systems [4,3] also provide some static analysis capabilities that can be used for type-based macro expansion and error reporting, but these are not general enough to support the wide range of static analyses that must be performed to meet our stated goals.

Attribute grammars [44,57] provide the foundation for what we believe will be a successful direction of inquiry. Language constructs are specified by productions and their explicit semantics can be defined by attribute definitions. The problem of modular language definition and extensibility has received much attention from the attribute grammar community, *e.g.* [24,42,25]. Some systems are guided by functional programming ideas and use, in essence, higher order functions as attributes in their quest for modular specifications, *e.g.* [23,28,1,46]. Others are inspired by the object-oriented paradigm and employ inheritance to achieve a separation of concerns [33, 7]. Of most use to us are higher-order attributes [73,65] that allow abstract syntax trees to be attribute values and reference attributes [34] that point to possibly remote nodes in the abstract syntax tree. Also of interest are Hedin's re-writable reference attribute grammars [18] in which a mechanism for rewriting the abstract syntax tree based on rewrite-rules is used. Attributes are only retrieved from rewritten trees and thus this technique differs from forwarding which allows attributes to be retrieved from the original and rewritten (that is "forwarded-to") tree (see Section 3.1). Hedin and Ekman's JastAdd system [20] has been used to implement an extensible Java compiler [19]. An analysis for checking non-null types in Java programs is one of several interesting language extensions [21] developed in their system.

Most closely related to the extensible language framework described below is Microsoft's Intentional Programming system (IP) [62,71] [16, Chapter 11]. This system allowed programmers to add domain specific features, called *intentions*, to their programming language in a style similar to attribute grammars. Although not as cleanly defined in IP as in attribute grammars, IP did contain the essence of reference attributes and higher order attributes. The main innovative feature of IP was *forwarding*, a technique used to define new constructs in terms of host language constructs. In [70], we showed how forwarding can be used in attribute grammars to modularly specify languages and how the absence of forwarding hinders the modularity we seek and makes the addition of new language features more difficult. Our main criticism of IP [71] was its ad-hoc nature that prevented any static analysis of language extensions to test their compatibility.

3 A Proposed Framework

In this section we will illustrate how attribute grammars extended with forwarding [70] can be used to define language extensions on top of a host-language. Note here that we do not categorically state that we believe this is the best solution to the problem outlined in Section 2.1,

we simply outline what we see as a promising direction to start a community-wide dialogue with the goal to establish a tools architecture that will satisfy our need for flexibility and extensibility.

We hypothesize that an *extensible language* implemented using attribute grammars with forwarding [70] can serve as a *host-language* for (1) a plethora of domain specific *language-extensions* that can be combined to construct new *modeling-languages* suitable for different audiences and domains, (2) the translation of these extension constructs into their semantically-equivalent representation in the host-language and the host-language translation into various target-languages, and (3) the direct translation of a language-extension construct to an analysis tool's language when the default translation provided through the host-language is inadequate.

In the proposed framework, a language extension is specified as an attribute grammar *fragment* that contains new productions and attribute definitions for these productions and those in the host language. The activity of creating an extended language specification can be as simple as taking the union of all the productions and attribute definitions in the host language and language extension specifications. This is performed by the framework tools and provides for a significant degree of modularity between language extensions. In traditional attribute grammars the modularity and reuse of language features specified as attribute grammar fragments is achieved only by writing attribute definitions that "glue" new fragments into the host language attribute grammar. These attribute grammars, which do not have forwarding, cannot implicitly define semantics for a language construct and this is crucial for the modularity we seek.

To give the reader an introduction to attribute grammars and forwarding, we will first illustrate in Section 3.1 the mechanism of forwarding in terms of a construct familiar from the programming language domain—a *foreach* loop. We will then illustrate how a well defined language suitable for the embedded systems domain— Lustre [30]—can be used as a host-language and illustrate how extensions can be defined to start building up a modeling-language on top of this host-language. We provide some examples of using language extensions to allow for flexibility in the modeling notation in Section 3.2. In Section 3.3 we describe how forwarding may be used to easily implement high-quality translations to many analysis engine languages while avoiding many pitfalls of intermediate languages.

3.1 Forwarding in Attribute Grammars

Forwarding is a *unifying technique that allows us to mimic common language extension processing techniques* like macro expansion, simple term rewriting, and meta-object protocols inside an attribute grammar framework and thus makes it possible to declaratively specify expressive language extensions. To use *forwarding*, a language construct specifies a *semantically equivalent* construct that defines the semantics not explicitly defined by the "forwarding" construct. In attribute grammar terms, a production defines a *distinguished* attributed abstract syntax tree that provides default values for synthesized attributes that are not explicitly defined by the production.

```
foreach : for⟨Stmt⟩ ::= elemType⟨Type⟩ elem⟨Id⟩
                        collection⟨Expr⟩ body⟨Stmt⟩
  for.pp = "foreach " + elemType.pp + elem.lexeme +
           " in " + collection.pp + " do " + body.pp
  for.errors = if collection.type.implements (Collection)
               then no-error
               else mkError ... for.pp ...
  forwardsTo parse
    "{ 'elemType.pp' 'elem' ;
       for ( Iterator 'iter' = 'collection'.iterator();
             'iter'.hasNext() ; )
         { 'elem' = ('elemType.pp') 'iter'.next() ;
           'body' }
     }"
    where iter = generate_new_unique_Id ( )
```

Fig. 1 The production specifying the *foreach* loop extension.

