Building Extensible Specifications and Implementations of Promela with AbleP

Yogesh Mali and Eric Van Wyk*

Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN 55455, USA
yomali@cs.umn.edu, evw@cs.umn.edu

Abstract. This paper describes how new language features can be seamlessly added to an extensible specification of Promela to provide new (domain-specific) notations and analyses to the engineer. This is accomplished using ABLEP, an extensible specification and implementation of Promela, the modeling language used by the SPIN model checker. Language extensions described here include an enhanced select-statement, a convenient tabular notation for boolean expressions, a notion of discrete time, and extended type checking. ABLEP and the extensions are developed using the SILVER attribute grammar system and the COPPER parser and scanner generator. These tools support the modular development and composition of language extensions so that independently developed extensions can be imported into ABLEP by an engineer with little knowledge of language design and implementation issues.

1 Introduction and Motivation

Modeling languages such as Promela, Lustre, Esterel, and others, allow engineers to specify problem solutions in an abstract high-level declarative language that more closely matches the problem domain than low-level programming languages such as C. These languages are typically designed to enable some sort of analysis of the specifications written in them. The Promela model checker SPIN [10, 9] takes advantage of the high-level constructs, such as processes and guarded-statements, and the restrictions in Promela, such as the absence of floating point numbers, to efficiently analyze and verify properties on Promela programs.

Yet, modeling languages often lack linguistic support for some commonly-used general purpose and less commonly-used domain-specific concepts; this leaves the engineer to encode them as idioms, a time-consuming and error-prone process. This is not necessarily the fault of the language designers. It is a problem for them as well as they must try to strike a balance in determining which features to include in the language, weighing the benefits of the feature against the cost of implementing, maintaining, and documenting a larger language.

* This material is based on work partially supported by NSF Awards No. 0905581 and No. 1047961 and DARPA and the United States Air Force (Air Force Research Lab) under Contract No. FA8650-10-C-7076.

A few examples may illustrate these points. Consider the `select` statement added to Promela in SPIN version 6 that non-deterministically assigns a value within a specified range to a variable. For example, `select (v : 1..10)` will assign a value between 1 and 10 to the variable `v`. In previous versions of Promela, engineers had to "code this up" using an idiom of several (non-guarded) assignments as options in an `if` statement. Presumably the perceived need of this features outweighed the costs of adding it to the language. This feature and the new `for` loop are added in the Promela parser and expanded into Promela `do`-loops a the idioms engineers have typically used previously to express them.

As an *domain-specific* example, consider the discrete-time features introduced in DTSPIN [1]. A new `timer` datatype and operations on values of this type are provided. These features can be implemented as simple C pre-processor macros, a very lightweight approach to language extension. Another approach modifies the SPIN source code to provide a more efficient implementation of an underlying clock-tick operation, a rather heavyweight solution. Domain-specific extensions like this are often very useful, but to a smaller group of users than may justify their direct inclusion in the language.

A third example is the type analysis provided by ETCH [4]. This tool provides a constraint based type inference system to detect type errors in Promela specifications that have not always been detected by SPIN. Finding such errors statically saves engineers time. This tool implements its own scanner and parser for Promela, using SableCC [7], to build an abstract specification on which type checking is performed. Developing a new scanner and parser for Promela is a non-trivial task that detracts from the developer's main aim of type checking.

This paper describes a *middle-weight* solution to these language evolution and extensions problems. ABLEP (Attribute grammar-Based Language Extensions for Promela) is an extensible language framework that lies between the lightweight solution of simple macros and the heavyweight solutions of modifying the SPIN code base or building an entirely separate system for parsing and analysing Promela specifications. ABLEP allows new language features to be specified in a highly modular manner so that engineers can easily select the language extensions that they desire for a particular problem and seamlessly import them into ABLEP. These features may be new language constructs that add new syntax to the *host language* Promela. They may also be new semantic analyses of these new constructs or new semantic analyses to the Promela constructs. For frameworks such as ABLEP to be useful to a wide audience we believe that new features, packaged as *language extensions*, must be easily imported by an engineer or programmer without requiring significant knowledge of language design and implementation techniques. Thus, while extension *developers* need this sort of knowledge, extension *users* need not; users simply direct the framework to *import* the desired set of extensions. This is not unlike importing libraries in traditional programming languages. In such frameworks, the engineer imports a set of extensions to create a customized language, and its supporting analysis and translation tools, that address the specific programming or modeling task at hand.

Section 2 of this paper describes several language extensions in ABLEP and shows how an engineer uses ABLEP and imports extensions into it. These extensions include aspects of those described above as well an extension that provides a tabular representation for complex boolean expressions. Section 3 shows how ABLEP and these modular extensions are implemented using the SILVER attribute grammar system [14] and the COPPER parser and context-aware scanner generator [16]. It also describes how these tools support the seamless and automatic composition of extensions designed by independent parties. Section 3.3 describes how these tools handle some challenging aspects of the Promela language including inlining and embedded C code. Section 4 describes related work and section 5 discusses the challenges faced by, and merits of, ABLEP.

2 Use of AbleP and its language extensions

In Section 2.1 we describe some sample language extensions to Promela and show how they are used in the ABLEP extensible language framework in which Promela plays the role as the host language. In Section 2.2 we show how an engineer can import a chosen set of extensions into ABLEP and generate a processor for the custom language with the features that suit his or her specific needs.

