Experiences Report on the Implementation of EPTs for GNAT

Rodrigo García García, Alfred Strohmeier

Software Engineering Laboratory
Swiss Federal Institute of Technology in Lausanne (EPFL)
CH-1015 Lausanne EPFL, Switzerland
{rodrigo.garcia, alfred.strohmeier}@epfl.ch

Abstract. Extensible Protected Types were devised to integrate concurrent and object-oriented features of Ada 95. This paper reports on a feasibility study based on implementing Extensible Protected Types for the GNAT compiler.

1 Introduction

Ada 95 supports both object-orientation and concurrent programming. However, the mechanisms corresponding to these two programming paradigms have been kept separate in the language, perhaps because the language designers were afraid of the problems that arise when mixing synchronization constraints with object-oriented programming. Indeed, combining them can lead to the so-called inheritance anomaly. Extensible Protected Types (EPTs), as proposed in [W00], extend the language by avoiding this phenomenon.

2 Structure of GNAT

GNAT is divided into two main components: the front-end and the run-time library. The front-end is the part responsible for translating Ada source code into the intermediate machine-independent representation that the back-end uses for generating machine-dependent object code. The run-time library offers to the front-end, among others, a procedural interface to the tasking features and implements them. Therefore, when the front-end finds a tasking related construct in the source code of a program, it replaces it by appropriate calls to the run-time library.

The front-end divides the compilation process into the following stages:

1. Lexical Analysis: This stage reads the characters from the source program and groups them into tokens.
2. Syntax Analysis (Parsing): It verifies that the tokens are ordered following Ada syntax rules and builds the abstract syntax tree (AST).
3. Semantic Analysis: It decorates the AST by adding semantic information to it.
4. Expansion: The representations in the AST of some complex Ada constructs are reduced to combinations of simpler constructs.
5. GiGi (GNAT-to-GNU translator): It translates the GNAT tree into the format understood by the code generator of GCC.

In general, the source code for each stage is composed of packages, one for each chapter of the Ada Reference Manual [ARM95]. In this way, each feature of the language can be easily found among the numerous source files that compose GNAT.

6 Possible Solution Strategies

6.1 Source Preprocessor

The first strategy that came to our mind was to preprocess the source files. The preprocessor would have to search for extensible protected types and related constructs in the application code and transform them into regular Ada source, e.g. by turning an extensible protected type into a regular protected type embedded in a tagged record type. This approach has the big advantage of being compiler independent.

Unfortunately it seems impossible to get the complete functionality of EPTs, as defined in [W00], by just combining the features of tagged and protected types. See e.g. [HB95] for a preliminary study of implementing EPTs with tagged types, and [BW95] for a discussion on how to combine tagged and protected types in order to build extensible objects with synchronization constraints.

The preprocessor-based approach has in addition the usual drawbacks of this technology: additional names have to be created, different from all programmer-defined names, error-reporting and debugging will refer to the modified source code, and not to the programmer-provided version, etc.

6.2 Modifying the GNAT Compiler

Since the source of the most popular Ada compiler, i.e. GNAT, is publicly available, we decided not to reinvent the wheel. Also, in order to keep the changes to a minimum, we tried not to add any new kind of node in the Abstract Syntax Tree (AST) of the compiler. We will see later how we tried to reduce EPTs to other constructs in the AST.

The document [MG99] contains an exhaustive description of the GNAT front-end, and it was indeed a valuable source of information. We used this document along with the comments in the source code of GNAT to understand how the compiler works, paying special attention to the parts dedicated to the implementation of protected and tagged types.

7 Syntax of EPTs

The set of production rules proposed in [W00] provide a complete syntax of EPTs. However, these rules are not stated in a way that relates them to the rules of the [ARM95]. In addition, for a feasibility study, or at least its first part, we didn't want to support the full EPT model. We therefore defined our own more limited syntax for extensible protected types.