A familiar example from programming languages will help to clarify. Consider adding a simple *foreach* construct to Java to iterate over Java Collections.[1] An attribute grammar production for doing so is shown in Figure 1. This puts syntactic sugar on a popular programming idiom but also defines its own simple error-checking semantic analysis. The *foreach* named production has a left-hand side statement non-terminal (⟨*Stmt*⟩) named *for* and right-hand side non-terminals for the type expression (⟨*Type*⟩), the identifier (⟨*Id*⟩), the collection over which the loop iterates (⟨*Expr*⟩), and the body of the loop (⟨*Stmt*⟩). All are named as indicated. It explicitly defines the synthesized pretty-print *pp* and *errors* attributes allowing it to generate errors messages containing code written by the programmer. (We use the familiar dot (.) notation to access attribute values.) The attribute *type* which occurs on expression non-terminals and is referenced here on *collection* is a *reference* [34] attribute that "points to" a node in the program's abstract syntax tree that defines the type of the collection. The *implements* method on that node is used to perform the type checking that defines *errors*.

The production also specifies its *forwards-to* tree as the expected host-language block-loop construct. This construct is built by parsing the given string and using the "unquote" operator ('_') to reference right-hand side subtrees, their attributes and the identifier *iter*. For example, the *pp* attribute on *elemType* is used to cast the Java *Object*-type value returned from the iterator method *next*.

