COMPUTING INTEGRITY
INCORPORATED
60 Belvedere Avenue
Point Richmond, CA 94801-4023
510.233.5400 Sales
510-233.5444 Support
510.233.5446 Facsimile

A Path to Model-To-Code Translation in ABL

16 March 2011

Thomas Mercer-Hursh, Ph.D.

Model-to-Code translation has the potential to provide the high development productivity and responsiveness required to meet modern business requirements. This paper reviews issues of completeness, action language, realized code, user interfaces, alternate models, and frameworks. This discussion leads to a proposed recommended path for the development of these tools.

Background

In a previous document[1] I have argued that there is a key opportunity for Progress Software to dramatically enhance the productivity and responsiveness of ABL systems by the development of tools to implement Model-To-Code translation. Such tools can dramatically speed new development while improving fit to requirements. These tools also provide rapid response at dramatically lower effort when there is need to modify an existing system to fit changed requirements. Following, I review issues important for determining the desired characteristics of such a tool, resources available for input to the design, and opportunities for supporting features. I then propose a development path by which such features can be added to the ABL toolset.

Completeness

The greatest benefit of Model-to-Code translation comes from 100% translation, i.e., one works entirely in the model and does no hand creating or modification of the code. Nevertheless some users are likely to be most comfortable with only partial code generation, i.e., generating shells and some standard components, but manually filling in the details of methods and other logic. Moreover, delivering 100% complete translation might be an ambitious goal for a first release, especially if one wants early availability. So, it is reasonable to ask what one can do with less than complete translation and what shortcomings there are in delivering such a tool.

The normal way of providing less than complete translation is to provide reverse engineering. I.e., one generates code, modifies that code manually in an editor, and then reads the modified code, detects any changes, and captures the added code as text associated with the methods, new properties, etc. in the model. Then, on the next generation, this text is included in the generated code, thus recreating the code as previously modified. This approach presents several problems, including:

1. Synchronization of model and code is dependent on the user following a well-specified workflow. Failure to check in modified code or working on the model without generating new code will lead to the code and the model falling out of sync. If this persists and the versions diverge, it can be difficult to re-synchronize the two.

[1] Rapid Business Change and ABL Productivity, 16 March 2011.

2. The workflow requires moving back and forth between contexts - working on the model in the UML editor, generating code, moving to Architect to make modification, returning to the editor to reverse engineer the code, etc.
3. Code written outside the model is not coordinated with the model. Thus, it can easily contain material which should have been expressed in the model itself or which is at variance with the structure of the model. Since the code is treated merely as text, there is no way to resolve such discrepancies or even to detect them systematically.
4. Manually written code is likely to follow the implementation model of the developer and thus including this code will embed that implementation in a model which one would prefer to remain implementation independent. This reduces the flexibility of future evolution of the model and translation.
5. If the capabilities of the tool increase in the future or the user desires to make greater use of the tool's capabilities, there is no obvious way in which to evolve the existing model with its embedded code to a new model in which some aspect of what is now in the code is moved to a model element.

Together, these make a strong argument against anything less than a complete solution and consistent 100% Model-To-Code operation, but nevertheless, we have the issue of both potentially wanting to deliver a partial solution and having some users who will prefer a partial solution.

Action Language
When the concept of 100% Model-to-Code translation is introduced to those not familiar with it, the reaction is often skepticism that 100% code generation for any meaningful application from "only pictures" is possible. One element which helps make this capability more understandable, but which is not well known, is Action Language. Thus, before exploring the completeness issue further, it is worth while reviewing what one does to achieve a complete solution beyond "drawing pictures", i.e., beyond the graphical elements of UML. This question also raises some interesting issues about how code is included in the model.

The Object Management Group[2] (OMG), which defines the standard for UML, has defined a standard syntax for an Action Language. The purpose of Action Language is to supplement the relations and actions indicated by the more familiar UML diagrams with an implementation-independent language for use in describing algorithms, i.e., any procedural sequence of operations which is necessary to the correct functioning of the target software. There is as yet no OMG standard Action Language, only a specification for the syntax. Each vendor of translation tools has their own Action Language[3].

One idea, previously advanced by John Green and I[4], is that one could construct an Action Language which was essentially a subset of ABL. This would preserve the high level character and would greatly simplify both Model-to-Code and reverse engineering. This approach might also provide a more natural transition for existing ABL developers. This might be consistent with reverse

[2] http://www.omg.org/
[3] See Executable UML: A Foundation for Model-Driven Architecture, Steven J. Mellor & Marc J. Balcer for a general discussion of these concepts.
[4] Personal communication with John Green of Joanju software, source of Proparse, ProRefactor, Analyst, and other tools for analyzing and modernizing ABL code.

engineering modified ABL code back into the model as discussed above, but is also subject to the cautions cited above.