2.1 Using an AbleP extended language framework

In this section we describe a set of ABLEP extensions that add an enhanced `select` statement, tabular boolean expressions, type-checking analysis in the spirit of ETCH, and features for discrete time based on DTSPIN. A model that uses these features is shown in Fig. 1. This extended language model is based on a sample altitude switch for an aircraft that computes an altitude status value of *Unknown*, *Above*, or *Below* from a simulated altitude reading, an indication of confidence in the altitude readings, an altitude threshold, and a hysteresis value.

An *instantiation* of ABLEP, the framework extended with a specific set of extensions, is used by the engineer to analyze a model and translate it down to its semantically equivalent model in pure Promela to be used by the SPIN model checker. Thus, ABLEP instantiations are essentially sophisticated pre-processors that go beyond the capabilities of simple macro pre-processors by performing syntactic and semantic analysis on the model to detect certain (domain-specific) errors and direct the translation down to pure Promela. ABLEP processors are generated by the SILVER [14] and COPPER [16] tools from the specifications of the host language and chosen extensions. These tools are Java-based and the generated instantiation is realized as a Java `.jar` file. In Section 2.2 we show how the engineer creates these, but here we assume that this has been done, resulting in a instantiation stored in `ableJ.aviation.jar`. The engineer need only provide the name of extended-language model to the processor, in this case `AltSwitch.xpml`, which checks for semantic errors and generates the translation down to pure Promela stored in a file with a `.pml` extension, here `AltSwitch.pml`. This can be processed by SPIN as the engineer would normally.

```
mtype = {Unknown, Above, Below } ;    /* altitude status values */
mtype = {High, Med, Low} ;            /* quality of instrument readings */

chan startup = [0] of {int, int} ;
chan monitor = [0] of {mtype, mtype} ;

timer trouble_t, above_t ;          /* expands to "int trouble_t = -1;" */

active proctype determineStatus ()
{
 int altitude, threshold, hyst ;
 mtype altQuality, altStatus = Unknown ;

 startup ? threshold, hyst;  /* receive threshold and hysteresis values */
 run monitorStatus ( );      /* start monitoring process */

 check:
  /* select, non-deterministically, values for altitude and altQuality  */
  select (altitude: 1000 .. 10000 step 100) ;
  select (altQuality: High, Med, Low) ;

  /* use condition tables to assign a value to altStatus */
  if :: tbl altStatus == Unknown        : T *
           altQuality == High           : T T
           altitude > threshold         : T T
           altitude > threshold + hyst : * T
       lbt -> altStatus = Above ;
     :: tbl altQuality == High   : T
           altitude > threshold : F
       lbt -> altStatus = Below ;
     :: else; altStatus = (altQuality == High -> Unknown : altStatus) ;
  fi ;

  if :: altStatus == Above -> goto above;
     :: altStatus == Below || altStatus == Unknown -> goto trouble;
  fi ;

 above:
   delay(above_t,1);                  /* delay until next "tick" */
   goto check ;

 trouble:
   monitor!altStatus,altitude ;        /* send msg to monitor     */
   trouble_t = 1; expire(trouble_t);   /* what delay expands to   */
   goto check ;
}
```

Fig. 1. The extended-Promela program `AltSwitch.xpml` processed by ABLEP.

The primary goal of the model in Fig. 1 is to compute a value for the `altStatus` variable and assign to it a value from the `mtype` values of `Unknown`, `Above`, and `Below` specified on the first line. The quality of the instrument readings is represented by the second `mtype` declaration of the values `High`, `Med`, and `Low`. Next, a `startup` channel is declared and used to receive the integer threshold and hysteresis values. A `monitor` channel is declared; it is used to send the status and altitude values to a monitoring process `monitorStatus` (whose definition is not shown here). Skipping for now the declarations of variables of type `timer`, the `determineStatus` process declares its local variables `altitude`, `threshold`, `hyst`, `altQuality` (which takes values of `High`, `Med`, and `Low`), and `altStatus` (which takes values of `Unknown`, `Above`, and `Below`). The process `determineStatus` first receives values for the threshold and hysteresis variables and starts the `monitorStatus` process.

Enhanced select statements: The first use of a language extension follows the `check` label. This enhanced `select` statement is an extended version of the one introduced to Promela in SPIN version 6. Here, we've added a `step` value of 100. This statement non-deterministically selects an altitude value in the range beginning at 1000, increasing by values of 100, and ending at 10000. The second example non-deterministically picks a value for altitude quality from one of the three valid `mtype` values of `High`, `Med`, or `Low`.

ABLEP supports language extensions that have the same look-and-feel as built-in constructs. Thus, their syntax should be natural and they should report error messages based on analysis of the extension, not on its translation to pure Promela. While at least the first extended-select could easily be specified as a CPP macro, as is commonly done in Promela models, we choose not to do this in order to support this semantic analysis. Consider changing the second expression `Med` in the second `select` statement to `startup`, which has type `chan`. The language extension will report the following error message:

```
Error: select statement requires all possible choices to have the
 same type as variable assigned to, which is "mtype":  line 17.
```

This ABLEP extension translates this second `select` statement to a semantically equivalent construct in pure Promela; in this case that phrase is

```
if ::altQuality=High; ::altQuality=Med; ::altQuality=Low; fi;
```

The first `select` statement can be translated to a `do` loop in which the variable `altitude` is initially set to the lower bound of 1000 and is incremented by the step value of 100 until the upper bound is reached or the loop non-deterministically exits, which is shown in Fig. 2(a). The extension, however, inspects the lower bound, upper bound, and step expressions. If they are all constants (known statically), then a (possibly large) non-deterministic `if` statement can be generated instead. Part of the translation that is generated for the first `select` statement is shown in Fig. 2(b). An advantage of this translation is that during interactive simulation with SPIN of the generated Promela model,

```
altitude = 1000 ;
do  :: goto l30 ;
    :: (altitude < 10000) ;
       altitude = (altitude + 100) ;
od ;
l30: skip ;
```

(a)

```
if :: altitude = 1000 ;
   :: altitude = 1100 ;
   :: altitude = 1200 ;
   :: altitude = 1300 ;
   ...
fi ;
```

(b)

Fig. 2. Two translations of `select` to pure Promela.

the user is asked to pick a transition that will select the desired value without having to step through the loop the appropriate number of times.

