The Application of Compile-Time Reflection to Software Fault Tolerance using Ada 95

P. Rogers[1] and A. J. Wellings[2]

[1]Ada Core Technologies
rogers@adacore.com
[2]University of York, York, UK
andy@cs.york.ac.uk

Abstract. Transparent system support for software fault tolerance reduces performance in general and precludes application-specific optimizations in particular. In contrast, explicit support – especially at the language level – allows application-specific tailoring. However, current techniques that extend languages to support software fault tolerance lead to interwoven code addressing functional and non-functional requirements. Reflection promises both significant separation of concerns and a malleability allowing the user to customize the language toward the optimum point in a language design space. To explore this potential we compare common software fault tolerance scenarios implemented in both standard and reflective Ada. Specifically, in addition to backward error recovery and recovery blocks, we explore the application of reflection to atomic actions and conversations. We then compare the implementations in terms of expressive power, portability, and performance.

Keywords. Reflection, software fault tolerance, Ada, backward error recovery, recovery blocks, atomic actions, conversations

1 Introduction

Lives and property increasingly depend on the correct operation of computer software. This dependence may be absolute because older technologies such as electromechanical, hydraulic, or pneumatic mechanisms are either inadequate or violate other system constraints, typically those of cost, weight, and power consumption. For example, the NASA Space Shuttle fleet is purely "fly-by-wire"; there is no (say) hydraulic backup control system. A larger number of people are potentially affected by fly-by-wire commercial aircraft. For example, the Boeing 737/300 and the Airbus A320, A330, and A340 aircraft are completely dependent upon reliable operation of the control software. Even more people will be affected by "drive-by-wire" automobiles as they become ubiquitous.

However, software for current and future applications – such as flight control systems – is both large and complex, such that full testing is not feasible. Furthermore, complete proofs of correctness are at best inherently limited by the potential for specification faults. Indeed, specification faults are considered the cause of the majority of safety mishaps [4]. The combination of potential specification

errors and overall complexity define the problem as one of handling unanticipated software faults. "Software fault tolerance" is the use of software mechanisms to deal with these unanticipated software faults [5, Preface].

Software fault tolerance is expensive and adds to the overall complexity of the system (which may even reduce reliability as a result). Nevertheless, software fault tolerance must be explicitly considered for safety-critical applications because software faults are unavoidable, as we discussed above, and because the techniques used for hardware fault tolerance generally do not handle software faults. Hardware fault tolerance is based on replication, on the grounds that the hardware may eventually "wear out" but does not contain permanent design flaws. Software faults, on the other hand, are widely held to be permanent design mistakes. Replication would simply create multiple copies of the same permanent mistake.

One of the software technologies considered for handling software faults is "reflection". Reflection is the ability of a computational system to observe its own execution and, as a result of that observance, perhaps make changes to that execution [6, 11]. Conceptually, software based on reflective facilities is structured into distinct levels: the baselevel and one or more metalevels. The baselevel addresses the functional requirements of the application. The metalevels represent and control the baselevel. As such, the metalevels are responsible for the non-functional aspects of the system. The differences in these levels can be illustrated in terms of a stage production: the baselevel is everything seen by the audience; the considerable activity off-stage occurs in the metalevels [3].

The metalevel and baselevel are causally related: modification of one affects the other. This relationship may be achieved by making the actual baselevel implementation available to the metalevels, such that changes by the metalevels are automatically effective at the baselevel. To the degree that the implementation is made available, everything in the implementation and application – the syntax, the semantics, and the run-time data structures – is "opened" to the programmer for modification via the metalevels.

Reflection offers a clean separation of concerns with great flexibility and it has, therefore, been a focus of research in software fault tolerance. However, most of these research efforts focus on handling hardware faults and those that address software fault tolerance use languages that are limited in one respect or another. Indeed, reflection has not been applied to a language with the features integrated within Ada. Furthermore, such a language has not been used to address software fault tolerance with reflective programming even though Ada is especially appropriate for systems in which reliability is critical.

We have implemented a reflective programming facility for Ada [9] and applied it to scenarios not otherwise explored by the fault tolerance community. Specifically, in addition to backward error recovery and recovery blocks, we explore the application of reflection to atomic actions and conversations. Having implemented the scenarios using both standard and reflective Ada, we then compare the implementations in terms of expressive power, portability, and performance. In section 2 we describe the concept of compile-time reflection as applied to our reflective Ada implementation "OpenAda". Section 3 compares the results of the scenario implementations using both standard Ada and OpenAda, and section 4 provides closing remarks.

