INTRODUCTION TO THE ADAMS C++ SOLVER

G. Óttarsson

Mechanical Dynamics, Inc.
2300 Traverwood Drive
Ann Arbor, Michigan 48105-2195
gisli@adams.com

1 Introduction

The ADAMS Solver is a powerful numerical analysis program that automatically solves the equations governing kinematic, static, quasi-static, and dynamic system simulations. The ADAMS Solver is the solution kernel for all ADAMS products which facilitates building, testing, and refining virtual prototypes of mechanical systems. The current Solver has a structured modular design and is largely composed of FORTRAN subroutines. Over the past thirty years, additional functionality has raised the complexity of the program such that its data structures and top down design are impeding further development and maintenance. MDI's response to this problem was to begin development of the next generation ADAMS Solver founded on the object-oriented programming paradigm. In general, the object-oriented program decomposes a solution algorithm into subgroups of related parts, called classes, that take into account both the code and the data related to each group. In addition to controlling access to the data, classes have the enormous benefit of encapsulating complex implementation details behind a usable class interface. This enables developers of other related functionality to leverage the class while being spared the intellectual challenge of comprehending these details. The classes are then organized into a class hierarchy which further facilitates the addition of new objects and functionality.

The first part of this paper will formally introduce the next generation ADAMS Solver, explain the motivation for its development, and describe the benefits of its use. The second part of the paper, will describe advancements in solution technology and other features unique to the new Solver. Subsequently, compatibility issues between the new and legacy solvers, as well as what the user can expect from the new Solver, will be discussed. Lastly, quality assurance and deployment plans for the new Solver will be presented.

2 The Next Generation ADAMS Solver

In the late 1990s Mechanical Dynamics, Inc. (MDI) committed to developing the next generation ADAMS Solver, otherwise known as the ADAMS C++ Solver. The C++ Solver is a compact and complete object-oriented class library for the computer simulation and analysis of dynamics systems. It is designed to be the solution kernel for all ADAMS products. Within the next development cycle, the C++ Solver will support all the current modeling elements, statements, and commands, while offering novel functionality not available in the current Solver. Through its well defined Application Programming Interface (API), the C++ Solver is accessible as a stand alone ADAMS Solver, as an integrated solver in ADAMS/View, or as an embedded solver in a larger client applications. Note that this latter configuration allows users to link their own programs against the library and make calls to the API to build, simulate, and analyze a dynamic system.

Some readers will be curious about the choice of C++ over other OOP languages. Other languages were considered and rejected. Among these were FORTRAN 90, SmallTalk and Java. The C++ programming language was the language which combined best support for object-oriented programming, performance, and compatibility with C and FORTRAN. Familiarity among developers and the benefit of international standardization also played a role.

With its object-oriented programming paradigm and streamline class hierarchy, the C++ Solver promises to reduce both development time and maintenance costs. Furthermore, the API enables larger CAD/CAE applications to embed ADAMS Solver technology, making system level simulation more widely available. Clearly, users will benefit from the initiative. Shorter development cycles and improved maintenance, puts higher quality, feature rich applications in the hands of analyst sooner. Increased availability of ADAMS products through embedding, promises that users can remain in their favorite CAE environment, while accessing state-of-art system simulation tools.

3 Enhancements in Solver Technology

The C++ Solver features numerous enhancements compared to the FORTRAN Solver directly or indirectly related to the choice of programming language and its object-oriented architecture.

3.1 Software modularity

Although the choice of programming languages has no practical importance to the end users of software, it is difficult to justify MDI's decision to rewrite the ADAMS Solver in a modern programming language without outlining some of the direct benefits of this decision. These benefits are extremely well illustrated by a case study of the comparative effort of adding support for flexible bodies to the FORTRAN Solver on one hand, and C++ Solver on the other.

One of the principal advantages of the Object Oriented Programming paradigm is the ability to encapsulate, or hide, implementation details of one software module from another. This desirable quality is not one of the strengths of the FORTRAN Solver because the FORTRAN language provides almost no tools to facilitate an encapsulating design. Consequently, the developers of the FLEX_BODY modeling element were required to comprehend and modify nearly every area of the software — an error prone and intellectually challenging project. The magnitude of the task was such that it has yet to be fully completed. For instance, some force elements can only be connected to a FLEX_BODY via an intermediate *dummy* PART.