These extensions are straightforward and primarily introduced as an initial example. The first could be specified as a do-loop-generating CPP macro, but the second requires a bit more processing. We see how this is done in Section 3.

Boolean Table Expressions: In Fig. 1, after values for `altitude` and `altQuality` have been selected a guarded `if` statement assigns a value to `altStatus`. The guards on the first two options are boolean expressions represented in a tabular form. Due to the complexity of these conditions, the engineer may decide to use *condition tables*, such as those found in RSML^{-e}. These are sometimes useful when reviewing the models with domain experts. These tables, indicated with keywords `tbl` and `lbt`, consist of rows with leading boolean expressions followed by a list of "truth values": `T` for true, `F` for false, and `*` for "don't care". These specify the required truthfulness of the expression at the beginning of the row. All rows must have the same number of truth values. The value of a table is determined by taking the disjunction of the boolean value computed for each column. This column value is the conjunction of checking that each expression has the required truth value. For example, the first column in the table is true if all expression are true except for the last one, which can take any value. The first table is translated to the following pure Promela expression.

```
(((altStatus == Unknown) && ((altQuality == High) &&
 ((altitude > threshold) && true))) ||
(true && ((altQuality == High) &&
 ((altitude > threshold) && (altitude > (threshold + hyst))))))
```

Besides generating this translation, the extension also checks that each expression has an appropriate boolean type and that the rows have the same number of truth values, something not easily done on the generated expression. This extension is based on an extension that we originally developed for Lustre [8].

Extended Type Checking: Donaldson and Gay created a tool to perform extended type checking of Promela models called ETCH [4]. It adds no syntactic constructs to Promela but adds a sophisticated typing analysis. If we modify the `run monitorStatus()` statement in the `determineStatus` process to pass

in some number of arguments we would create a semantic error that SPIN would detect. If we made additional use of channels, specifically passing them as parameters to processes, their types become more difficult to check. This is because channel types are specified simply as `chan` which does not indicate the types of values to be passed along it. SPIN does not detect these types of errors statically, only at simulation or verification time. ETCH, however, includes an static type inference and checking analysis to find type errors such as those that SPIN does not. When ETCH was developed it detected the simple typing errors, such as incorrect number of arguments, that SPIN has now been extended to detect.

ETCH implements its own scanner and parser for Promela, using the SableCC compiler toolkit, in order to generate the representation on which its typing analysis can be performed. This effort of implementing the scanner and parser can be avoided in ABLEP. Instead one can specify the typing analysis as a *semantic* extension to ABLEP that works over the abstract syntax tree of the Promela model that is generated by the scanner and parser in ABLEP. Such an extension can then contribute messages describing any errors that it detects to the error-reporting facility in ABLEP. In Section 3 we will show this is done.

The language extension described here is also, in a sense, separate from the host language implementation of Promela in ABLEP, and it illustrates how analysis of this sort can be done as a composable language extension. It does not, however, implement the sophisticated type inference and checking algorithms of ETCH, but instead provides a simpler type checking facility.

Discrete Time in SPIN*:* With DTSPIN, Bosnacki and Dams introduce the notion of discrete time into Promela [1]. Discrete time occurs in many domains and the correctness of the system may include timing aspects. This is often the case in communication protocols. DTSPIN introduces a `timer` type and operations on values of this type. These include a `set(`t`,`v`)` operation to set timer t to integer value v, an `expire(`t`)` operation that waits until timer t counts down to 0, and a `delay(`t`,`v`)` operation that is syntactic sugar for `set(`t`,`v`); expire(`t`)`. Bosnacki and Dams define these using simple macros as follows:

```
#define timer int
#define set(tmr,val) (tmr=val)
#define expire(tmr) (tmr==0)
#define delay(tmr,val) set(tmr,val); expire(tmr)
#define tick(tmr) if :: tmr>=0 -> tmr=tmr-1 :: else fi
proctype Timers() {do ::timeout -> atomic{tick(t1); tick(t2)} od}
```

Note that the `Timers` process must be modified in each model to apply the `tick` operation to all timers on a `timeout`.

We've implemented similar constructs as a modular language extension to ABLEP to address some of the shortcoming of simple macros. Fig. 1 makes use of a few of these. The model declares two timers, `above_t` and `trouble_t`, and uses them after the `above` and `trouble` labels respectively. In the first, `delay` is used to wait for the timer to proceed one tick of the clock and in the second the `set`/`expire` pair is used instead, but `set` is done using a type-safe assignment.

The extension overloads assignment to allow for the case of a timer variable on the left and an integer value on the right. Otherwise the extension treats timers as separate types from numeric ones and does not allow coercions between them. The timer values are represented by integers in the generated pure Promela model, but only after this type checking has been done to prevent timers from mistakenly being used as integers.