2 OpenAda

The inefficiencies incurred from reflection, especially with interpretive implementations, are a significant problem. These inefficiencies and those imposed by reifying the entire processor have led to "open compilers" that allow users to change the language's semantics without necessarily incurring performance penalties. In these compilers the internals, including parsing, the data structures, semantic analysis, and so forth, are reified such that new functionality and even new syntax may be added to the language. These changes are achieved by subclassing these reified internal classes.

A distinct alternative is to have the translation driven by a specific, individual "translator" metaclass rather than by (subclasses of) the reified compiler internals. A metaclass is, in this alternative approach, a specialization of a predefined translator class [2]. The input source code contains an annotation that affects the translation by specifying the specific metaclass to be used to perform the translation. As a result, the metaclass can be said to customize the translation of the input source code rather than customizing the compiler's internals. The resulting metaobject protocol is referred to as a "compile-time MOP". With such a MOP the metalevel code only runs during compilation: the metalevel code controls translation of the program and, thereby, albeit indirectly, run-time behaviour. The metaclass can use this MOP to inquire about the primitive operations and components of a given type, for example, and may add, remove, or arbitrarily change them as required by the goals of the metaclass. As an illustration, these changes could involve invoking an acceptance test and, if it fails, restoring state and invoking a secondary variant routine. Translating metaclasses can be reused across applications whenever the corresponding translation is required.

OpenAda is our compile-time reflection facility for Ada 95 using the "translating metaclass" approach. A pragma named Metaclass in the baselevel source code specifies the translating metaclass to be applied to that baselevel code. The metaclass alters the baselevel source to implement the non-functional requirements of the system, thereby achieving the intended separation of the source code implementing the functional and non-functional requirements.

For example, a metaclass that translates baselevel code into code that incorporates recovery blocks could be declared as follows.

```
…
with OpenAda.Meta;
with OpenAda.Syntax.Constructs;  use OpenAda.Syntax.Constructs;
with Ada.Strings.Wide_Unbounded; use Ada.Strings.Wide_Unbounded;

package Inline_Recovery_Blocks is

  type Class is new OpenAda.Meta.Class with private;

  Failure : exception;

  procedure Finalize_Translator( This : in out Class );
```

```
  procedure Translate_Handled_Statements
    ( This    : in out Class;
      Input   : in out Handled_Statements;
      Control : in out OpenAda.Syntax.Visitation_Controls );

private

  type Class is new OpenAda.Meta.Class with
    record
      Required_Units          : Unbounded_Wide_String;
      Recovery_Point_Inserted : Boolean := False;
    end record;
   …
end Inline_Recovery_Blocks;
```

The metaclass Inline_Recovery_Blocks.Class translates any handled-sequence-of-statements it encounters in the baselevel code. Specifically, it converts any pragma Recovery_Block in the baselevel into a block-statement that implements a recovery block. (See Section 3.2 for more details of the pragma Recovery_Block and the expanded block statement it generates.) The variants and acceptance test for any inserted recovery block are specified as pragma Recovery_Block parameters. These subprograms need not be local to the translated unit so the translator inserts with-clauses for them as necessary. The names of these units are stored in the Required_Units component of the class. Similarly, the translator inserts a with-clause for a recovery mechanism named "Recovery_Point" but only does so once for the sake of understandability. Hence a Boolean flag is used to control that insertion and is also a component of the class.