[1] This feature was missing from Java until version 1.5 of the language.

```
type Status = enum { Unknown, Above, Below } ;
node ASW (AltQuality:Q, AltThres:int, Hyst:int)
     returns (AltStatus:Status);
let AltStatus = Unknown ->
      if AltQuality = Good and Altitude > AltThres
         and (PREV(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. 2 A Lustre like definition of the state variable *AltStatus*.

The left-hand side node *for* of type $\langle Stmt \rangle$ generated by the production *foreach* is called the "forwarding-node." When it is queried for an attribute that it *does not explicitly define*, for example *jbc*, an attribute that defines its Java byte code, it passes, or "forwards", that request to the "forwards-to" node, the block-loop construct in this example. The forwards-to node returns the attribute's value. Thus, the *jbc* attribute for a foreach loop is the same as the *jbc* attribute on the block-loop that is forwarded to. The main idea here is that we re-use all attributes defined on the block-loop except those with explicit overriding definitions. Because the forwards-to tree will require inherited attributes, these are copied from the forwarding-node unless they are explicitly defined. Note that in some cases the forwards-to node may also forward the query; we will eventually find a value for the attribute since *all language extensions forward (directly or indirectly) to constructs in the host-language.*

Forwarding is similar to macro expansion in that both reuse the semantics of existing language constructs, but unlike macros, forwarding productions also define semantics, by way of attribute definitions. In the example here, these attributes generate proper error messages. This is something that macro systems cannot do. Also note that we have not specified how the concrete syntax of this extension is specified since this is done in a straight-forward and expected way. We thus limit our discussions to the more interesting issues in specifying the semantics of language extensions via attribute grammars.

Silver [69] is an extensible attribute grammar system which supports forwarding and other modern extensions to the attribute grammar paradigm. It has been used to construct an extensible specification of Java [72] in which several language extensions, including the for loop example above, have been implemented.

3.2 Modeling-Languages as Extensions to a Host-Language

To illustrate how a new modeling-language can be specified as a set of language extensions to a host-language, we will use a simple example—the Altitude Switch (ASW). The ASW is a (somewhat hypothetical) avionics system that turns power on to another system when the aircraft descends below a threshold altitude and turns it off when the aircraft ascends above the threshold altitude plus a hysteresis factor. As an example, we focus on one of the state variables that models the ASW behavior—the *AltStatus* variable used to track whether the aircraft should be considered above or below the threshold.

Non-Terminals: $\langle Program \rangle$, $\langle NodeOrTypeDefList \rangle$, $\langle Node \rangle$, $\langle TypeDef \rangle$, $\langle DclList \rangle$, $\langle Dcl \rangle$, $\langle Type \rangle$ $\langle Stmt \rangle$, $\langle Expr \rangle$, $\langle Id \rangle$

Attributes: $pp : String$, $type : \langle Type \rangle$, $env : [\ (\ String, \langle Type \rangle\ )\ ]$ $smv\_trans : String$, $pvs\_trans : String$

Productions: (Format: $name : l.h.s. ::= r.h.s.$)

$program : prog\langle Prog \rangle ::= nts\langle NodeOrTypeDefList \rangle$
$typedef : td\langle TypeDef \rangle ::= name\langle Id \rangle\ typeExpr\langle Type \rangle$
$node : node\langle Node \rangle ::= inpts\langle DclList \rangle\ outpts\langle DclList \rangle\ lcls\langle DclList \rangle\ assgns\langle StmtList \rangle$
$dcl : id\_dcl\langle Dcl \rangle ::= name\langle Id \rangle\ type\langle Type \rangle$
$bool : boolt\langle Type \rangle ::= \epsilon$
$assign : assgn\langle Stmt \rangle ::= name\langle Id \rangle\ expr\langle Expr \rangle$
$add : expr\langle Expr \rangle ::= left\langle Expr \rangle\ right\langle Expr \rangle$
$ifthen : i\langle Expr \rangle ::= c\langle Expr \rangle\ t\langle Expr \rangle\ e\langle Expr \rangle$
$not : nexpr\langle Expr \rangle ::= expr\langle Expr \rangle$
  $nexpr.type = global\_bool\_reference$
  $nexpr.errors =$ **if** $expr.type.is(\ nexpr.type\ )$ **then** *no-error* **else** *mkError ...*
$idref : expr\langle Expr \rangle ::= name\langle Id \rangle$
  $name.type = lookup\ (name.lexeme,\ expr.env)$

Fig. 3 Lustre host-language attribute grammar.

A Lustre-like specification of *AltStatus* is shown in Figure 2. In it, the initial value of *AltStatus* is undefined (indicated by the '*Unknown* →' construct) and thereafter the variable is 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 (the previous *AltStatus* was *Unknown*) and we are above the threshold or (2) *AltStatus* has been established and we are above the threshold plus the hysteresis. We declare *AltStatus* to be *Below* if we have reliable altitude readings and the altitude is less than or equal to the threshold. If the altitude readings are not reliable *AltStatus* is *Unknown*.

Lustre as a host-language:

We hypothesize that Lustre [30] may be a suitable host-language in our domain of interest for two reasons. First, Lustre is expressive enough to capture a large class of interesting behaviors and it has a well defined and simple semantics making it suitable for formal treatment by various tools [31]. Second, Lustre is supported by commercial tools which provide, for example [22], a code generator that has been qualified for use in safety critical applications. Thus, it is of interest to industrial partners. Lustre may be implemented as an extensible host lan-

guage by writing an attribute grammar that defines its language constructs and defines attributes that perform semantic analysis and translation.

Part of the context free grammar that defines the abstract syntax of the Lustre host-language and some of the associated attribute definitions are shown in Figure 3. For our purposes here, we will consider a Lustre program to be an interleaving sequence of nodes (represented by $\langle Node \rangle$ non-terminals) and type definitions (represented by $\langle TypeDef \rangle$ non-terminals). This list is represented by the non-terminal $\langle NodeOrTypeDefList \rangle$.[2] The *node* production essentially defines a function that takes a set of input values, specified by the declaration list *inpts*, defines some local variables (*lcls*), and computes values for the locals and the output variables (*outpts*) via the sequence of assignments statements (specified by the list of $\langle Stmt \rangle$ non-terminals *assgns*). Declarations bind a name to a type and assignment statements are defined by the *assign* production as expected. As in the *for-each* example, we do make use of reference attributes and the *type* attribute is a reference to $\langle Type \rangle$ nodes in the abstract syntax tree. To link identifier references to declarations we use an inherited environment attribute *env* that is passed down the tree and is a list of *string* and reference to $\langle Type \rangle$ pairs. The *idref* production searches its *env* list for its type. The relational, logical, and arithmetic expressions are mostly self-explanatory and only a few are shown here. The *not* production specifies that its type is boolean (where *global_bool_reference* is a reference to a type node generated by the *bool* production) and generates an error if its child is not a boolean expression. Though not shown in Figure 3, all of these productions also define a pretty-print attribute *pp* that can be used to display error messages. We may also define an *errors* attribute similar to the one in the *foreach* example. As explained further in Section 3.3, the productions may also define the translation attributes *smv_trans*, *pvs_trans*, and others that specify how the constructs are translated in the various target languages.

The Lustre host language can now be extended with modeling language features that may be useful in different domains or preferred by different user communities. We describe how RSML^{-e} state variables and events can be added as language extensions.

Implementing RSML^{-e} as a collection of language extensions:

Since the Lustre host-language may not be the most suitable for review by domain experts (for example, pilots or air traffic controllers) an alternate notation would be of interest. The modeling notations SCR and RSML^{-e} are such notations. They have been well-received and shown to be relatively easy to understand and use [39, 50, 75]. Figure 4 shows a fragment of an RSML^{-e} specification of the ASW. The figure shows the definition of

[2] Note that in Figure 3 we have omitted the list productions of the $\langle X\_List \rangle ::= \langle X \rangle \langle X\_List \rangle$ as they can be inferred by the reader.

```
STATE_VARIABLE AltStatus :
  VALUES : { Unknown, Above, Below }
  INITIAL_VALUE : Unknown

  EQUALS Above IF
   TABLE
    PREV(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
   TABLE
    AltQuality = Good : F ;
   END TABLE
END STATE_VARIABLE
```

Fig. 4 RSML^{-e} like definition of the State Variable *AltStatus*.

the state variable *AltStatus* discussed above. The conditions under which the state variable changes value are defined in the *Equals* clauses in the definition. The tables are adopted from the original RSML notation [50]—each column of truth values represents a conjunction of the conditions in the leftmost column (T represents the case when the condition is true, F represents the negation of the condition, and a '*' represents a "don't care" condition). In a table with several columns we take the disjunction of the columns; thus, the table is a way of expressing conditions in a disjunctive normal form.

As we stated above, a language extension is an attribute grammar *fragment* that specifies productions that define the abstract syntax of any new language constructs, attribute definitions for these new productions, and attribute definitions for existing productions in the host-language. To implement the *State_Variable* language construct in Figure 4 we follow this pattern and define a set of productions to define the syntax of state variable constructs. We also define attributes on these constructs that check for errors and assist in building the semantically equivalent Lustre language construct that the state variable will forward to. Part of this specification is given in Figure 5.

The *State_Variable* construct is essentially a variable assignment statement like those in the host-language with the exception that it also serves as a declaration of the assigned variable. The production defining a state variable ($stateVar$) has on its right hand side an identifier *name* (in the example this is *AltStatus*), a non-terminal for the possible values, its initial value $init\langle Expr \rangle$, and an expression $expr\langle Expr \rangle$ for subsequent values. The implicit semantics of a state variable are provided by the Lustre assignment statement that it forwards to and is