One of the chief goals of the C++ Solver project was to achieve much higher levels of encapsulation. The developers of the FLEX_BODY element (or other future body implementations) should not have to be concerned with the technical details of forces and joints connecting two bodies, the various numerical methods, or the numerous other facilities of the ADAMS software.

The ADAMS modeling paradigm presents natural encapsulation barriers. An ADAMS model consists of **bodies** connected to each other by forces and joints (which we will collectively refer to as **connectors**). Connectors attach to bodies at **marker** locations. The characteristics of a force (the force law) may be described using function expressions and user subroutines (collectively referred to as **expressions**) that contain **measures** measuring marker kinematics such as position, orientation, velocity, *etc.*

The encapsulation goal involves implementing each class of objects in such a way that each object may view the other in a generic fashion. In other words, a connector should not be concerned whether the markers that it attaches to are on a rigid body or a flexible body. Similarly, a force should not be concerned about the contents of an expression governing its force law, nor should the kinematic measures contained in the expression know what kinds of markers are being measured.

It is a testament to the success of this implementation that a flexible body and a flexible MARKER were added to the C++ Solver in a fraction of the time it took to add the FLEX_BODY to the FORTRAN Solver. Neither the forces and joints nor the expressions required modification because the FLEX_BODY was automatically supported by them. Similarly, future joint or force development will not need to make special provisions for flexible bodies.

3.2 Bodies and markers

In the C++ Solver, a marker operates as a kinematic interface to a body. An implementer of a new body type must, in addition to formulating the equations of motion of this new body type, implement a new type of marker belonging to this body.

All other facilities in ADAMS interact with markers in generic terms. A kinematic marker measure in a function expression (*e.g.* `DX()`, `VR()` or PSI()) , a force applied to a marker or a joint constraining the motion of a marker need not know the specifics of this marker, e.g., what kind of body the marker belongs to.

Once the marker has been implemented, the new body is fully supported by the rest of ADAMS, which significantly limits the scope and complexity of a project to implement a new body type.

This also justifies greater creativity in the definition of new types of markers, as seen by the following examples.

The C++ implementation of FLEX_BODY markers will be enhanced in a number of significant ways. In the FORTRAN Solver, markers on a flexible body must coincide with a particular node and will deform along with this node. In the C++ Solver, markers on a FLEX_BODY can be attached to zero or more nodes coinciding with none of them and move among them, as is the case with a FLOATING marker. The marker will move as a weighted average of the motion of the nodes that it is attached to, and a force applied to the marker will be distributed among its nodes.

The C++ Solver will also feature a CURVE MARKER, a marker which slides along a curve maintaining its X axis along the curve and its Y axis along the normal to the curve. This will allow the replacement of the PTCV (point-curve) and CVCV (curve-curve) constraints because these can be implemented using the fundamental joints, e.g. SPHERICAL and REVOLUTE joints. It will also allow users greater creativity in creating higher pair constraints. A Surface marker is also expected to appear in a future release of C++ Solver.

3.3 Measures and Expressions

One of the greatest advances in solver technology came in the area of expressions. Expressions in the FORTRAN Solver exist in two forms: text expressions and user written FORTRAN subroutines. These may be used to define general modeling elements, e.g.,

```
DIFF/1, FUN=DIF1(1)-VX(2,3)
```

or

```
SFORCE/1, FUN=USER(1,2,3)
```

3.3.1 Lifting limitations

Expressions in the FORTRAN Solver suffer from several limitations.

- They only exist in a scalar form. Consequently, even when defining a vector force, the individual components of the force must be specified and it is incumbent on the user to correctly expand vector inner products and cross products.
- Text based function expressions and FORTRAN subroutine based expressions cannot be mixed. Constructs such as

  ```
  SFORCE/1, FUN=2.*USER(1,2,3)
  ```

  are not possible.
- In order to compute derivatives of function expressions, as are needed to evaluate the system Jacobian for numerical methods in ADAMS, the FORTRAN Solver resorts to finite differencing. Although this is adequate for the Newton-Raphson Jacobian, it is insufficient for situations where derivatives are required directly in the Equations of Motion, such as when projecting constraint Lagrange multipliers on force balance equations. This prevents the use of expressions to define constraints and is the reason that MOTION generator functions must only be functions of time, not system state.