The implementation of procedure Translate_Handled_Statements verifies that the statement immediately preceding pragma Recovery_Block is a call to the procedure named as the first variant specified to the pragma. It then removes that procedure call and inserts a tailored block statement that implements recovery block semantics for the specified variants. All variant procedure names are captured and have corresponding with-clauses inserted by another routine (Finalize_Translator) if necessary.

```
procedure Translate_Handled_Statements
  ( This    : in out Class;
    Input   : in out Handled_Statements;
    Control : in out OpenAda.Syntax.Visitation_Controls )
is
  Iterator      : Parsed_Content.List_Iterator;
  Next_Item     : Any_Node;
  Prev_Item     : Any_Node;
  The_Pragma    : Any_Pragmata;
  Pragma_Args   : Parsed_Actuals;
  New_Statement : Any_Node;
  Statements    : Parsed_Content.List renames
    Sequence_of_Statements(Input'Access).Content;

  use Parsed_Content, Syntax.Utilities;
```

```
begin
  Iterator := Make_List_Iterator( Statements );
  while More( Iterator ) loop
    Prev_Item := Next_Item;
    Next( Iterator, Next_Item );
    if Next_Item.all in Pragmata'Class then
      The_Pragma := Any_Pragmata( Next_Item );
      if Image(Pragma_Name(The_Pragma),Lowercase) =
         "recovery_block"
      then
        Pragma_Args := Parsed( Args(The_Pragma) );
        -- Any preceeding statement must be a procedure call,
        -- and must be a call to the first variant listed in the
        -- pragma Recovery_Block params.
        Verify_Primary_Call( Prev_Item, Pragma_Args );
        -- replace the pragma with the recovery block
        New_Statement := New_Recovery_Block( Pragma_Args );
        Replace( Old_Node => Next_Item,
                 New_Node => New_Statement,
                 Within   => Statements );
        -- delete the preceeding procedure call
        Delete_Call( Prev_Item, Within => Statements );
        if not This.Recovery_Point_Inserted then
          Append( This.Required_Units, "Recovery_Point " );
          This.Recovery_Point_Inserted := True;
        end if;
        Append_Package_Names(This.Required_Units, Pragma_Args);
      end if;
    end if;
  end loop;
end Translate_Handled_Statements;
```

3 Comparing Standard and Reflective Ada

Compared to other mainstream languages, Ada is unusual due to its integrated support for concurrency (especially asynchronous interactions), high-integrity systems, real-time systems, and object-oriented programming. Indeed, the language provides standardized support for the typical capabilities added to other languages via reflection, namely concurrency and distribution. These integrated facilities make expression of a wide assortment of reusable fault tolerance components – particularly those involving cooperating threads – easier than in other typical languages. We have implemented a number of scenarios using common software fault tolerance facilities to determine the potential advantages offered by reflection for such a language. These scenarios are written in both standard Ada 95 and OpenAda and involve reusable components implementing backward error recovery, recovery blocks, atomic actions, and conversations. We now compare these scenario implementations in terms of expressive power, separation of concerns, and performance.

3.1 Expressive Power

Lacking a widely accepted definition, we define expressive power as a matter of implementing requirements concisely. In this subsection we compare the expressive power of standard Ada against the combination of the baselevels and the metaclass translators. We do not examine the expressive power of the metaclass code itself, but, rather, the result of applying the metaclasses to the baselevels.

Using Pragmas to Annotate Baselevel Source. Several of the reflective scenarios use metaclass-specific pragmas to annotate the baselevel code for specific treatment by the translating metaclass. The pragma Recovery_Block, for example, is expanded into code implementing the corresponding functionality. Pragmas Atomic_Action and Conversation are similarly expanded into interactions with reusable components implementing the required services. Each occurrence of pragma Conversation individually enumerates the entire set of variants to be expanded into a conversation call. These variants need not be from the same package and need not be local to the unit containing the pragmas – the metaclass determines which variants require with-clauses and generates them accordingly. These pragmas thus represent a great deal of functionality with a very simple and succinct expression. One could argue that their power exists only in combination with the metaclass translators, but that is also true of standard Ada syntax translated by a standard Ada compiler.

These annotation pragmas also help hide arcane syntax and language rules. For example, the backward error recovery implementation requires a discriminant for each recoverable object declaration. These discriminants link the recoverable application object with a controlling "recovery manager" object. A discriminant is used for the sake of robustness because the compiler ensures the linkage is specified when the object is declared. However, discriminants add syntactic weight to the code and impose rules beyond those of normal record components. Moreover, these discriminants provide a permanent linkage to a single controlling cache object even though different associations might make sense at different points in the program execution.

The reflective version of the scenario carries none of this baggage because the baselevel code does not address recovery. The metaclass programmer is responsible for inserting the discriminant specifications, not the baselevel programmer. Indeed, the metaclass could translate the code to use a procedural registration facility instead, but neither approach appears in the baselevel.