```
stateVar : statevar⟨Stmt⟩ ::= name⟨Id⟩ type⟨Type⟩
                              init⟨Expr⟩ expr⟨Expr⟩
   statevar.errors = if ¬ name.type.is(init.type)
                     then mkError ...
                     else if ¬ name.type.is (expr.type)
                          then mkError ... else no-error
   expr.state_var_name = name
   statevar.liftedDcls = parse "'name' : 'type'"
   forwardsTo parse "'name' = 'init' → 'expr'"

equalsList : ⟨Expr⟩ ::= eqs⟨EqualsList⟩
   forwardsTo eqs

equalsList1 : ⟨EqualsList⟩ ::= value⟨Expr⟩ cond⟨Expr⟩
   forwardsTo parse "if 'cond' then 'value'
                     else pre('eqs.state_var_name')"

equalsListn : ⟨EqualsList⟩ ::= value⟨Expr⟩ cond⟨Expr⟩
                               eqs⟨EqualsList⟩
   forwardsTo parse "if 'cond' then 'value' else 'eqs'"

table_p : table⟨Expr⟩ ::= tableRows⟨TableRowList⟩
   table.errors = "check rows have same num of columns"
   forwardsTo "disjunctive normal form expression"

tablerow : tabrow⟨TableRow⟩ ::= cond⟨Expr⟩ tfss⟨TFSList⟩
```

Fig. 5 The *State_Variable*, *Equals* and *Table* specifications .

semantically equivalent to, essentially, "'name' = 'init' → 'expr'". Since a *State_Variable* production also declares a variable, this production builds the declaration as a Lustre declaration of the form "'name' : 'type'" and indicates that this declaration should be *lifted* to the enclosing *node*. This is done by placing the declaration, implemented as a higher-order (a abstract syntax tree-valued) attribute in a synthesized attribute that collects such declarations and moves them up the tree to the point that they can be inserted into the enclosing Lustre node construct.

In RSML^{-e} the value of a state variable remains unchanged if none of the conditions in the *Equals* clauses hold. Thus, the expression *expr* in the Lustre assignment that a *stateVar* forwards to must know the name of the variable being defined. It uses this variable in the definition of the default value of the expression *expr*, *pre*(*eqs.state_var_name*), if the condition of each *Equals* clause in the list fails to hold. This is seen in the *if* condition that the *equalList1* production forwards to. To accomplish this, the *stateVar* production defines the inherited attribute *state_var_name* that is passed down the abstract syntax tree of the expression by the *state_var_name* attribute definitions on the *stateVar*, *equalsList*, *equalsListn equalList1* productions. The specification for constructing the disjunctive normal form expression that becomes the condition of the if-then-else that the table translates to is more verbose than it is difficult. Note that the *table_p* production must perform some of its own error checking. For example, checking that each row in the table has the same number of columns is something that it must do; the expression that it forwards to could not detect this sort of error.

```
STATE AltStatus :
   VALUES : { Unknown, Above, Below }
   INITIAL_VALUE : Unknown
   TRANSITION Unknown TO Above IF
     TABLE
         AltQuality = Good          : T ;
         Altitude > AltThres        : T ;
      END TABLE

   TRANSITION Below TO Above IF
     TABLE
         AltQuality = Good          : T ;
         Altitude > AltThres + Hyst : T ;
      END TABLE
   ...
END STATE
```

Fig. 6 Partial RSML^{-e} Transition Style Definition of the State Variable AltStatus.

Some might find the assignment style in RSML^{-e} somewhat counterintuitive and prefer to think about the enumerated variable *AltStatus* as a state machine. This can easily be accommodated with another language extension that is quite similar to the *Equals* clauses presented above. The first *Equals* clause shown in Figure 4 can be represented by the first two transition clauses shown in Figure 6. The attribute grammar specification for the transition language extension is quite similar to the *Equals* clauses and again uses higher-order (tree-valued) attributes to construct the Lustre constructs that it would forward to.

Events as an extension:

Many find events to be useful in specifications of systems similar to the altitude switch discussed here. In Figure 7 we show part of a specification for the altitude switch written in the host-language extended with a notion of events. Here, the occurrence of the externally defined event *AltReceivedEvent* which indicates when new altitude measurements have been received, is used in the computation of *AltStatus*. The value of *AltStatus* is only updated when the *AltReceivedEvent* event is thrown. When *AltStatus* is updated it throws either a *AltClassifiedEvent* or *AltLostEvent* event to indicate the outcome of the assignment of the *AltStatus* value. We have abbreviated the three expressions that appear in the original Lustre ASW in Figure 2 as C1, C2, and C3 here to simplify the presentation. In Figure 8 we see how these events can be emulated using Boolean variables. This is the code as it looks after some of the "transformations", implemented by forwarding, have taken place.

Events can also be implemented as a language extension. The attribute grammar specification for events defines three primary productions—one to generate or "throw" an event, another that evaluates to *true* if it "catches" the specified event, and an event declaration. The specifications for these productions are straight forward, but they do require a more complex set of supporting attributes to generate the boolean variable declarations and their defining expressions that form their Lus-

```
var AltClassifiedEvent : Event ;
    AltLostEvent : Event ;

AltStatus = Unknown ->
  if catch AltReceivedEvent and C1
     then throw AltClassifiedEvent return Above
  else if catch AltReceivedEvent and C2
       then throw AltClassifiedEvent return Below
  else if catch AltReceivedEvent and C3
       then throw AltLostEvent return Unknown
  else pre(AltStatus)
```

Fig. 7 Event/Action style constructs added to the host-language.