Expressions in the C++ Solver lift all of these limitations. The C++ Solver has both 1D and 3D expression. In addition to the expression's value, the expression classes can compute the first and second time derivative, and partial derivatives relative to system state and time, without resorting to finite differencing. In other words, the C++ Solver computes analytical derivatives of arbitrary user defined expressions. Analytical expressions can often be computed more efficiently and are always more accurate. This accuracy can translate into faster convergence and allows them to be used in user defined constraints.

Expressions based on user subroutines deserve a separate mention. The legacy user subroutine interface is fully supported by the C++ Solver and may be written in the FORTRAN or C programming languages, conforming to the natural calling conventions of these languages. A user subroutine may of course also be written in C++, but naturally, the C++ name mangling must be disabled by defining the function as `extern "C"`.

3.3.2 Analytical derivatives

It is worthwhile to describe briefly how the C++ Solver achieved its analytical derivative capability When the FORTRAN Solver requires partial derivatives of user expressions, it does so through finite differentiation, by perturbing each state of the body. The C++ Solver uses a completely different approach. The C++ Solver recognizes that a users expression never depends directly on state. Rather it depends on measures of state, such as `DX()`, `VARVAL()`, *e.g.*

$$f = f(M_1(q), M_2(q), \ldots)$$

where M_1 and M_2 are measure quantities. The C++ Solver utilizes the chain rule to differentiate such expressions:

$$\frac{\partial f}{\partial q} = \frac{\partial f}{\partial M_1}\frac{\partial M_1}{\partial q}\frac{\partial f}{\partial M_2}\frac{\partial M_2}{\partial q} + \ldots$$

The measure implementation in the C++ Solver provides the analytical partial derivatives of the measures relative to the state variables, $\frac{\partial M_1}{\partial q}$. Meanwhile, the partial derivatives of the function relative to the measure $\frac{\partial f}{\partial M_1}$, are achieved by developing differentiation capabilities into the various operators and intrinsic functions of the expression language.

User subroutines are the only area in the expression system where the C++ Solver must resort to finite differencing to compute partial derivatives. Since a user subroutine is a *black box* for the Solver it is not capable of applying its analytical derivative algorithms. In user subroutines, measures are accessed through calls to the SYSFNC and SYSARY utility subroutines and the partial derivatives of the user subroutine are obtained via finite differencing, by perturbing the values of the measures. Furthermore, a new utility subroutine `SYSPAR`, companion to the `SYSFNC` and `SYSARY` utility subroutines has been added, allowing the user to contribute partial derivatives, on a voluntary basis, thereby suppressing finite differencing. Refer to the Subroutine manual for ADAMS 12.0 for details on SYSPAR.

3.3.3 User defined built-in functions

In addition to user subroutines, users will be able to customize the expression language by extending the library of built-in functions. However, because of the expression language's analytical derivative capabilities, this carries with it the burden of computing, not only the value of the function, but also the derivatives of the function. A trivial, contrived example illustrates how a user would completely implement a cosine function, were it not already provided:

```
class MY_COS : public USER_1ARG
{
   static double func( const double* args, int iord1 )
   {
      switch(ord){
      case 0:  return  std::cos(args[0]);
      case 1:  return -std::sin(args[0]);
      case 2:  return -std::cos(args[0]);
      case 3:  return  std::sin(args[0]);
      default: return  func(args,iord-4);
      }
   }
public:
   MY_COS(Expression1D x) : USER_1ARG("COS", func, 1, &x);
};
```

which could now be used as any other built-in function:

```
using namespace AdamsSolver;

Expression1D exp = MY_COS(AX(mar1,mar2));
double now = exp.val();
sforce1->expression(exp);
```

The example opens the namespace for the ADAMS Solver and instantiates a 1D expression object called exp. The example shows how the expression can provide its value and/or be used in the definition of modeling elements, in this case an SFORCE object called sforce1.