However, pragmas are not as expressive as dedicated syntax. A pragma is no more expressive than a procedure call. The Ada tasking constructs are, in part, a reaction the limitations of the expressive power of procedure calls. Prior to Ada, an application programmer made calls into an operating system to achieve concurrency. These operating system calls provided neither visibility into thread interactions nor compile-time type checking. In contrast, dedicated syntax provides explicit interactions with compile-time checking and is much more expressive than a procedural interface (as found in Java, for example).

Limitations Due To Semantics That Cannot Be Reified. The expressive power of the reflective approach is limited by the fact that some semantics of the Ada language cannot be readily reified. A metaclass cannot alter the semantics of task activation or general object creation, for example, because there is no corresponding syntax to translate. In a reflective programming context these limitations do not compare well with other languages that use explicit syntax, such as constructors for object creation or explicit method calls for task activation, either of which are easily translated by a metaclass.

These limitations did not affect our scenario implementations because such translations were not necessary. However, had we been required to alter object creation or task activation and completion these restrictions would have been onerous. For example, one could imagine a requirement to register each task's activation for the sake of debugging.

Expressing Otherwise Inexpressible Requirements. Atomic actions and conversions are meant to be indivisible, such that no internal state changes are discernable outside the action until the action completes. "Information smuggling" occurs when these internal state changes are inadvertently leaked. Prevention is critical because system recovery cannot be reduced to atomic action recovery if action recovery is not complete. Smuggling cannot be fully prevented in Ada because of the visibility rules [10].

A significant feature of compile-time reflective programming is the ability to do complex semantic analysis during compilation. This analysis can be applied to enforce language restrictions to a project-defined subset, for example. In the case of atomic actions and conversations, we apply such analysis to enforce the semantic requirement against information smuggling. Wellings and Burns mention the use of pragma Pure as a means of precluding smuggling by rejecting with-clauses that name library units containing state [12]. Unfortunately, using pragma Pure is neither sufficient nor entirely desirable. It is not desirable because the "purity" is required of the library units referenced, not in the unit itself where the pragma would be placed. It is not sufficient because a devious programmer could import any arbitrary Ada unit via pragma Import – including one that leaks internal state to an impure unit – thereby circumventing pragma Pure.

In our reflective implementation the metaclass explicitly checks the library units in the transitive closure of the specified baselevel unit for object declarations. However, this approach is not sufficient for the same reason that the semantics of pragma Pure are not capable of detecting potential leaks: pragma Import can be used to create an undetected leak. Therefore, the metaclass also checks for pragma Import occurrences and rejects the package if any are found. Information smuggling is thus prevented.

3.2 Separation of Concerns

In the context of this comparison, "separation of concerns" is a matter of the separation between the code meeting the functional requirements and the code meeting the non-functional requirements.

Standard Ada has extensive support for separation of interface from implementation in the form of abstract data types, physically separate interfaces and implementations, and dynamic binding. But separating interface from implementation is not the goal. We wish to separate the code meeting the functional requirements from the code meeting the non-functional requirements.

Reflection promises a significant degree of this separation – potentially complete transparency – unless explicit interaction is intended, but in practice there are impediments to complete separation. We analyze these impediments in this subsection.

Baselevel Annotations. In some scenarios, the translating metaclass detects and expands metaclass-defined pragmas in the baselevel "inline" to implement the corresponding fault tolerance facility. For example, pragma Recovery_Block specifies the checkpoint manager object, the acceptance test function, and an unbounded list of procedures to be called as variants. The pragma is placed immediately after a procedure call that occurs in the baselevel to implement the functional requirements. For example:

```
…
Calculate_New_Position;
pragma Recovery_Block( Checkpoint,
                       Reasonable,
                       Calculate_New_Position,
                       Estimate_New_Position,
                       Reuse_Old_Position );
…
```

The metaclass removes both the pragma and the baselevel procedure call and then inserts code to implement a recovery block using the specified recovery object [8], acceptance test, and variants, with the original procedure called as the primary variant.

```
…
declare
  Variant_Failure : exception;
  Recovery_Block_Failure : exception;
  Num_Variants : constant := 3;
  use Recovery_Point;
begin
  Establish( Recovery_Data'Class(Checkpoint) );
  for Variant in 1 .. Num_Variants loop
    begin
      case Variant is
        when 1 =>
          Calculate_New_Position;
        when 2 =>
          Estimate_New_Position;
        when 3 =>
          Reuse_Old_Position;
      end case;
```

```
      if not Reasonable then
        raise Variant_Failure;
      else
        exit;
      end if;
    exception
      when others =>
        Restore(Recovery_Data'Class(Checkpoint));
        if Variant = Num_Variants then
          Discard( Recovery_Data'Class(Checkpoint) );
          raise Recovery_Block_Failure;
        end if;
    end;
  end loop;
  Discard( Recovery_Data'Class(Checkpoint) );
end;
…
```