The extension also initializes timer values to -1 (as done in DTSPIN). Global declarations and local declarations in the initial declaration section of a process are translated into initializing integer declarations (*e.g.* `int trouble_t = -1` in Fig. 1). Local declarations after this section are translated to declaration/assignment statement pairs (*e.g.* `int t; t = -1`).

It is straightforward for the extension to generate the `Timers` process to "tick" the globally-declared timers; note that it is not present in Fig. 1. For locally declared timers it is not so simple and the current extension assumes that engineers will handle them explicitly. A possible solution, however, is to represent local timers differently, essentially lifting them a global array of timers and overloading references to them to access this global array. We are currently investigating this possibility.

Our extension does not, however, address the efficiency issues that are solved by modifying the SPIN source code to implement the clock *tick* operator directly. But as it stands the extension does provide some benefits over a plain macro based implementation and may be sufficient for some purposes. Extensions of the type possible in ABLEP may provide a good "proving grounds" for ideas before taking on the larger effort of moving them into the host language. For discrete time, the best solution may be to do what DTSPIN has done and modify the SPIN source code.

Extension utility: Note that we are not making claims about the utility of these specific extensions. The papers that introduced these features do a fine job at that. Our goal here is to demonstrate that such features can be specified in a modular and composable manner in ABLEP and then easily imported and used by the engineer.

2.2 Extending AbleP with independently developed extensions

A distinguishing feature of extensible language frameworks such as ABLEP is that it is quite straightforward to create new language processors and translators by combining the host language specifications with those of the chosen language extensions. The underlying tools, SILVER and COPPER, have a number of features which enable this high degree of modularity and ensure that compositions will be well-defined. Fig. 3 shows the complete specification that an engineer would need to write to compose the host language with the four extensions described above and used in the example in Fig. 1.

The first line of this specification names the grammar and indicates that this instantiation is in the `artifacts` directory in ABLEP and has the name `aviation`, to indicate, perhaps, that this combination of features is suitable for

```
grammar edu:umn:cs:melt:ableP:artifacts:aviation ;

import edu:umn:cs:melt:ableP:host ;
import edu:umn:cs:melt:ableP:extensions:tables ;
import edu:umn:cs:melt:ableP:extensions:enhancedSelect ;
import edu:umn:cs:melt:ableP:extensions:typeChecking ;
import edu:umn:cs:melt:ableP:extensions:discreteTime ;

parser aviationParser :: Program_c {
 edu:umn:cs:melt:ableP:host ;
 edu:umn:cs:melt:ableP:extensions:tables ;
 edu:umn:cs:melt:ableP:extensions:enhancedSelect ;
 edu:umn:cs:melt:ableP:extensions:discreteTime ;       }

function main  IOVal<Integer> ::= args::[String] mainIO::IO
{ return driver (args, aviationParser, mainIO) ;  }
```

Fig. 3. SILVER specification of the avionics-inspired ABLEP instantiation.

avionics applications. Below this `import` statements indicate the host language and the set of extensions that are to be combined by SILVER to form the semantic analysis and translation of the extended language. Below this, a parser (`aviationParser`) is defined and includes the host grammar and extensions that add new language constructs (new concrete syntax) to the extended language. Since the type-checking extension does not add new constructs it need not be listed, though doing so causes no problems. Finally, a `main` function is provided that calls the `driver` function that controls the ABLEP translation process. To generate the `ableP.aviation.jar` file used above the engineer needs to run the following on the command line.

```
% silver -o ableP.aviation.jar    \
         edu:umn:cs:melt:ableP:artifacts:aviation
```

Similar files for other instantiations have the same simple, boiler-plate format. There is little here except the naming of the host language and the desired language extensions. We have considered extensions to SILVER that would simplify this so that the start nonterminal in the grammar (`Program_c`), the parser name, and the main function need not be specified but can easily be generated. We choose not to use them here in order to show, in a plain SILVER specification, that there is no needed glue code or other specifications that require any language processing skills on the part of the engineer.

3 Implementing AbleP and its language extensions

In our approach to extensible language frameworks, the instantiations are generated by composing the specifications of the host language (Promela) and the

the user-chosen language extension specifications. The two primary challenges are composing concrete syntax specifications in order to generate a scanner and parser for the extended language, and composing semantic specifications to generate the part of the instantiation the performs semantics analysis of the model and that generates the translation of the extended model down to pure Promela. To address these challenges we developed Copper [16], a parser and context-aware scanner generator, and Silver [14], an extensible attribute grammar system.[1] As we saw above, Silver reads a file containing a specification, follows the `import` statements to collect all the grammars that are to be included, and generates the files needed by Copper to create the scanner and parser.

3.1 Specification and composition of syntax

Copper specifications are context free grammars in which the terminal symbols have associated regular expressions; from these the parser and scanner are generated. The generated parser is slightly modified LALR(1) parser that uses the *context-aware scanner* [16] that is generated from the regular expressions. These specifications, in essence, contain the same information as found in specifications to popular parser generators such as Yacc or Bison and scanner generators such as Lex — though the information is processed in a slightly different manner to address challenges in parsing extensible languages.