```
var AltClassifiedEvent : bool ;
    AltLostEvent : bool ;

AltStatus = Unknown ->
  if      AltReceivedEvent and C1 then Above
  else if AltReceivedEvent and C2 then Below
  else if AltReceivedEvent and C3 then Unknown
  else pre(AltStatus)

AltClassifiedEvent = False ->
      (catch AltReceivedEvent and C1) or
      (not (catch AltReceivedEvent and C1) and
       (catch AltReceivedEvent and C2))
AltLostEvent = False ->
      (not (catch AltReceivedEvent and C1) and
       not (catch AltReceivedEvent and C2)
           (catch AltReceivedEvent and C3)
```

Fig. 8 Events emulated through boolean variables.

tre implementation (that these extensions forward to). Declarations of events need to be transformed into declarations of boolean variables. These variables replace events and are true under the condition in which the original event would have been thrown and false otherwise. Thus, the *catch* production simply forwards to a reference to the boolean variable emulating the event since both are true under same conditions. The *throw* production simply forwards to the expression that it "returns" since when events are emulated the boolean event variable and its defining expression effectively replace the throw constructs. This is seen in the beginning of Figure 8.

Of particular interest is how the assignment statements for the event-emulating boolean variables are generated. In the example, the boolean variable *AltClassifiedEvent* is generated to replace the *AltClassifiedEvent* event. The assignment statement to the *AltClassifiedEvent* variable in Figure 8 (with catch constructs yet to be removed) is also generated to set this variable to *true* under the conditions that the event would be thrown. The *AltClassifiedEvent* variable is naively computed to be *true* if the first if-condition (in Figure 7) is *true* or if the first if-condition is *false* and the second if-condition is *true*. For an event e, it is a relatively straight forward process to compute this expression. For each *throw* e construct, we build an expression from the enclosing if-conditions that must be *true* to cause that throw instance to "fire." Again, we leave out the implementation of this since an *inherited* higher order attribute can be specified to pass down the enclosing if-condition $\langle Expr \rangle$-trees to a throw statement. It constructs an expression that is true when the event is thrown. Such expressions are collected from each throw, via a synthesized attribute, and the expression that is the disjunction of them is created to become the expression in the defining boolean assignment as seen in the example above.

Given the mechanisms discussed above, it is easy to combine events and state-machine transitions into a "transitions with events" notation similar to the original definition of RSML [50]. This flexibility in introducing new language constructs that can be targeted for various stake-holder needs is highly desirable. In addition, the attribute grammar framework provides an opportunity for static analysis at the appropriate level of abstraction, that is, in the extended modeling-language rather than in the host-language. For example, in the examples above we have assumed a lazy-evaluation semantics of the *Equals* or transition clauses with which we have extended our host-language—this semantics is captured through the nested if-statements to which the extensions forward. With lazy-evaluation, it does not matter if two guards overlap—we will simply use the first one that evaluates to true. If the evaluation order is nondeterministic—a choice that makes perfect sense if one thinks of the variable as a state machine in a graphical notation—forwarding to nested if-statements does not capture the semantics accurately. In this case, we could define an *extension specific analysis* to assure that the *Equals* and transition clauses are mutually exclusive [36]. We believe the capability to define extension specific analysis will be invaluable to help assure that language extensions are properly used. The same mechanism could also be used to assure that a modeling language is used according to various domain specific style guidelines.

Another example of static analysis of extension constructs involves error checking. In all of these extensions, we would want to define the basic error checking attribute *errors* on the productions defining the new language constructs. If we do not define the error checking attributes any type errors will be reported by the *errors* attribute on the forwarded-to construct. Thus, many errors will be discovered but the generated error message will be in terms of the forwarded to expression, not the transition expression that was written by the user. As we noted in the case of the *table_p* production in Figure 5 some errors can only be detected on the forwarding extension construct. This demonstrates one aspect of how attribute grammars with forwarding provide a convenient yet powerful mechanism for specifying language extensions. By specifying additional attributes, in this case *errors*, our language extensions behave more and more like first-class constructs in the language and pro-

vide all of the capabilities that one expects from "built-in" language features.

We have specified a subset of Lustre as a host language in Silver and implemented several of the constructs described above as language extensions. A more complete description of those specifications can be found in a previously published paper [26].

3.3 Execution, Debugging, and Translation

Execution, debugging, and translation tools are important components of any mature tools framework. In this section we briefly describe how these tools can be implemented as language extensions in which the host-language and language extensions productions define a specified set of attributes. There are two ways that we could implement the execution and debugging tools. The first is to build an interpreter as a collection of attributes that traverse the abstract syntax tree computing values and in the case of debugging, observing break points and reporting intermediate values to the user. The second is to translate the specification to a programming language and compile the resulting code. For debugging, we would insert additional information into the translation that a debugging tool could use. This translation model is also used for translating specifications into the input languages of various analysis tools such as NuSMV and PVS.

3.3.1 Execution and debugging by interpretation