3.4 Connectors

Although the applied forces and kinematic constraints are fundamentally different in nature, they share the common attribute of connecting bodies by applying forces to them at marker locations.

Little needs to be said about applied forces. User customizable forces are already one of the greatest asset of the FORTRAN Solver and few changes were needed. Since user defined forces are based on expressions, they have of course benefited from enhancements to the expression system, particularly the the ability to define 3D expressions directly. For instance, the statement

```
  VTORQUE/1, I=1, JFLOAT=2
  , TX = DY(3,1)*FZ(3)-DZ(3,1)*FY(3)\
  , TY = DZ(3,1)*FX(3)-DX(3,1)*FZ(3)\
  , TZ = DX(3,1)*FY(3)-DY(3,1)*FX(3)
```

will be writable as

```
  VTORQUE/1, I=1, JFLOAT=2, FUN = DXYZ(3,1) % FXYZ(3)
```

Where the percent symbol will denote a cross product.

Kinematic constraints have seen *much* greater improvements. Constraints in the C++ Solver will, much like the forces, be customizable by the user. The FORTRAN Solver features some user defined constraints, e.g. the UCON and the MOTION, but the use of the UCON is hampered by its complexity while the customization of the MOTION constraint is limited to functions of time. A user defined constraint, will not suffer from such limitations. As an example, please consider the following user defined constraint.

```
CONSTRAINT/1
, FUN = VARVAL(1) - VARVAL(7)
, FUN = UVZ(99) * ( W(11,10) % D(99,11) - W(21,20) % D(99,21) )
```

This constraint manages two scalar expressions and has the purpose of constraining these expressions to zero. The first expression enforces the equality of two variables. The second expression is a non-holonomic gear constraint containing the dot product of a unit-vector in the Z direction of a common velocity marker, 99 and the difference between two cross products. Note that `W` and `D` are angular velocity and position *vectors*, respectively.

The developers of the C++ Solver took the idea of user defined constraints further. To ensure that user defined constraints work efficiently and correctly the built in constraints, JOINT, JPRIM, GEAR, COUPLER, MOTION are all implemented internally based on expressions. For example, the internal representation of the SPHERICAL joint between markers 1 and 2 is equivalent to:

```
CONSTRAINT/1
, FUN=DX(1,2)\
, FUN=DY(1,2)\
, FUN=DZ(1,2)
```

Those familiar with the formulation of the equations of motion of constrained multibody dynamics will appreciate the complexity of the capabilities required in the expression system to support this feature.

3.5 The Application Programming Interface, API

To guarantee modularity of the ADAMS product line and the ability to embed ADAMS C++ Solver in third party products, the C++ Solver is provided as a C++ class library. Using this class library, a software developer may create a dynamic analysis program, either by integrating it with another program or by writing a dedicated driver program. The ADAMS/View program is an example of the former, while the command line based C++ Solver is an example of the latter.

The C++ class library exposes the ADAMS Solver to the full power of the C++ programming language. Using this class library, it is possible to:

- Instantiate modeling elements and set and modify their attributes, similar to the ADAMS statement and command languages.

  ```
  using namespace AdamsSolver;
  Model mod1;

  Ground ground(&mod1);
  Marker ground_mar1(&ground);
  ground_mar1.qp(Vec3(1,2,3));

  Part par1(&mod1);
  par1.mass(10.);
  par1.qg(Vec3(1,-1,1));

  Marker mar1(&par1);
  mar1.qp(Vec3(1,2,1));
  ```

- Instantiate and evaluate solver run-time expression objects

```
Expression1D exp=Sin(AZ(&mar1, &ground_mar1));
double a=exp.val();

Storque sto1(&mod1);
sto1.i(&mar1);
sto1.j(&ground_mar1);
sto1.expression(exp);
```

- Create user defined classes that are composites of ADAMS modeling elements. It is also possible to create measure classes for those modeling elements. An contrived example in Appendix A illustrates these concepts. Note that although the user defined element in the example is only a composite of ADAMS objects, it could, similarly, be a composite of ADAMS objects and objects from the 3rd party produce, *e.g.* graphical primitives or material primitives.