Consider first the concrete syntax specification for the select-from construct used to assign to `altQuality` in Fig. 1:

```
s::Special_c ::= sl::'select' '(' v::Varref_c ':' es::Exprs_c ')'
{ s.ast = selectFrom (sl, v.ast, es.ast) ; }
```

This production's left-hand-side nonterminal, `Special_c`, is a type of statement. On the right are the select keyword, some punctuation, and the variable reference and expression list from which the value is chosen. In Silver productions the symbols are named; the names precede the `::` operator (read "has type") which precedes the type assigned to the name. The types are the terminal and nonterminal symbols found in the grammar. The name `sl` is used for the *select* keyword terminal defined in the ableP host language grammar and `es` is used for the comma-separated list of expressions that are derived from the host language nonterminal `Exprs_c`. This production is defined in the extension grammar `edu:umn:cs:melt:ableP:extensions:enhancedSelect` but uses only terminals and nonterminals defined in the host language grammar.

The semantics associated with the production build the abstract syntax tree (AST) for the construct using the extension-introduced abstract production `selectFrom` and attribute `ast`. The `selectFrom` parameters are the select-keyword terminal and the ASTs taken from the variable reference `v` and the list of expressions `es`. Below we will see how this abstract production detects errors on the construct and translates it down to a pure Promela guarded-if statement.

[1] Both are licensed under the LGPL open source license and distributed in both source and executable formats (as Java `.jar` files). See `http://melt.cs.umn.edu`.

To see where context-aware scanning has an impact consider the following terminal declarations and concrete productions for the `for` loop defined in the host language grammar (the semantics are elided).

```
terminal IN  'in' ;
terminal FOR 'for' lexer classes {promela_kwd} ;
fp::ForPre_c ::= f::FOR '(' v::Varref_c   { ... }
fp::ForPost_c ::= '{' s::Sequence_c os::OS_c '}'  { ... }
s::Special_c ::= fpre::ForPre_c ':' low::Expr_c '..' upp::Expr_c ')'
                 fpost::ForPost_c  { ... }
s::Special_c ::= fpre::ForPre_c in::IN v::Varref_c ')'
                 fpost::ForPost_c  { ... }
```

These productions are the same as the ones that appear in the Yacc grammar `spin.y` in the SPIN distribution. Of special interest is the reserved keyword `in` shown in the last production above. As of SPIN version 6, `in` may not be allowed as an identifier. This is because the (hand-coded) SPIN scanner does not take context into account when processing the string "`in` ..." and thus simply always treats `in` as a keyword. Context-aware scanning lets us specify a scanner and parser in which the phrase `in` can be treated as the keyword in the context of a for-loop and as an identifier in other contexts. (Since ABLEP generates Promela we need to rename identifiers named `in` for SPIN to properly handle them.)

In COPPER this notion of context comes from the LR parser state. Each state associates an action (*shift*, *reduce*, *accept*, or *error*) with each terminal symbol of the (composed) grammar. When the context-aware scanner is called to retrieve the next token it is passed the set of terminals which are valid in the current state (those with action *shift*, *reduce*, or *accept*, but not *error*). The scanner will only return tokens for terminals in this set. If the parser is in a state where it has so far matched input derivable from the nonterminal `ForPre_c` then, as seen from the last production above, the terminal `IN` is valid. Since the identifier terminal (`ID`) is not valid in this state the scanner will return the keyword `IN` token and not the identifier `ID` token. On the other hand, the terminal `FOR` is specified as being a keyword and thus has lexical precedence (in the usual sense) over identifiers. Since both identifiers and the keyword `FOR` are valid at the beginning of a statement we must reserve `FOR` as a keyword to prevent lexical ambiguities.

This use of context in the scanner means that there are fewer lexical conflicts and we can use different terminals for overlapping regular expressions. In our experience this makes it easier to write a grammar that stays within the class of LALR(1). This leads to the modular determinism analysis [11] that allows extension developers to "certify" the concrete syntax of their extensions against the host language grammar to determine if the extension can be safely combined with any other "certified" extensions (those that also pass this analysis). Thus, when the engineer selects only extensions whose concrete syntax passes this analysis he or she has a guarantee that the composed grammar containing the host language grammar and the extension grammar fragments will have no conflicts (so that an LALR(1) parser can be generated from it) and no lexical ambiguities (so that a deterministic scanner can be generated from it).

The concrete syntax for the enhanced select statement and the discrete time extensions are syntactically quite simple. The boolean tables example provides a more interesting example. The productions themselves are straightforward, and thus not shown, but it shows how non-trivial new sub-languages can be embedded into the host language. The enhanced-select does not pass this analysis, since it uses the same keyword at the beginning of the production as others in the host language. The other two do pass the analysis, however. Previous papers on context-aware scanning [16] and the modular determinism analysis [11] provide a more detailed description of context-aware scanning as used by Copper.

3.2 Specification and composition of semantics

Silver [14] is an extensible attribute grammar system with features that enable the composition of semantic analysis and translation specifications when they are specified as attribute grammars. It has many modern attribute grammar features (higher order, reference, and collection attributes, forwarding and aspect productions) borrowed from the attribute grammar literature, see [14] for details. It also has features from functional programming languages (parametric polymorphism and pattern matching).

In an attribute grammar (AG) a context-free grammar is used to define the abstract syntax. The nonterminals of the grammar are decorated with attributes (not unlike fields in records or objects) that are computed to store some semantic information for the AST. For example, a nonterminal may be decorated with an *errors* attribute, for example, that contains a list of semantic errors detected on a node (of that nonterminal type) or its descendants in the AST. Productions are seen as tree-constructing functions that specify equations for computing the values of the attributes defined on the constructed tree's root node and the nodes of its immediate children.