To implement an interpreter for our specification, each construct in the host-language will define an *eval* attribute that has the following functional type: $Store \rightarrow (Value, Store)$, where a *Store* is a mapping from variables to *Value*s. We can use this attribute to build a lazy evaluator that threads a store through the abstract syntax tree and adds new values to the store as the values of variables are computed. The execution process begins with the "main" node calling the *eval* attribute on its output variable[3] giving it the initial store containing values for the input variables. As part of the evaluator we also define a set of attributes that associate with the output and each local variable a reference to its defining assignment statement node in the abstract syntax tree. The *eval* function for an identifier checks to see if its value has been computed and thus exists in the store. If it does, it returns the original store and that value. If it does not, it calls, and returns the result of, the *eval* function on its defining assignment statement. This function will call the *eval* function on its component expression and return that value and a store containing its left hand side variable and its value. The expression *eval* function will, in turn, cause the execution of the eval function on its component variables which will trigger the execution of the *eval* function on their defining assignment statements. This process continues until all the needed values are computed and the value of the output variable can be determined.

The evaluator we have sketched above can be seen as a language extension defined using the same techniques as those discussed previously. The extension specification consists of the definitions for the *eval* attribute on the host-language productions, the definition of the attributes linking variable references to their assignment, and a set of attributes that ensure that there are no data dependency cycles among the variables. This evaluator is thus a modular extension that defines a set of attributes on host-language productions and fits into the framework provided by the proposed tools infrastructure.

Language extension productions, such as those defining events, would not need to define their *eval* attributes since their evaluation would be the same as the semantically equivalent construct that they forward to. This is not the case with debugging. A language extension, like events, may want to specify its debugging behavior to be different than the debugging behavior of its semantic equivalent. For example, the user may want to break into the evaluation of their specification when a certain event is fired and to be shown the *throw* construct that was responsible for that event being fired. Thus, the *catch* and *throw* productions would provide their own definitions of attributes defining the debugger.

3.3.2 Translation

Translators, either for execution, debugging, or analysis purposes, can also be defined as a language extension consisting of a set of attribute definitions for the host-language and some language extension productions. For a translation-based evaluator, we would specify definitions for an *eval_trans* attribute on the host-language productions that specify a translation into, for example, C that can then be compiled and executed. This model is also used for translating specifications to various analysis tools such as the NuSMV model checker and the PVS theorem prover. To specify these translations, we would specify two string-valued attributes, *pvs_trans* and *smv_trans*, that on a language construct define its translation into, respectively, PVS and NuSMV. Definitions for these attributes would be required for all productions in the host-language. Thus any language extension, such as events, will have a default translation into NuSMV and PVS. For example, the PVS translation for a *throw* expression is determined by retrieving the *pvs_trans* attribute from the semantically equivalent construct that it forwards to.

For other language extension constructs, this translation through the host-language may lose information (as described in Section 2.2) and result in a degraded representation in the target language that makes the

[3] For simplicity we will assume each node has a single output variable.

analysis more expensive. Such constructs should instead provide a definition for the *pvs_trans* or *smv_trans* attributes to specify their direct "non-lossy" translation. Consider adding template types as a language extension. Template types exist in PVS but not in NuSMV or our host-language. We want our translation to PVS to maintain this information and thus the productions that specify the template type extension would provide explicit definitions for *pvs_trans* that specifies the PVS-template code that implements the language extension template types. This ensures a *high-quality*, "non-lossy" translation to PVS. Since the host-language does not support template types, the extension must specify, via a collection of attribute definitions, how each node declaration with template type parameters can be *instantiated* into several instances that do not contain template types. Each generated instance is the result of the concrete types used in a specific node call construct. This is not a trivial analysis or transformation, but it can be done with forwarding attribute grammars and is similar to the techniques used for instantiating C++ templates. These transformations create the host-language constructs that the template types forward to in the host-language. The translation of template types to NuSMV then relies on forwarding and the definition of *smv_trans* on the host-language productions. This approach avoids the pitfalls of intermediate notations described in Section 2.2.

To summarize, as we have outlined in this section, we believe attribute grammars with forwarding can be used to capture the semantics of a host-language and define language-extensions that can be combined to create new modeling-languages, as well as define how these languages are translated to various target-languages. We believe that this formalism is highly suitable as a foundation for model-based tools and that it will help us provide the high-level of flexibility, ease of change, and high quality for which we aim. Do note again, however, that we are presenting this idea to generate a dialogue in the formal methods, model-based, and static analysis communities with the goal of evolving a consensus for the architecture of the next generation of hyper-flexible modeling and analysis tools.