As can be seen, the annotation pragmas in the baselevel indicate what operations are required and where they are required, but do not indicate how those operations are to be provided. This is the very essence of abstraction and separation of concerns. However, the pragmas do exist in the baselevel code, introducing a coupling between the baselevel and the metalevels. Essentially the pragmas are another form of explicit reference.

An alternative to the annotations is to have the metaclass automatically recognize the baselevel code to alter. In some cases the metaclasses do take this approach, for example with type and object declarations. This approach only makes sense, however, when either all such code is intended for transformation or the metaclass can distinguish between those occurrences that should be altered and those that should not. That distinction cannot be guaranteed in all cases.

Counter-intuitive Baselevel Type Declarations. In the standard Ada version of the backward error recovery scenario, a type used for simple counting was necessarily defined as an extension to a base type providing backward recovery. In the reflective version we wanted to hide recovery from the baselevel programmers for the sake of separation of concerns. To that end we removed the recovery code from the baselevel – the type is no longer derived from a recoverable base type – but the baselevel type must still be declared as a record type because the translated usage will be as a type extension rather than a numeric type. One must wonder whether the baselevel programmer would, in practice, declare a simple numeric counter as a record type.

The intuitive reflective approach would be to declare the type in the baselevel as a simple numeric type and alter it by the metaclass to become a tagged extension type. That approach is not viable. Although overloaded operators could be declared by the metaclass, numeric literals would no longer be available within clients and value assignment would require aggregates. A metaclass could conceivably make these translations within clients but the effort is difficult to justify.

3.3 Performance

Using compile-time reflection, the reflective implementations will ideally be at least as efficient as the standard Ada versions. The reflective versions may even be more efficient than those using standard Ada due to tailored translations taking advantage of application-specific knowledge. Our benchmark programs show this expectation is valid, although performance will vary with the specific translation strategies chosen. In other words, compile-time reflection need not impose performance penalties but a poor metaclass translation may very well generate source code exhibiting lower performance.

Backward Error Recovery and Atomic Actions. After translation, the sources for the reflective versions of both the backward error recovery and atomic actions scenarios are semantically equivalent to the standard Ada versions and, as a result, the average execution times are essentially identical. This result demonstrates that reflective techniques need not impose any performance penalty whatsoever, while nonetheless providing complete separation of concerns.

Conversations. The conversations benchmarks illustrate the fact that the reflective version may be slower than the non-reflective version; in this case, about five percent slower. However, the performance penalty is not inherent in the use of reflection – it is due to the translation scheme implemented by the metaclass programmer. In this case, the source code for the two versions is not semantically equivalent. The metaclass implements a translation that is relatively easy to produce but is not as fast as the non-reflective version. This difference was not intentional, although in hindsight the implemented translation scheme is clearly not optimal. Rather than revise the metaclass, however, we left it unchanged for the sake of illustrating the potential for deleterious effects.

Specifically, the standard Ada version makes better use of generic instantiations than does the reflective version. The standard version instantiates a generic conversation role procedure template at the outermost level of the enclosing package and shares this instance across the routines exported to the calling tasks. The shared instance is, consequently, instantiated and elaborated only once. In contrast, the reflective version (after translation by the metaclass) instantiates the generic procedure within the role procedures themselves and elaborates the instances on each invocation by the tasks. This approach is necessary because the metaclass can not "know" that a single shared instance is applicable.

Recovery Blocks. The recovery blocks benchmarks provide the best illustration of performance improvements due to source tailoring based on application-specific knowledge. The reflective version is approximately 30% faster than the standard Ada version.

The speed difference is primarily due to the fact that the reusable component applied by the standard Ada version protects itself against aborts, including both task abort and aborts due to asynchronous select statements. The reflective version is tailored to ignore aborts and, as a result, does not pay the price of the unnecessary

protection. (Certainly another client might need protection from abort, requiring a different translation.)

We created another standard Ada reusable recovery block component to verify that the performance difference is due to the abort protection. This component does not protect itself from aborts. The resulting performance profile is essentially identical to that of the reflective version (approximately one percent difference, i.e., within the margin of measurement uncertainty).