For instance, our syntax does not take into account the extension of entries, or propose a notation for redispatching calls or deal with private extensions. The rules below are devised as a replacement of their counterparts in chapter 9 of the [ARM95]. It should be noted that type derivation is treated in chapter 3 and not in chapter 9 of the [ARM95]. The final placement of rules for EPTs would therefore have to be studied carefully.

protected_type_declaration ::=
    protected type defining_identifier [known_discriminant_part] **is** protected_type_definition;

protected_type_definition ::=
    [[**abstract**] **tagged**] protected_definition
    | derived_protected_definition

```
protected_definition ::=
        { protected_operation_declaration }
    [ private
        { protected_element_declaration } ]
    end [protected_identifier]

derived_protected_definition ::=
    [abstract] new parent_subtype_indication [protected_extension_part]

protected_extension_part ::=
    with protected_definition
```

8 Implementation of the Additional Syntax in GNAT

Our syntax for EPTs does not add any new reserved word into the Ada language. The lexical analyzer of GNAT was therefore left unmodified. The first stage of the compiler that had to be modified was the syntax analyzer, also known as the parser. We first had to add the new production rules for EPTs in the specification of the package *Sinfo*. This package holds the complete definition of the Ada language in the form of Ada comments. Each comment corresponds to a production rule of the language and describes its representation in the AST. Based on these comments, the compiler automatically generates a set of files containing the operations needed to manipulate the syntax tree. After including all the new rules, we modified the parser itself by adding the necessary instructions to correctly process protected tagged type declarations. These new instructions also recognize keywords such as **tagged**, **abstract** or **new** in the context of EPTs and call the standard GNAT error procedures in the case of syntax errors. For our implementation we took the code that parses regular tagged types as a model. On the overall, syntax analysis is quite mechanical, and we did not encounter any special problems in this part of our implementation.

9 Semantics and Expansion

The semantic rules of a language are more difficult to formalize than syntax rules. In [ARM95], semantics are stated in natural language, whereas syntax is expressed using a simple variant of BNF grammar. In GNAT, semantic analysis is mixed with expansion in a recursive fashion. Each node of the AST is analyzed and then expanded, resulting in new nodes that will be in turn analyzed, until we reach a node that cannot be further expanded, i.e. a so-called “leaf” node. Not astonishingly, this part of the compiler was more difficult to change.

10 Expansion of Protected Types

Protected types are expanded in GNAT by using a so-called corresponding record to store their private data fields and a synchronization component (figure 1). Besides containing a lock, the synchronization component acts as a link to the run-time library. Each subprogram, i.e. a procedure or function, of the protected type is then replaced by two related subprograms: one for internal calls and another one for external calls. Both have a formal parameter of the record type holding the private data fields. Using this parameter, the subprograms can manipulate the internal state of the protected object. The subprogram intended for internal calls does not require any synchronization and it contains the code of the original subprogram. The one intended for external calls starts by executing synchronization code on the synchronization component mentioned previously. Once it has acquired the right to continue, it calls the non-blocking internal subprogram to complete the operation.

Calls to entries are expanded in a different way: they are replaced by a direct call to the run-time library, passing all the necessary information in the arguments of the call.

```
protected type poT is
    procedure p;
private
    open : boolean := false;
end poT;

type poTV is limited record
    open : boolean := false;
    _object : aliased protection;
end record;
procedure poPT__pN (_object : in out poTV);
procedure poPT__pP (_object : in out poTV);
freeze poTV [
    procedure _init_proc (_init : in out poTV) is
    begin
        _init.open := false;
        _init_proc (_init._object);
        initialize_protection (_init._object'unchecked_access, unspecified_priority);
        return;
    end _init_proc;
]
po : poT;
_init_proc (poTV!(po));

procedure poPT__pN (_object : in out poTV) is
    poR : protection renames _object._object;
    openP : boolean renames _object.open;
    ...variable declarations...
begin
    ...B...
    return;
end poPT__pN;

procedure poPT__pP (_object : in out poTV) is
    procedure _clean is
    begin
        unlock (_object._object'unchecked_access);
        return;
    end _clean;
begin
    lock (_object._object'unchecked_access);
    B2b : begin
        poPT__pN (_object);
    at end
        _clean;
    end B2b;
    return;
end poPT__pP;
```

Figure 1. Expansion Schema for Protected Types in GNAT

11 Implementation of EPT

11.1 Use of a Tagged Record Type for Storing its State

As we have seen, for each protected type, the compiler generates a so-called corresponding record. This led us to the straightforward idea of making this record a tagged record in the expansion of EPTs. The following consequences result from this approach:

- Protected procedures and functions would become the dispatching operations of the tagged record, since they have it as a controlling parameter. They are not declared in a package specification though, and the compiler hence does not recognize them as primitive operations. It is however easy to modify the procedure in the compiler that takes this decision.
- Extension of an EPT would be achieved by derivation of a tagged record: The derived EPT would build its corresponding record by extending the corresponding record of the parent EPT. Therefore, no new extension mechanisms would have to be added.
- Like primitive subprograms of a tagged type, the expanded protected operations would be inherited by the derived record type. Operations could be overridden or new ones added.

