FLEXIBILITY IN MODELING LANGUAGES AND TOOLS: A CALL TO ARMS[1]

Eric Van Wyk and Mats Per Erik Heimdahl
Department of Computer Science and Engineering
University of Minnesota

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 in 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.

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.,* [1], model checking, *e.g.,* [2], theorem proving, *e.g.,* [3], and test case generation, *e.g.,* [4]. 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 [5], Statemate from i-Logix [6], Simulink and Stateflow from The Mathworks Inc. [7], and SpecTRM from Safeware Engineering [8]; examples of research tool are SCR [9] and RSML^{-e} [10].

Our goal has for the last decade 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-

[1] This work was partially funded by NSF CAREER Award #0347860 and NSF CCF Award #0429640.

control systems called specification-based prototyping [10]. 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) [11] and its execution and analysis environment NIMBUS [12]. We have successfully evaluated the capabilities on various case studies from the avionics, transportation, and mobile robotics domains [10, 11], and the environment has been used for several years in industrial research projects [12, 13, 14]. 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 years of experience, 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 [5], SCADE [5], and SCR [9].

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 tools 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 [15], 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.

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.

The Problem

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 [7] or SCADE [5] are suitable for the control oriented aspect of a system design whereas Safe State Machines [5] or Statecharts [6] are more suited for the discrete aspect of a system. Other notations, for example, SCR [16] and RSML^{-e} [10] 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 the customers 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 customers' 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.

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 $n \times m$ 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 [17], IF for the exchange of models between model checkers [18], and SAL for general purpose applications [19]. 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, 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 which does not allow template types, but it may be wholly inappropriate when targeting an analysis tools such as PVS which does allow template types. Because of this information loss, the translation through an intermediate language to PVS can not take advantage of all of the capabilities of the target language thus making verification in PVS more difficult and time consuming than strictly needed. On the other hand, if the intermediate language has many high-level constructs, such as template types, the

translations to the different target languages becomes 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 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.

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 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, hygienic [20], and programmable [21] macros systems and embedded domain specific languages [22] do allow new language constructs to be added to a language. However, they lack an effective way perform the necessary static analysis. On the other hand, meta-object protocol systems, *e.g.,* [23], provide limited opportunities to add new language constructs but can perform the static analyses needed to, for example, check for domain specific errors.

Attribute grammars [24] 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.,* [25, 26]. 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.,* [27]. Others are inspired by the object-oriented paradigm and employ inheritance to achieve a separation of concerns [28]. Of most use to us are higher-order attributes [29] that allow abstract syntax trees to be attribute values and reference attributes [30] that point to possibly remote nodes in the abstract syntax tree.

Most closely related to the extensible language framework described below is Microsoft's Intentional Programming system (IP) [31, 32, 33 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 crisply 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 [34], 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 [32] was its ad-hoc nature that prevented any static analysis of language extensions to test their compatibility.

A PROPOSED FRAMEWORK

In this section we will illustrate how attribute grammars extended with forwarding [34] can be use 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 the previous section, 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 [34] 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 our 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 below 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—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 extension to allow for flexibility in the modeling notation in following section. In the section on translation we show how forwarding is used to easily implement high-quality translations to many analysis engine languages while avoiding many pitfalls of intermediate languages.

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 ( )
```

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

A familiar example from programming languages will clarify. Consider adding a simple *foreach* construct to Java to iterate over Java Collections. 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 <*Stmt*> non-terminal named *for* and right-hand side non-terminals <*Type*>, <*Id*>, <*Expr*>, and <*Stmt*> 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.) It also specifies its *forwards-to* tree as the expected host-language block-loop construct built by parsing the string and using the "unquote" operator (` `) to reference right-hand side subtrees, their attributes and the identifier *iter*. The reference attribute *type* points to a node in the program abstract syntax tree defining the collection's type. Its *pp* attribute is used to cast the Java *Object*-type value returned from the iterator method *next*. The left-hand side node 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*, its Java byte code, it passes, or "forwards", that request to the "forwards-to" node, the block-loop construct, that returns the attribute's value. Thus, 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, as attributes, that here generate proper error messages. 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.

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.

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*.

```
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
```

Figure 2. A Lustre-like definition of the state variable *AltStatus*.

Lustre as a host-language:

We hypothesize that Lustre [35] 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 [36]. Second, Lustre is supported by commercial tools [5], for example, a code generator that has been qualified for use in safety critical applications, thus, making it of interest to industrial partners. Lustre may be implemented as an extensible host language by writing an attribute grammar that defines its language constructs and defines attributes that perform semantic analysis and translation. We may define an *errors* attribute similar to the one in the *foreach* example. It may also involve defining string-type translation attributes named *smv_trans*, *pvs_trans*, and others that specify how constructs are translated to various target languages.

Below we illustrate how a Lustre host language can 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 that have been well-received and shown to be relatively easy to understand and use [16, 37, 38]. Figure 3 shows a fragment of an RSML^{-e} specification of the ASW. The figure shows the definition of 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 [37]—each column of truth values represents a conjunction of the propositions in the leftmost column (*F* represents the negation of the proposition 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.