The intention of UML is that models remain independent of implementation specifics as much as possible. Use of a subset of ABL as an Action Language would violate this principle both in that some ABL syntax would not readily be translatable into any other language and that there would be a strong temptation on the part of experienced ABL programmers to “solve” parts of the problem in the Action Language segments which properly should be solved elsewhere in the model.

One of the things which may surprise an ABL programmer is the apparent “primitiveness” of Action Languages (See, for example, Platform Independent Action Language[5], BridgePoint[6], or SMALL[7]). The secret to understanding its apparently simple character when compared to ABL lies in two factors. One factor is the desire for the Action Language to be highly abstract from the implementation, i.e., for the syntax of the Action Language to be independent of the syntax of the target language or anything else about the target platform. The other factor lies in understanding the simple job which is required of an Action Language. In most cases, an Action Language fragment is nothing more than the specification of what happens in a particular method and thus is inherently quite simple.

The best Action Language to use for ABL translation will need further exploration, but it seems likely that the best choice will be and existing formal Action Language like like SMALL, BridgePoint, or PAL[8] rather than an ABL subset for three significant reasons:

1. Formal Action Languages are language neutral;
2. Formal Action Languages are implementation independent; and
3. Use of an existing Action Language may make it possible to use existing translation tools.

To be sure, the typical user may prefer a subset of ABL as something which was already familiar, but the entire language of a true Action Language can be documented on a single side of a single sheet of paper, so there is not a huge investment in learning required. Moreover, working in an Action Language will reduce the temptation to simply “start writing code” resulting in a model which is tied to a specific implementation model and in which important information is represented in the AL instead of in the model structure.

Once introduced to the concept of Action Language, a skeptical person may think one is going to have just as much code in Action Language as one would have written without the UML. One might then wonder whether anything is really being gained. This reaction may be likely in ABL programmers who are certain that the seemingly “primitive” nature of existing Action Languages will result in even more Action Language code than there would have been ABL. In fact, a great deal of the application code comes from the model itself and the Action Language is only required for limited purposes. One very complex device driver of which I am aware was about 750,000 lines of code altogether, but had only about 10,000 lines of Action Language[9].

[5] http://www.pathfindermda.com/resources/whitepapers.php
[6] http://www.ooatool.com/docs/BPAL97.pdf
[7] http://www.ooatool.com/docs/SMALL97.pdf
[8] PAL is the working name for Platform-Independent Action Language as described in the document cited above.
[9] Personal communication from H.S. Lahman, a 20 year veteran of Model-to-Code translation in complex real time applications.

Practical Issues of Completeness

Returning to the question of completeness of the initial release of Model-to-Code functionality in ABL, it appears that there are somewhat contradictory strategies. One strategy aims at an initial release which will provide less than 100% Model-to-Code translation and which uses reverse engineering to incorporate manually coded implementations into the model. This approach may be more comfortable to some developers, especially as a first step, but has a number of hazards in the quality of the model and no clear path to evolving a more complete translation. This approach seems to point to the use of ABL as an "Action Language", since the reverse-engineered code would be filling the rôle of the Action Language, albeit in an implementation dependent way.

The other strategy aims at an initial release which provides 100% Model-to-Code translation. This approach seems to point to the use of an formal Action Language, not ABL, possibly an existing language in order to articulate with an existing translation engine. User acceptance would require more persuasion with this approach, but the tangible benefits are also substantially higher, providing a more compelling story.

While it may seem intuitive that the incomplete translation would require less work than complete translation, this may not be the case since the complete translation approach may take advantage of existing technology while the incomplete translation implies building all tools from scratch. It might seem like some of Phil Magnay's work[10] on ABL with Enterprise Architect would provide a foundation for a solution involving reverse engineering, but issues about that work may mean that it doesn't provide a foundation which would meaningfully speed development.

Therefore, I suggest there should be a two pronged attack on development. One branch will assume that a complete solution is the preferred solution and will investigate existing Action Languages and translation tools to evaluate whether they would form a sound and acceptable basis for an implementation along with a projection of the time and costs involved in such an implementation. The other branch will survey potential users about acceptability and preferences relative to user preparedness to use a 100% solution and what factors would influence their attitudes and acceptance. This branch will also explore potential avenues to reverse engineering as well as any existing efforts at Model-to-Code translation for ABL.

Articulation with "Realized Code"

Systems utilizing Model-to-Code translation for some of the overall system often also include pre-existing code, other software packages, purchased components, etc. In these cases, it is not possible to model that part of the system, often because one does not have source code. Such pre-existing code external to the modeled code is called "realized code", i.e., code that has already been realized into working form by some process other than that being used to produce the modeled code.

While the historical distinction of ABL was built on the idea that it was a 4GL in which one could write 100%[11] of an application, as the computing world changes, this may no longer be desirable. One is not speaking here of a return to 4GLs in which a portion of the application was written in C,

[10] http://communities.progress.com/pcom/community/psdn/modeling?view=documents
[11] Except for unusual cases where the system included device drivers and such as part of the overall application.

since that too is code that might be modeled, but rather to the use of applications and tools which are given responsibility for some part of the application.