The C++ Solver API is a collection of insulating classes which present to its user the public interface of selected ADAMS C++ Solver classes, without exposing proprietary implementation details. Because of its design, the API provides a compilation firewall, such that an application that is dynamically linked against one version of the C++ Solver library does not have to be recompiled even though a new ADAMS version is released.

3.6 ADAMS/Flex

It was already mentioned, in Section 3.1 how strong encapsulation allowed a rapid implementation of A/Flex and that all appropriate forces and joints were automatically supported.

Section 3.2 discussed some of the advances to flexible body markers. By allowing markers to be offset from nodes, attaching them to multiple nodes (or no nodes at all) and allowing them to be *floating* it is believed that superior modeling practices will be possible.

The FLEX_BODY derives disproportionately high benefit from the removal or modification of the finite differencing scheme described in Section 3.3.2. This earlier section described how the FORTRAN Solver must perturb each state in order to compute partial derivatives w.r.t. this state. When a flexible body has a large number of modal states, this can be extremely time consuming. Even when the C++ Solver must resort to finite differentiation, it perturbs measures rather than states, recognizing that the number of measures that a function depends on is often smaller than the number of states. This is particularly true in the case of flexible bodies.

The last noteworthy modification to ADAMS/Flex is the introduction of a new analysis coordinate system, coincident with the center of mass of the undeformed flexible body. This is believed to have benefits for numerical robustness in models with flexible bodies. Also, since the C++ Solver restricts modes to be modes of an unconstrained body, a change which trades minor modeling flexibility for significant computational efficiency, this opens the possibility of creating a center-of-mass marker for the flexible body and making it accessible to the user.

4 Compatibility

As these words are written, the C++ Solver falls short of supporting all the functionality of the FORTRAN Solver. However, the functionality gap is rapidly closing, and following the deployment schedule outlined in Section 7, the C++ Solver will offer the same functionality as the FORTRAN Solver and than some. When the C++ Solver is complete, users can expect legacy models and user subroutines to be completely compatible with the new solver. Both a C/C++ and FORTRAN user subroutine interface will be supported, so users will not have to convert FORTRAN user subroutines to C/C++. The new Solver will also support standard Solver input and output. A model can be defined using the standard ADAMS Solver Dataset language, and then modified or simulated using the ADAMS Solver command language. For users who prefer to work in the ADAMS/View environment, simulations using the C++ Solver can be invoked from here as well.

With the exception of the ".out" file, the C++ Solver will generate the same output files as the FORTRAN Solver. Result files will contain the kinematic and kinetic history of your model. Request files will contain the time history of the user's special data request. The format of these two machine readable files is invariant to both solvers. Message files will contain information concerning the progress of your simulation, but the human readable message file will receive some format changes and message content will be altered. Furthermore, the C++ Solver will support the FEMDATA statement to export component load, deformation, stress, and strain fields for input to subsequent finite element of fatigue life analysis.

When completed the C++ Solver will perform all analysis types, including linear analysis, and be fully integrated with all ADAMS horizontal and vertical products.

The Section 5.4, below, discusses a few exceptions to the general compatibility rules described in this section.

5 User Experience

Users contemplating a switch to the C++ Solver will be curious to know what benefits and drawbacks to expect. This section describes what the users of the new C++ Solver are likely to experience.

5.1 Speed

The conventional wisdom among the developers of high performance numerical analysis software is that the FORTRAN programming language is inherently more suitable to such tasks than languages such as C++. But MDI's initial exploration of the C++ was not motivated by speed considerations. Rather, the modern, object-oriented features of the C++ language were seen as a way of managing the ever increasing complexity of the ADAMS Solver and to promote innovation.

Fortunately, initial concerns about loss in performance have proven unfounded. Although the C++ Solver is occasionally slower, it is never significantly slower and often faster than the FORTRAN Solver. This in spite of the fact that serious efforts on performance tuning have not yet been a priority.

We do not expect users migrating to the C++ Solver to be disappointed by its lack of speed.