Consider the assignment production in the host language as shown below:

```
abstract production defaultAssign
s::Stmt ::= lhs::Expr rhs::Expr
{ s.pp = lhs.pp ++ " = " ++ rhs.pp ++ " ;\n" ;
  lhs.env = s.env;   rhs.env = s.env;     s.defs = emptyDefs();
  s.errors := lhs.errors ++ rhs.errors ;    }
```

The `abstract` production, named `defaultAssign`, has a statement nonterminal (`Stmt`) on the left and on the right has two expression nonterminals (`Expr`) for the left and right hand side of the assignment operator, which is abstracted away here. The pretty-print string attribute `pp` is defined on an assignment as the concatenation (`++`) of the `pp` attribute values on the children with the expected punctuation. This is a *synthesized* attribute as its value is synthesized from values on child nodes. The environment attribute `env` plays the role of a symbol table and passes bindings on names to their declarations down the tree; such attributes are called *inherited*. Since assignments do not alter the environment the value passed to the statement (`s.env`) is copied to its children. A corresponding synthesized attribute, `defs`, collects declarations to populate the `env` attribute,

but since assignments do not declare names this attribute is the empty set of definitions. Any errors that occur on the children (such as an undeclared variable) are passed up the tree in the synthesized attribute `errors`. Recall the type checking is done in the `typeChecking` extension (described below) so no type errors are added here. The operator `:=` is used for collection attributes and is described below; for now it can be seen as the same as `=`. Attribute evaluation is the process of computing the values for attributes from these equations.

The abstract syntax for the host language in ABLEP is defined in SILVER using productions similar to the one shown above. It implements the scope rules as found in SPIN version 6 to bind variable uses to their declarations using reference attributes. These can be seen as pointers to remote nodes in the tree. The `env` attribute maps names to references to variable declarations; variable references look up their name in the `env` attribute to get the reference, if it exists, to their declaration. If it is not found an error is placed into the `errors` attribute and propagated up the tree to be reported by ABLEP.

The attribute grammar specifications of the extensions are easily composed with the host language specification since the specifications are seen as sets: sets of nonterminals, sets of terminals, sets of productions, sets of attribute equations on productions, etc. Thus, simple set union over these sets from the host and extension creates the composed language. In the case of the enhanced-select extension, it defines new productions for the concrete and abstract syntax of the new statements. These are then composed with those in the host language to create an extended grammar. These productions do basic type checking on the new constructs and thus extend both the host language and the type checking extension, which we thus discuss first.

Type Checking: The type checking extension performs basic type checking of Promela models and makes use of the variable-use-to-declaration binding that is done in the host language. It is certainly more natural, and common, to consider this as part of the host language. Here we have pulled it out as an extension to illustrate that tools such as ETCH can be implemented as extensions in ABLEP.

This extension adds no new concrete syntax. Instead it adds new attributes that decorate existing host language nonterminals and new attribute equations to provide values for these attributes. Specifically, a `typerep` attribute decorates expressions and declarations to represent the type of the expression and the type of the variable declared on the declaration. In Fig. 4 *aspect productions* are provided for the abstract productions `varRef` and `defaultAssign` that exist in the host language. These allow new attributes to be defined on existing productions. On the `varRef` production, the `typerep` of the variable is found by looking up its declaration in the environment and getting its type from it. This code is elided for space reasons, but is straightforward. On `defaultAssign`, the `typerep` attribute on the child expressions is used to determine if the types are compatible. If they are not, it adds a new error to the list of those in the `errors` attribute, defined in the host language. In the host language, `errors` is defined as follows:

```
synthesized attribute errors :: [ Error ] with ++ ;
```

```
grammar edu:umn:cs:melt:ableP:extensions:typeChecking ;
synthesized attribute typerep::TypeRep occurs on Expr, Decls ;
aspect production varRef
e::Expr ::= id::ID
{ e.typerep = ... retrieve from declaration found in e.env ... ;   }

aspect production defaultAssign
s::Stmt ::= lhs::Expr rhs::Expr
{ s.errors <- if isCompatible(lhs.typerep, rhs.typerep) then [ ]
              else [ mkError ("Incompatible types on assigment ...") ];  }
```

Fig. 4. Partial SILVER specification of simplified ETCH-like error checking.

This attribute is a list of `Error` values that are constructed uses in `mkError` function, the details of which are not important here. What matters is that this is a *collection attribute* [2], as indicated by the `with ++` clause. This specifies that when aspect productions contribute additional values, using the `<-` assignment operator, they are combined with the base value for errors, specified using the `:=` operator that we saw above. These different values are then folded together using the `++` operator, which is both string and list concatenation. Any errors that this aspect production finds will be folded into the errors defined in the host language. Thus, it is important that the host language define `errors` as a collection attribute so that the extensions can "plug into it" to provide additional or improved semantic analysis on the model.

While this extension does not perform the same sophisticated analysis that is done by ETCH, it demonstrates how such analyses can be implemented as modular and composable language extensions. By doing so, such analyses do not need to be implemented as stand alone tools that require building their own scanner and parser, as ETCH has done.

Enhanced select and boolean tables: The enhanced-select extension provides two new versions of `select`, but we discuss just the select-from version that was used to assign to `altQuality`. Its concrete syntax specification was shown above; its abstract syntax production is given in Fig. 5. This (new) production defines its pretty print attribute `pp` as expected and computes type errors based on the `typerep` attributes introduced in the type-checking extension. This check is straightforward and more verbose than interesting, and thus elided here. Since this is a new production, not an aspect, we see that the definition of `errors` uses the `:=` operator to assign the base value for errors, consisting of errors on the children `v` and `es` and errors detected on the select statement itself.