Unfortunately, this simple idea and approach leads to some tricky problems. Up to now, when the compiler found a protected type, it created entities for its subprograms and components. Then it expanded them to normal subprograms and a record to hold the components, creating links from the original entities to these expanded entities. In the case of a derived EPT, we will have the opposite case: we inherit a record which already comes with the parent components and dispatching operations. Therefore, we have to create new entities in the derived EPT that correspond to these components and subprograms. These entities are necessary due to visibility problems: if they are not present in the protected type, their expanded versions will not be visible in the application source code, i.e. the code submitted for compilation. The implementation must therefore keep track of the existing relationship between “original” entities of the EPT and their expanded counterparts. It is rather difficult to maintain this correspondence, and our current implementation fails in some cases.

11.2 The Synchronization Component

Another serious limitation to this approach is due to the synchronization component that belongs to the corresponding record of the EPT. There are different synchronization components GNAT can choose depending on the characteristics of the protected type to expand. The selection depends mainly on the number of entries of the protected type: whether it has zero, one or more entries. As the synchronization component is inherited by all the descendants of the root EPT, we decided that we would need the most complex synchronization component, that is the one supports several entries. Unfortunately, this synchronization component does not allow a variable number of entries. The number of entries must be fixed at the creation of the synchronization component because the compiler allocates a queue for each entry for keeping the order of the calls. One simple solution would be to choose a synchronization component with a big number of entries to accommodate even a large hierarchy of EPTs, but this might be inefficient. Another possible solution could be to replace the inherited synchronization object by one with the appropriate number of entries. This would be ideal, but we do not know how to remove a component that was inherited in a tagged extension. Yet another solution could be to modify the run-time library, adding a new type of synchronization component which allows a variable number of entries.

12 Conclusions

Although the idea of implementing EPTs by means of a corresponding tagged record seemed to be promising, there were a lot of additional complications that did not allow us to complete the implementation. To sum up, the main problems are the following:

- Matching derived entities of an EPT with those created by the expander for the parent type is difficult.
- The number of entries in synchronization objects is fixed.
- Entries do not become dispatching operations of the EPT.

Since we were not able to solve these problems given our limited resources, we did not even try to start to work on redispatching or on inheritance of entry barriers. However, since entry barriers are implemented in GNAT as boolean functions, it should not be too difficult to use them for implementing inherited entries. Indeed, the barrier can only be modified by "and-ing" additional conditions [W00].

13 References

[W00] A. J. Wellings et al "Integrating Object-Oriented Programming and Protected Objects in Ada 95" ACM Transactions on Porgramming Languages and Systems, Vol. 22, No. 3, May 2000, pages 506-539.

[MG99] J. Martín, J. Miranda, F. Guerra and A. González "Estudio del frontal de GNAT. Incorporation of Drago" ("Study of the front-end of GNAT. Insertion of Drago") University of Las Palmas de Gran Canaria. March 1999. Only available in Spanish.

[ARM95] "Ada 95 Reference Manual. International Standard ISO/IEC 8652:1995(E)" Lecture Notes in Computer Science 1246. Springer, 1997.

[BW95] A. Burns and A. Wellings "Concurrency in Ada" chapter 13, pages 318-325. Cambridge University Press, 1995.

[HB95] P. de las Heras, F. J. Ballesteros, J. Centeno and J. M. González "Toward Protected Tagged Types in Ada 95" Carlos III University. November 1995.

[GB95] E. W. Giering and T. P. Baker " Implementing Ada Protected Objects - Interface Issues and Optimization" Proceedings of TRI-Ada'95, Anaheim, California, 1995.

[SB94] E. Schonberg and B. Banner "The GNAT project: A GNU-Ada 9X Compiler" Proceedings of TRI-Ada'94, Baltimore, Maryland, 1994.

[CP94] C. Comar and B. Porter "The GNAT Implementation of Tagged Types" Proceedings of TRI-Ada'94, Baltimore, Maryland, 1994.