5.2 Accuracy

It may seem like a contradiction to say that the C++ Solver and the FORTRAN Solver are equally accurate solvers, yet are not guaranteed to yield the same solution. To resolve this apparent contradiction it is helpful to recall that the ADAMS solution is only accurate to a user controlled error tolerance and that two "correct" solutions can fit within this error bound. Furthermore, in the ideal situation, if the error tolerance is tightened, the two solvers will converge to the same exact solution. Unfortunately, reality sometimes falls short of this ideal. As veteran users of ADAMS know, it is not possible to tighten the error tolerance indefinitely because due to finite computer precision convergence failures will often set in before the desired convergence is achieved. An attempt to push the C++ and FORTRAN Solvers to the same exact solution will often be foiled by this shortcoming.

The developers of the C++ Solver were presented with the challenge of reaching an *identical* solution as the FORTRAN Solver, while attempting to improve the ADAMS formulation in every way possible. During the development of the C++ Solver, many enhancements were made to the way the

FORTRAN Solver formulates the mathematical description of a dynamic model. None of these deviations from the formulation in the FORTRAN Solver was taken lightly and several modification were abandoned when their benefits were marginal.

The numerical time integration of nonlinear dynamical systems can be an extremely challenging proposition. Many ADAMS models push the envelope of the capabilities of the ADAMS integrators and even before the advent of the C++ Solver many ADAMS users would experience the need to adjust integration or modeling parameters after upgrading from one version of the FORTRAN Solver to another or, in extreme cases, between two different hardware architectures running the same version of FORTRAN Solver.

It would therefore be naive to expect all customer models to migrate without incident from the FORTRAN Solver to the C++ Solver given the number of enhancements that were made to the C++ Solver.

It is fair to ask the question whether the C++ Solver could have offered a FORTRAN Solver compatibility mode, selectable by a user interested in migrating a model from the C++ Solver to the FORTRAN Solver. This idea was briefly considered, but abandoned by the C++ development team because of the following arguments.

- The team would *never* have been able to mimic completely *all* the behavior of the FORTRAN Solver.
- Some of the differences between the C++ and FORTRAN Solver are extremely fundamental. FORTRAN behavior could, in these cases, only have been mimicked at very high development cost.
- A compatibility mode would have greatly increased the cost of quality assurance, because both modes would have to be rigorously tested.
- Inevitably users with interest in compatibility would prefer different levels of compatibility, leading to multiple compatibility flags and an even greater quality assurance cost in the face of multiple combinations of these compatibility flags.

Compatibility modes were, in the end, not considered tenable.

5.3 Hidden Benefits

Users will benefit from the C++ Solver development will in ways not measured by new capabilities and performance. As we described earlier, the complexity and fragility of the FORTRAN Solver had reached a point where a stagnation of innovation was a definite risk.

The C++ Solver, through its modern architecture, is a considerable improvement in this regard, and users should expect a more rapid development of features in the future.

Users will also benefit as developers now have to opportunity to experiment with changing fundamental assumptions in the ADAMS software in the search for improved performance. For instance, it has long been speculated that Euler Parameters are a better choice of orientation states than Euler Angles, a hypotheses that ADAMS developers were powerless to study in the past. The C++ Solver makes such an experiment possible.

5.4 Potential Pitfalls

Users managing existing ADAMS models will chiefly be concerned whether the new C++ Solver will have the full capabilities of the FORTRAN Solver and, whether it will generate identical or comparable results.

5.4.1 Obsolete capabilities

Although one of the goals of the C++ Solver is to be a drop-in replacement for the FORTRAN Solver, there are certain capabilities of the FORTRAN Solver that MDI considers obsolete. To illustrate the various reasons for dropping capabilities, this section would discuss a selection of deprecated capabilities:

Dating back to the time before the ADAMS/View graphical user interface, is an often overlooked **graphical display** built into the FORTRAN Solver. Although the graphical display is often useful to those familiar with it, the cost of replicating it in the C++ Solver could not be justified.

Some capabilities of the FORTRAN Solver are associated with discontinued products. The NFORCE element, created in conjunction with the ADAMS/FEA product is such an element. Although this element may have found uses outside the ADAMS/FEA application, these uses are believed to be served by the far superior ADAMS/Flex capability.