4 A Call to Arms

In this report we assert that a critical obstacle to the widespread adoption of modeling and analysis tools lies in the lack of flexibility to accommodate customer preferences when considering the adoption of tools. As stated in the introduction, we believe that the following conjectures describe fundamental obstacles to the widespread adoption of formal modeling and the potential for automation that comes with it; (1) no single modeling notation will suit all, or even most, modeling needs, (2) no analysis tool will fit all, or even most, analysis tasks, and (3) flexible and stable tools must be made available for realistic evaluations and technology transfer.

For automated software engineering techniques to be more useful for a wider developer audience, and to allow us to move forward as a community, we must develop a foundation for building flexible and extensible modeling language processing tools. We believe that such tools must satisfy, at a minimum, the following requirements—successful modeling tools must (1) allow easy extension and modification of formal modeling notations to meet the domain specific needs of a class of users, (2) allow easy construction of high-quality translations from the modeling notations to a wide variety of evolving analysis tools, and (3) enable controlled reuse of tool infrastructure to make tool extensions cost effective.

In an effort to spark some debate and discussion in the community, we have hypothesized that higher-order attribute grammars with forwarding can be used to build languages and tools that serve as a basis for such flexible and extensible language processing tools; tools that will allow us to unify our efforts and help bring our collective work to a broader audience. Naturally, although we like to think so, we do not presume that our direction is the best (or even practicable) and it serves only as an illustration as to what we would like to achieve in a unifying language processing framework. The aim of this report is to stimulate the discussion and, hopefully, move the community into action so we can join forces in developing the solid tools infrastructure needed to have an impact on software development practice.

References

1. S. R. Adams. *Modular Grammars for Programming Language Prototyping.* PhD thesis, University of Southampton, Department of Electronics and Computer Science, UK, 1993.
2. M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. In *User Interfaces for Theorem Provers*, 1998.
3. J. Baker and W. Hsieh. Maya: Multiple-dispatch syntax extension in java. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, June 2002.
4. D. Batory, D. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–53. IEEE, 2–5 1998.
5. S. Bensalem, P. Caspi, C. Parent-Vigouroux, and C. Dumas. A methodology for proving control systems with Lustre and PVS. In *Proc. of Seventh Working Conference on Dependable Computing for Critical Applications (DCCA 7)*, 1999.
6. S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Ruess, J. Rushby, V. Rusu, H. Saidi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. M. Holloway, editor, *Proc. of LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196. NASA Langley Research Center, June 2000.
7. J. T. Boyland. Remote attribute grammars. *J. ACM*, 52(4):627–687, 2005.

8. M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J. Krimm, L. Mounier, and J. Sifakis. IF: An intermediate representation for SDL and its applications. In *Proceedigns of the SDL-Forum99*, page 423440. Elsevier Science, 1999.
9. T. J. Cheatham. The introduction of definitional facilities into higher level programming languages. In *AFIPS (Fall Joint Computer Conference, 29)*, pages 623–637, 1966.
10. S. Chiba. A metaobject protocol for C++. In *Proc. of Object-oriented programming systems, languages, and applications*, pages 285–299. ACM Press, 1995.
11. Y. Choi and M. Heimdahl. Model checking RSML^{-e} requirements. In *Proceedings of the 7th IEEE/IEICE International Symposium on High Assurance Systems Engineering*, October 2002.
12. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model verifier. In *Proc. of International Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499. Springer-Verlag, 1999.
13. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
14. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36 – 72, January 1993.
15. W. Clinger. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–162. ACM Press, 1991.
16. K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.
17. L. Dillon and R. Stirewalt. Inference graphs: A computational structure supporting generation of customizable and correct analysis components. *IEEE Transactions on Software Engineering*, 29(2):133–150, February 2003.
18. T. Ekman and G. Hedin. Rewritable reference attributed grammars. In *18th European Conference on Object-Oriented Programming, ECOOP 2004*, volume 3086 of *Lecture Notes in Computer Science*, pages 144–169. Springer, June 2004.
19. T. Ekman and G. Hedin. The JastAdd extensible java compiler. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007. ACM.
20. T. Ekman and G. Hedin. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, December 2007.
21. T. Ekman and G. Hedin. Pluggable checking and inferencing of non-null types for Java. *Journal of Object Technology*, 6(9):455–475, October 2007.
22. T. Esterel. Corporate web page. www.esterel-technologies.com, 2004.
23. R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars. In *19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 223–234, 1992.
24. H. Ganzinger. Increasing modularity and language-independency in automatically generated compilers. *Science of Computer Programing*, 3(3):223–278, 1983.
25. H. Ganzinger and R. Giegerich. Attribute coupled grammars. *SIGPLAN Notices*, 19:157–170, 1984.
26. J. Gao, M. Heimdahl, and E. Van Wyk. Flexible and extensible notations for modeling languages. In *Fundamental Approaches to Software Engineering, FASE 2007*, volume 4422 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, March 2007.
27. A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, Nov. 1999.
28. R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25(4):355–423, 1988.
29. O. Grumberg and D.E.Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.
30. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proc. of IEEE*, 79(9):1305–1320, September 1991.