What is of specific interest here is the "`forwards to`" clause. This clause specifies a tree (of the same nonterminal type as on the left hand side of the production) that is defined as being *semantically equivalent* to the "forwarding" tree. When the forwarding tree (node `s`, here) is queried for an attribute for which the production does not have a defining equation, that query is passed to

```
grammar edu:umn:cs:melt:ableP:extensions:enhancedSelect ;
abstract production selectFrom
s::Stmt ::= sl::'select' v::Expr es::Exprs
{ s.pp = "select ( " ++ v.pp ++ ":" ++ es.pp ++ " ); \n" ;
  s.errors := v.errors ++ es.errors ++
     if  ... check that all expressions in 'es' have same type as 'v' ...
     then [ mkError ("Error: select statement requires ... ") ]
     else [ ] ;
  forwards to ifStmt( mkOptions (v, es) ) ;        }
```

Fig. 5. SILVER specification of the semantics of the select-from extension.

the forwards-to tree. In the case of the assignment to `altQuality` from above, the forwards-to tree is the `if` statement with three options, one for each possible assignment, that was shown above in Section 2.1. The `mkOptions` function builds the options for the `if` statement from the variable reference and expressions.

Consider removing the equation defining `errors` from the `selectFrom` production. If this were done, the query for the `errors` attribute would be forwarded to the semantically equivalent `if` statement. This means that the user gets error messages from code that he or she did not write, even though they would be semantically correct. The boolean tables extension follows this same pattern. Abstract productions for tables and the component rows perform type checking and compute the less-readable semantically equivalent boolean expression of `&&` and `||` operators that the table will forward to. In this case however, providing a definition of the `errors` attribute is more critical since from the translation down to the equivalent boolean expression it may not be able to detect when the rows in the table are of different lengths. Forwarding provides a mechanism for *implicitly* specifying the semantics (attributes) for a new construct when *explicit* specifications for them are not provided by the production.

Discrete Time: Collection attributes and forwarding as described above play important roles in defining the discrete-time language extensions and we need not introduce any additional SILVER features. Because of this, and space limitations, we only describe the implementation of this extension. The new constructs (`set`, `expire`, and `delay`) are defined as one would expect. The use of the existing assignment operator for assignments to timers, however, is more interesting. Overloading on assignment is accomplished by defining an abstract `assign` production that is used by the concrete syntax in building the abstract syntax tree. This production has an `overloads` collection attribute of type `[Stmt]` that is initialized to the empty list and which extensions can write aspects for that add new trees into this list. The discrete-time extension defines an aspect on `assign` that checks if the expression on the right hand side of the assignment is of type `timer`. If it is, it adds a `Stmt` tree to this `overloads` list that is built using a new production for assignments to timers that is defined by the extension. The "dispatching" `assign` production can then inspect this list in `overloads` and if

it contains a tree, it takes it from the list and forwards to it. If this list is empty, the `assign` forwards to a tree constructed using the `defaultAssign` production shown above. (If the list contains more than one tree, then two extensions are trying to overload the same syntax and an internal error is raised.) This provides a "hook" in the host language that extensions can exploit to overload existing syntax. Here the timer-specific assignment is used to allow assignment between integer and timer types, which would otherwise not be allowed since the timer type is specified as being incompatible with numeric types in the host language. A similar mechanism is used to generate the `Timers` process and `init` clause which will run it. The root-level production for Promela models has a collection attribute to which extensions can add new global declarations that they want inserted at the end of the Promela program.

3.3 Solutions to some challenges in Promela

An interesting feature of Promela is that it allows C code to be embedded inside Promela specifications, as in, for example, `c_code` { *... C code ...* } phrases. In SPIN this is accomplished by recognizing the C code as a single token, not by parsing the embedded C code. SPIN reports syntax errors in the C code not when SPIN runs but when the generated verifier program, `pan.c`, is compiled. In ABLEP, however, we can parse the embedded C code directly and report a syntax error in the C code just as a syntax error in the Promela code is reported. Context-aware scanning makes this possible. We create a single grammar that includes the Promela concrete syntax and a separate specification of ANSI C concrete syntax. Even though many terminal symbols overlap in their regular expressions this does not cause a problem. For example, the composed grammar has two terminals for recognizing the keyword `int`. But because the parser is in a different LR-state when parsing the C code than it is in when parsing Promela, none of these overlapping terminal symbols occur in the same set of valid terminals that are passed to the scanner. Thus the scanner is never asked to return, for example, a Promela `int` keyword or a C `int` keyword - even though these two distinct terminals exist in the combined language specification. This fact is verified by COPPER when the scanner and parser are generated. Besides allowing syntax errors in the embedded C code to be detected and reported in a natural way, it also means that the language specification for Promela with embedded C code is declarative and easy to understand.

Forwarding is used to handle inlining as specified by the `inline` keyword in Promela. The declaration of an statement to be inlined is parsed and added to the environment `env` attribute. A inlining use then looks up this statement in the environment, instantiates the statement with the arguments provided at the call site and the forwards to this instantiated statement.