Other capabilities of the FORTRAN Solver have found little favor among ADAMS users due to their complexity and lack of usability. The UCON element is an example of this such a capability, which will be replaced with a more simple modeling element, the CONSTRAINT, in the C++ Solver.

5.4.2 Miscellaneous legacy behavior

In the past, users have discovered, or accidentally taken advantage of undocumented behavior of the FORTRAN Solver. Consider, for instance, that the FORTRAN Solver makes calls to the VARSUB and VFOSUB user subroutines in a particular order. Some users have taken advantage of this, e.g., by caching values computed in one subroutine for use, later, in the subroutine. Unfortunately, this behavior has not been replicated in the C++ Solver, where modeling elements are evaluated in no particular order.

In other cases, users with complex user subroutines have taken advantage of undocumented function interfaces to the FORTRAN Solver. Usually, these users have learned about these interfaces through communications with MDI personnel and feel justifiably entitled to continue to call these functions. However, records have rarely been kept at MDI, indicating that a user has been provided information about an undocumented interface. It is, consequently, impossible to guarantee successful deployment of all existing user subroutines.

6 Quality Assurance

MDI takes the testing of the C++ Solver extremely seriously. Fortunately, the developers of the C++ Solver benefit from an arsenal of tests, developed through the history of the FORTRAN Solver, with carefully validated results. Throughout the development of the C++ Solver its developers have ensured that the C++ Solver also returns valid results from these tests. Coverage analysis will be performed to ensure that the FORTRAN Solver test suite adequately exercises the C++ Solver.

The second round of testing will involve the test suites managed by MDI's Vehicle Products Group. These tests contain complete models that are more representative of *real world* models than the in the FORTRAN Solver's test suites that many are targeted at a single element or capability (unit tests).

Additionally, the C++ Solver will take advantage of its API, which will facilitate unit testing at a much finer granularity than is currently possible with the FORTRAN Solver. For instance it can be hard to target exceptional numerical problems by building a model in an ADAMS dataset, the only input method for the FORTRAN Solver, while presenting this numerical challenge through the API may be quite viable.

The C++ Solver will also have its own performance analysis test suite to gauge its performance against the FORTRAN Solver, to prevent performance regressions and to measure performance gains as developers shift their focus to improving the performance of the software.

In parallel with these testing activities users are encouraged to run their models with the C++ Solver and report their experience to MDI. The developers of the C++ Solver have strived to anticipate the myriad ways in which creative users have utilized the ADAMS software in their models, so that the transition for the FORTRAN Solver to the C++ Solver may go smoothly. However, the best way for users to prevent future problems running their models with the C++ Solver is to participate in the testing of this software, as soon as it supports the set of capabilities used in their models.

7 Deployment

A C++ Solver deployment plan is emerging and will be presented to the user community to communicate the transition process. The plan calls for a roll out of the C++ Solver in stages to different users, while scaling back FORTRAN Solver development activity until demand for it vanishes and it is eventually retired.

The C++ Solver has, for some time, been the Solver of choice for the ADAMS Software Development Kit (SDK), which is used in embedded CAD applications. These applications have, historically, had access to a limited feature set, making the C++ Solver an effective choice.

In recent versions of ADAMS, the C++ Solver has been available to the general ADAMS user, either as an alternative command line solver, or as an alternative built-in solver inside ADAMS/View. In an attempt to make it easy for the A/View user to experiment with the two solvers, switching solvers was made extremely easy. Valuable feedback has resulted from this configuration.

The goal with the next release of ADAMS, the 12.0 release, is to make the C++ Solver a credible alternative for the power users in the MDI Vehicle Product Group, thereby making it viable to test the C++ Solver with the Chassis products during the 13.0 development cycle. Although the C++ Solver will not be one hundred percent feature complete, relative to the FORTRAN Solver, before the 12.0 release, nearly all fundamental modeling and analysis capabilities will be available.

The goal of the 13.0 release is to make the C++ Solver the default solver for Chassis products. The FORTRAN Solver will remain an easily accessed alternative to ensure user comfort and to operate as a safety net in eventuality that the C++ Solver falls short of its goals. It is presumed that although the C++ Solver will, by the 13.0 release, offer capabilities not shared with the FORTRAN Solver, the Chassis products will use these features sparingly to ensure model compatibility.