```
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
```

Figure 3. RSML^{-e} like definition of the State Variable *AltStatus*.

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* construct in Figure 3 we follow this pattern and define a set of productions to define the syntax of state variable constructs as well as 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.

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 has on its right hand side an identifier *name* (in this case *AltStatus*), a non-terminal for the possible values, its initial value *init*<*Expr*>, and an expression *expr*<*Expr*> for subsequent values. The implicit semantics of a state variable are provided by the Lustre assignment statement that it forwards to and is 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 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. We do not provide the attribute grammar specification here. 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.

Events as an extension

In Figure 4 is part of the same altitude switch, this time specified in the host-language extended with a notion of events. Here, the occurrence of the externally defined event *AltRecvEvt* is used in the computation of *AltStatusEvt*. These computations also activate (or throw) the *AltClassifiedEvt* and *AltLostEvt* events to indicate the outcome of the assignment of the *AltStatusEvt* 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.

```
var AltClassifiedEvt : Event ;
    AltLostEvt : Event ;
AltStatus = Unknown ->
 if catch AltRecvEvt and C1
 then throw AltClassifiedEvt return Above
 else if catch AltRecvEvt and C2
      then throw AltClassifiedEvt return Below
      else if catch AltRecvEvt and C3
           then throw AltLostEvt return Unknown
           else pre(AltStatus)
```

Figure 4. Event/Action style constructs added to the host-language.

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 is 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 Lustre 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.

Of particular interest is how the assignment statements for the event-emulating boolean variables are generated. In the example, the boolean variable *AltClassifiedEvt* is generated to replace the *AltClassifiedEvt* event. The assignment statement below (with catch constructs yet to be removed) is also generated to set this boolean variable to *true* under the conditions that the event was thrown.

```
AltClassifiedEvt = False -> (catch AltRecvEvt and C1) or
  (not (catch AltRecvEvt and C1) and (catch AltRecvEvt and C2))
```

The *AltClassifiedEvt* boolean variable is naively computed to be *true* if the first if-condition (in Figure 4) 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-condition's that must be *true* to cause that throw instance to "fire." Again, we leave out the implementation of this since a *inherited* higher order attribute can be specified to pass down the enclosing if condition <*Expr*>-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 [37]. 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 modeling-language rather than in the host-language. For example, 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, all 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. 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 provide all of the capabilities that one expects from "built-in" language features.

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 background section) and result in a degraded representation in the target language that makes the 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 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 follows the techniques 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 above in the background section.

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.

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. 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.

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 a foundation for building extensible and flexible modeling language processing tools. 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 tools extensions cost effective.

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. 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. 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.

2. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

3. 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.

4. A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, Nov. 1999.

5. T. Esterel. Corporate web page. www.esterel-technologies.com, 2004.

6. 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 Engineer*ing, 16(4):403–414, April 1990.

7. MathWorks. The MathWorks Inc. web page. http://www.mathworks.com, 2004.

8. 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 *Lecture Notes in Computer Science*, pages 127–145, September 1999.

9. 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.

10. 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*, volume 1687 of *Lecture Notes in Computer Science*, pages 163–179, September 1999.

11. 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.

12. S. P. Miller, A. C. Tribble, and M. P. E. Heimdahl. Proving the shalls. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedings 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.

13. A. Joshi, S. P. Miller, and M. P. Heimdahl. Mode confusion analysis of a flight guidance system using formal methods. In *Proceeding of the 22nd Digital Avionics Systems Conference*, October 2003.

14. 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.

15. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1998.

16. K. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, 6(1):2–13, January 1980.

17. 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 of *Lecture Notes in Computer Science*, Bad Malente, Germany, September 1997. Springer-Verlag.

18. 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-Forum'99*, page 423–440. Elsevier Science, 1999.

19. 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.

20. 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.

21. D. Weise and R. Crew. Programmable syntax macros*. ACM SIGPLAN Notices*, 28(6), 1993.

22. P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es), 1996.

23. S. Chiba. A metaobject protocol for C++. *In Proc. of Object-oriented programming systems, languages, and applications*, pages 285–299. ACM Press, 1995.

24. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127--145, 1968. Corrections in 5(2):95–96, 1971.

25. H. Ganzinger. Increasing modularity and language-independency in automatically generated compilers. *Science of Computer Programming*, 3(3):223–278, 1983.

26. U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.

27. D. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *The Computer Journal*, 33(2):164–172, 1990.

28. G. Hedin. An object-oriented notation for attribute grammars. In *Proceedings of the European Conference on Object-Oriented Programming, ECOOP'89*, 1989.

29. 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).

30. G. Hedin. Reference attribute grammars. In *2nd International Workshop on Attribute Grammars and their Applications*, 1999.

31. C. Simonyi. The future is intentional. IEEE *Computer*, May 1999.

32. 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.

33. K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

34. E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proceedings of Conf. on Compiler Construction,* volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer-Verlag, 2002.

35. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. In *Proc. of IEEE,* 79(9):1305–1320, September 1991.

36. 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.

37. 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.

38. 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.