4 Related Work

The extensible language framework is based on the same principles that our colleagues and we have used to build ABLEJ, an extensible framework for Java

1.4 [15], a framework for a subset of Lustre [8] and a ongoing effort to do the same for C. There have been many investigations into the modular specification of languages in the attribute grammar community. Of particular interest is the JastAdd system [6] that adds a attribute grammar layer on top of Java. It has been used to build an extensible specification of Java 1.5 [5]. Other recent attribute grammar systems aimed at language extensibility have been specified as embedded DSLs: UUAG [13] in Haskell and Kiama [12] in Scala. MetaBorg [3], an language embedding framework based on term rewriting, has been used to develop JavaBorg - an extensible specification of Java.

There have been many efforts to extend Promela, with with new features or to provide tools that do additional analysis. In fact, the CPP macro processor is commonly used for these purposes. We discuss ETCH [4] and DTSPIN [1] as examples here because the cover much of the spectrum from light- to heavy-weight approaches and from ones that add new syntax to others that add only semantic analysis. Other examples can be found in the literature, primarily in past editions of the Spin Workshop, see [9].

5 Conclusion

One challenge with ABLEP is that SPIN, when simulating or verifying the ABLEP-generated Promela model, will report errors or issues on the pure Promela model written in the host language, not the extended language. We do not have a general solution for this problem, but SPIN itself suffers from this with its parser-based implementation of the `for` and `select` constructs. Still, sometimes a verbose but direct translation to SPIN works rather well, as is the case with the `select` statement that may generate many `if` options since, as described in Section 2, this may provide a more intuitive interaction with the user than the translation to a `do` loop. Our longer range goals are to investigate this problem in a general setting and we anticipate that ABLEP will be a good testbed for this effort. Another challenge arises if engineers introduce many different unfamiliar extensions as this may make it more difficult to read the specification. But the same thing happens if too many libraries are used in traditional programs. A bit of discretion on the engineer's part may be sufficient to address this concern.

Overall, we believe that extensible language frameworks such as ABLEP provide a good solution to the challenges of extending and evolving languages. ABLEP is middle-weight solution that enables more static analysis and syntactic expressiveness than is possible with simple macros, but at the cost of requiring an explicit pre-processing step to analyze the extended specification and generate the pure Promela version. It also does not provide the capabilities that one has in modifying the SPIN source code, but this is a significant effort that many will want avoid. An extensible framework such as ABLEPcan also take some of the pressure off of the host language developer. New language constructs and analyses can be tried out as language extensions, allowing for users to experiment with them and provide feedback based on their hands-on experiences. After this, some features may migrate into the host language if that is warranted. For

discrete-time constructs found in DTSPIN, this may be appropriate. In other cases, such as with the ETCH-like type analysis, the feature may be sufficient when packaged as a language extension.

References

1. Bosnacki, D., Dams, D.: Integrating real time into Spin: A prototype implementation. In: Proceedings of the FORTE/PSTV XVIII Conference. pp. 423–439. Kluwer (1998), reproduced in Bosnacki's thesis, available at `http://www.win.tue.nl/~dragan/Thesis/`
2. Boyland, J.T.: Remote attribute grammars. J. ACM 52(4), 627–687 (2005)
3. Bravenboer, M., Visser, E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In: Proc. of OOPSLA '04 Conf. pp. 365–383. ACM Press (2004)
4. Donaldson, A.F., Gay, S.J.: Type inference and strong static type checking for Promela. Science of Computer Programming 75, 1165–1191 (2010)
5. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: Proc. Conf. on Object oriented prog. sys. and applications (OOPSLA). pp. 1–18. ACM Press (2007)
6. Ekman, T., Hedin, G.: The JastAdd system - modular extensible compiler construction. Science of Computer Programming 69, 14–26 (December 2007)
7. Gagnon, E., Hendren, L.J.: SableCC, an object-oriented compiler framework. In: Proc. of 26th Technology of Object-Oriented Languages and Systems. pp. 140–154. IEEE Computer Society Press (1998)
8. Gao, J., Heimdahl, M., Van Wyk, E.: Flexible and extensible notations for modeling languages. In: Fundamental Approaches to Software Engineering, FASE 2007. LNCS, vol. 4422, pp. 102–116. Springer (March 2007)
9. Holzmanm, G.: Spin - formal verification. `http://www.spinroot.com`
10. Holzmann, G.J.: The SPIN Model Checker:Primer and Reference Manual. Addison-Wesley Professional (September 2003)
11. Schwerdfeger, A., Van Wyk, E.: Verifiable composition of deterministic grammars. In: Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM Press (June 2009)
12. Sloane, A.M.: Lightweight language processing in kiama. In: Proc. of the 3rd summer school on Generative and transformational techniques in software engineering III (GTTSE 09). pp. 408–425. Springer (2011)
13. Swierstra, S.D., Alcocer, P.R.A., Saraiva, J., Swierstra, D., Azero, P., Saraiva, J.: Designing and implementing combinator languages. In: 3rd Summer School on Adv. Functional Prog. LNCS, vol. 1608, pp. 150–206. Springer (1999)
14. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. Science of Computer Programming 75(1–2), 39–54 (January 2010)
15. Van Wyk, E., Krishnan, L., Schwerdfeger, A., Bodin, D.: Attribute grammar-based language extensions for Java. In: European Conf. on Object Oriented Prog. (ECOOP). LNCS, vol. 4609, pp. 575–599. Springer (2007)
16. Van Wyk, E., Schwerdfeger, A.: Context-aware scanning for parsing extensible languages. In: Intl. Conf. on Generative Programming and Component Engineering, (GPCE). ACM Press (October 2007)