Development activity on the FORTRAN Solver will soon be reduced to support and essential maintenance, recognizing that time is much better spent developing capabilities in the more extensible C++ Solver. MDI recognizes that with the slowing down of FORTRAN Solver development activity it is paramount that ADAMS users quickly migrate to the C++ Solver. It is hoped that the need for the FORTRAN Solver will gradually vanish, but if it does not, it is possible that the FORTRAN Solver will be available for a long time.

A A sample API model

```
#include <vector>
#include "adams_api.h"

using namespace AdamsSolver;

class Link
{
   Part   *link;
   Marker *left, *right;
   Joint  *left_pin;
public:
   Link(Adams* parent)
   {
      link = new Part(parent);
      left = new Marker(link);
      right= new Marker(link);
      left_pin=new Joint(parent);
      left_pin->jtype(Joint::REVOLUTE);
      left_pin->j(left);
   }
   ~Link()
   {
      delete link;
      delete left
      delete right;
      delete left_pin;
   }

   void mass(double ms){link->mass(ms);}
   double mass(double ms) const {return link->mass();}
   void length(double len)
   {
      left->qp(-len/2,0,0);
      right->qp( len/2,0,0);
   }
   double length() const
   {
      return DX(right,left,right).val();
   }
   Marker* i() const {return left_pin->i();}
   Marker* j() const {return right;}
   void i(Marker* pJ){left_pin->i(pJ);}
};

// ########################################################

class Chain : public Adams
{
   double total_mass, total_length;
   int linkCount;

   std::vector<Link*> allLinks;
   Joint  *right_end_pin;

public:
   Chain(Adams* parent)
      : Adams(parent), total_length(0), total_mass(0)
   {
      right_end_pin=new Joint(parent);
      right_end_pin->jtype(Joint::REVOLUTE);
      links(1);
   }

   ~Chain()
   {
      std::vector<Link*>::iterator i;
      for(i=allLinks.begin(); i!=allLinks.end(); ++i)
         delete *i;

      delete right_end_pin;
   }

   void links(int number_of_links){
      if(number_of_links==linkCount) return;
      if(number_of_links<1)
         throw Msg::ErrorMsg("Chain must one or more links");

      Marker* pI=i();

      std::vector<Link*>::iterator ln;
      for(ln=allLinks.begin(); ln!=allLinks.end(); ++ln)
         delete *ln;

      allLinks.clear();

      Link* previous=new Link(this);
      allLinks.push_back(previous);

      for(int j=1; j<number_of_links; j++)
      {
         Link* next=new Link(this);
         allLinks.push_back(next);
         next->i(previous->j());
         previous=next;
      }
      right_end_pin->i(previous->j());

      mass(total_mass);
      length(total_length);
      i(pI);
   }

   void mass(double m)
   {
      double  mass_each = m/linkCount;

      std::vector<Link*>::iterator i;
      for(i=allLinks.begin(); i!=allLinks.end(); i++)
         (*i)->mass(mass_each);
   }

   void length(double l)
   {
      double  length_each = l/linkCount;
      std::vector<Link*>::iterator i;
      for(i=allLinks.begin(); i!=allLinks.end(); i++)
         (*i)->length(length_each);
   }

   void i(Marker* imar){ allLinks[0]->i(imar);}
   void j(Marker* jmar){ right_end_pin->j(jmar);}
```

```
   Marker* i(void) const
   {
      if(allLinks.size()==0)
        return 0;
      else
        return allLinks[0]->i();
   }

   Marker* j(void) const { return right_end_pin->j();}

   class Span : public Expression1D
   {
   public:
      Span(Chain* chain)
        : Expression1D(DM(chain->i(),chain->j())){}
   };

};

// ########################################################


main()
{
   Model mod1;
   Chain ch1(&mod1);

   Part p1(&mod1), p2(&mod1);
   Marker m1(&p1), m2(&p2);

   ch1.mass(234);
   ch1.links(100);
   ch1.i(&m1);
   ch1.j(&m2);

   cout << Chain::Span(&ch1).val() << endl;
}
```