31. N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Transactions on Software Engineering*, pages 785–793, 1992.
32. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.
33. G. Hedin. An object-oriented notation for attribute grammars. In *Proceedings of the European Conference on Object-Oriented Programming, ECOOP'89*, 1989.
34. G. Hedin. Reference attribute grammars. In *2nd International Workshop on Attribute Grammars and their Applications*, 1999.
35. M. P. Heimdahl, Y. Choi, and M. Whalen. Deviation analysis via model checking. In *International Conference on Automatied Software Engineering*, pages 37–46, September 2002.
36. M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.
37. C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance, COMPASS 95*, 1995.
38. C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.
39. K. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, 6(1):2–13, Januaray 1980.
40. P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es), 1996.
41. A. Joshi, S. P. Miller, and M. P. Heimdahl. Mode confusion analysis of a flight guidance system using formal methods. In *To appear in the 22nd Digital Avionics Systems Conference*, October 2003.
42. U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.
43. G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the MetaObject Protocol*. MIT Press, 1991.
44. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Corrections in 5(2):95–96, 1971.
45. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161. ACM Press, 1986.
46. C. Le Bellec, M. Jourdan, D. Parigot, and G. Roussel. Specification and implementation of grammar coupling using attribute grammars. In M. Bruynooghe and J. Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP '93)*, volume 714 of *Lecture Notes in Computer Science*, pages 123–136. Springer-Verlag, 1993.
47. B. Leavenworth. Syntax macros and extended translations. *Communications of the ACM*, 9(11):790–793, 1966.

48. E. A. Lee. Overview of the ptolemy project. Technical Report Technical Memorandum UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2003.
49. D. Leijen and E. Meijer. Domain specific embedded compilers. *ACM Sigplan Notices, Papers of the 2nd USENIX Conference on Domain Specific Languages, 1999*, 35(1), January 2000.
50. N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.
51. N. G. Leveson, M. P. Heimdahl, and J. D. Reese. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *LNCS*, pages 127–145, September 1999.
52. MathWorks. The MathWorks Inc. web page. http://www.mathworks.com, 2004.
53. S. P. Miller, A. C. Tribble, and M. P. E. Heimdahl. Proving the shalls. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedigns of the International Symposium of Formal Methods Europe (FME 2003)*, volume 2805 of *Lecture Notes in Computer Science*, pages 75–93, Pisa, Italy, September 2003. Springer.
54. J. Niu, J. Atlee, and N. Day. Template semantics for model-based notations. *IEEE Transactions on Software Engineering*, 29(10):866 – 882, October 2003.
55. A. J. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, October 1999.
56. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proc. of Conference on Automated Deduction (CADE-11)*, number 607 in Lecture Notes in Computer Science, pages 748–752. Springer-Verlag, 1992.
57. J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995.
58. A. Poigne and L. Holenderski. On the combination of synchronous languages. In W. de Roever, H. Langmaack, and A. Pnueli, editors, *Proceedings of International Symposium on Compositionality*, volume 1536, Bad Malente, Germany, September 1997. Springer-Verlag.
59. S. Rayadurgam and M. P. Heimdahl. Coverage based test-case generation using model checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, pages 83–91. IEEE Computer Society, April 2001.
60. S. Rayadurgam, A. Joshi, and M. P. E. Heimdahl. Using PVS to prove properties of systems modelled in a synchronous dataflow language. In *Proceedigns of the 5th International Conference on Formal Engineering Methods, ICFEM 2003*, pages 167–186, Singapore, November 2003.
61. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1998.
62. C. Simonyi. The future is intentional. *IEEE Computer*, May 1999.
63. E. W. Sirko. Rif: A language and toolkit supporting research and education in optimizing compilers. Master's thesis, University of California Riverside, Riverside, California, 1999.
64. Standford SUIF Compiler Group. The SUIF parallelizing compiler guide, 1994. Stanford University. Version 1.3.
65. T. Teitelbaum and R. Chapman. Higher-order attribute grammars and editing environments. In *ACM Sigplan '90 Conference on Programming Languages Design and Implementation*, pages 197–208, 1990.
66. J. M. Thompson, M. P. Heimdahl, and D. M. Erickson. Structuring formal control systems specifications for reuse: Surviving hardware changes. In *Proceedings of the Fifth NASA Langley Formal Methods Conference (Lfm2000)*, 2000.
67. J. M. Thompson, M. P. Heimdahl, and S. P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.
68. J. M. Thompson, M. W. Whalen, and M. P. Heimdahl. Requirements capture and evaluation in NIMBUS: The light-control case study. *Journal of Universal Computer Science*, 6(7):731–757, July 2000.
69. E. Van Wyk, D. Bodin, L. Krishnan, and J. Gao. Silver: an extensible attribute grammar system. In *Proc. of LDTA 2007, 7^{th} Workshop on Language Descriptions, Tools, and Analysis*, 2007.
70. E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. Conf. on Compiler Construction*, volume 2304 of *LNCS*, pages 128–142. Springer-Verlag, 2002.
71. E. Van Wyk, O. de Moor, G. Sittampalam, I. Sanabria-Piretti, K. Backhouse, and P. Kwiatkowski. Intentional Programming: a host of language features. Technical Report PRG-RR-01-21, Computing Laboratory, University of Oxford, 2001.
72. E. Van Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute grammar-based language extensions for java. In *European Conference on Object Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science*, pages 575–599. Springer-Verlag, July 2007.
73. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher-order attribute grammars. In *Conference on Programming Languages Design and Implementation*, pages 131–145, 1990. Published as *ACM SIGPLAN Notices*, 24(7).
74. D. Weise and R. Crew. Programmable syntax macros. *ACM SIGPLAN Notices*, 28(6), 1993.
75. M. K. Zimmerman, K. Lundqvist, and N. Leveson. Investigating the readability of state-based formal requirements specification languages. In *Proc. 24th Conf. on Software engineering*, pages 33 – 43, Orlando, Florida, May 2002. ACM Press.