3.4 Comparison Conclusions

The expressive power of the reflective approach allowed the metalevel programmer to verify properties of the baselevel that cannot be expressed with standard Ada. As to separation of concerns, ample separation was achieved, although the reflective approach typically involved some degree of coupling between the baselevel and the metaclass. The code addressing non-functional requirements was, in general, both extensive and complex but did not appear in the baselevel. Finally, we saw that performance was at least that of standard Ada, given a reasonable metaclass translation approach.

Based on these comparisons we conclude that a compile-time implementation of reflective Ada does indeed provide enhanced expressive power and separation of concerns, with comparable or better performance, over that of standard Ada.

4 Concluding Remarks

We note that formal certification is a typical requirement for systems that might employ software fault tolerance techniques. Our implementations used the full Ada language, including tasks (which are inherent in atomic actions and conversations), exceptions, access types, access-to-subprogram types, dynamic dispatching, and other constructs that probably would not be allowed in certified application code. These constructs occurred both in the reusable components written in Ada 95 and the reflective and standard scenario implementations. Some of both the reflective and standard Ada scenario versions applied those reusable components as well. We have no insight into how to resolve the general conflict between certification and language subsets, other than to note that these subsets are expanding slowly over time (e.g., the ARINC-653 API [1] defines processes) and that a less reuse-oriented "inline" pragma expansion could probably have avoided prohibited features in some of the scenarios. We used the full language to avoid imposing a priori restrictions that could have unintentionally affected the later comparisons.

Finally, note that complete details about the compiler and the application to fault tolerance may be found in the Ph.D. thesis upon which this paper is based [7]. An electronic copy is available for download from the University of York as http://www.cs.york.ac.uk/ftpdir/reports/YCST-2003-10.pdf. The sources for the compiler, the fault tolerance components, and the benchmarks are available at http://www.classwide.com/OpenAda/.

References

[1] Airlines Electronic Engineering Committee, *Avionics Application Software Standard Interface, ARINC Specification 653-1*: Aeronautical Radio, Inc., 2003.

[2] S. Chiba, "A Metaobject Protocol for C++," Proc. Object-Oriented Programming Systems Languages and Applications (OOPSLA'95), Austin, Texas, 1995, pp. 285-299.

[3] G. Kiczales, J. des Rivières, and D. Bobrow, *The Art of the Metaobject Protocol*, Cambridge, Massachusetts: MIT Press, 1991.

[4] N. Leveson, "Software Safety: Why, What and How," *ACM Computing Surveys*, vol. 18, no. 2, pp. 125-163, 1986.

[5] M. Lyu, Ed. *Software Fault Tolerance*, in *Trends In Software*, vol. 3, Chichester: John Wiley & Sons, 1995.

[6] P. Maes, "Concepts and Experiments In Computational Reflection," *ACM SIGPLAN Notices*, vol. 22, no. 12, pp. 147-155, 1987.

[7] P. Rogers, "Software Fault Tolerance, Reflection, and the Ada Programming Language (YCST 2003/10)," in *Department of Computer Science*: University of York, 2003.

[8] P. Rogers and A. J. Wellings, "An Incremental Recovery Cache Supporting Software Fault Tolerance Mechanisms," *Journal of Computer Systems: Science and Engineering*, vol. 15, no. 1, pp. 33-48, 2000.

[9] P. Rogers and A. J. Wellings, "OpenAda: Compile-Time Reflection for Ada 95" in *Reliable Software Technologies -- Ada-Europe 2004*, vol. 3063, *Lecture Notes in Computer Science*, A. Llamosi and A. Strohmeier, Eds., Palma de Mallorca, Spain: Springer-Verlag, 2004, pp. 166--177.

[10] A. Romanovsky and L. Strigini, "Backward Error Recovery via Conversations In Ada," *Software Engineering Journal*, vol. 10, no. 8, pp. 219-232, 1995.

[11] B. C. Smith, "Reflection and Semantics in Lisp," Proc. 11th ACM Symposium on Principles of Programming Languages, 1984, pp. 23-35.

[12] A. J. Wellings and A. Burns, "Implementing Atomic Actions In Ada 95," *IEEE Transactions On Software Engineering*, vol. 23, no. 2, pp. 107-123, 1997.