One of the obvious current examples of including non-ABL components in an overall solution is externalizing the workflow of an application into a product such as Savvion BPM. Without Savvion, one would model the workflow as part of the application. With Savvion, one wants Savvion to be in charge of workflow. One might think this would lead to difficult orchestration of two separate solutions, but, in fact, there is a very natural way in which Savvion and ABL code can be integrated. Any place in the application where one wants to invoke Savvion one is going to send a message from the ABL. Any place that one wants to have Savvion invoke the application one will need to have a facility in the ABL to receive a message. This is no different really than sending or receiving such messages from an ABL component except that in this case the recipient or source is a black box of external code.

If there is an articulation issue with Savvion, it is that one wants to model processes in the UML to provide a complete model, but one also needs to model these processes in Savvion to generate the code in the Savvion system. There is some potential for importing and exporting BPMN models with Savvion that will warrant further exploration. With good integration, actual import and export may not be required. One might consider similar integration with Apama.

User Interfaces (UI)
User interfaces present a special problem for Model-To-Code translation because they involve appearance as well as knowledge and behavior. Many types of user interface also include significant amounts of realized code, e.g., the .NET controls in an ABL GUI for .NET interface. With a legacy architecture having UI, business logic, and data access all intertwined, it could be difficult to create models which had the desired UI, especially since one would not have the advantage of visual design tools. However, with modern architectures, there is a strong separation between the purely visual parts of the UI and the part managing logic and state. In some UI technologies, e.g., RIA, significant parts of this subsystem may even be running on the server.

Thus, the solution to handling one or more user interfaces in the context of Model-to-Code translation is to treat the visual part of the UI as realized code similar to the way in which one articulates to Savvion. The UI is a subsystem of realized code created in a separate tool which sends and receives messages to the rest of the application. In some cases, e.g., Java versus .NET OpenClient, one may use the same ABL components from the model and freely substitute the desired UI. In other cases, some alternate ABL components may be required specific to the UI, but these should be limited since most components should be independent of the UI and thus common across all UIs. This clear separation between the visual part of the UI in realized code and the rest of the application in modeled code should emphasize good separation such as is specified by OERA.

Alternate Models
One notable characteristic of the ABL development community is lack of a dominant programming model. ADM has probably been the model with the widest adoption, albeit in multiple versions, none of which are really OERA compliant or Object-Oriented (OO) as one would expect to get from a tool of this sort. Various models have been offered for OERA applications, although most have

limited publicly available documentation. How then is one going to accommodate this diversity of opinion in a Model-to-Code translation tool? Or should one?

One can argue, meaningfully, that many of the models now in use are not good models in a number of respects. Moreover, very few traditional models are based on OO[12]. One might wonder if providing a single, well considered model which implemented OERA principles and illustrated adherence to good OO principles would not be sufficient. While defensible, it also could be true that support for alternate models would increase acceptance. One important question one might ask is whether it is possible to support multiple models without undue burden.

This question really has two components – one about whether the transformation is open for the user to customize and the other about whether more than one model is supported out of the box. There is some need to support alternate models in any Model-to-Code translation simply because different approaches are often needed in different parts of an application for performance or compatibility issues. This mechanism is often called "colorization" or model markup. It means that properties are attached to the model which are not a part of its functional structure, but which tell the translation engine to do one thing versus another during the translation process. Naturally, providing such options add to the complexity of the translation templates. Thus, it would be possible to provide alternate models via markup, but it would add substantially to the work required to create the model.

The question of openness is also difficult, as illustrated by the history of ADM frameworks. Those who worked entirely with the ADM framework as shipped were often able to move to a later version without a great deal of difficulty. Those who customized the framework, however, (especially common with earlier versions) often had considerable difficulty in moving to new versions. While they could run the application with the old framework and the new version of Progress, they could not take advantage of the features of the new version of the framework running that way. If people customize the translation templates, we are likely to see similar issues here. However, at the same time, the ability to customize can often be the difference between acceptable and not.

It has been suggested that most of the difficulty experienced with user customization of framework components comes because a new framework is typically available only yearly. Consequently, a user feels a need to make customizations because he or she can't wait a year to see if desired functionality will be added. Meanwhile, the team responsible for maintaining the framework is modifying the framework according to some master plan which is unknown to the users. The result is divergent development over a substantial period and the consequent difficulty of resolving the two versions.

One solution to this dilemma is continuous availability of new versions, i.e., to make new versions available as rapidly as they are created, making well documented changes in individual components rather than sweeping restructuring whenever possible. This might be done through an on-line community which also provided a feedback mechanism by which individuals could contribute candidate changes for possible inclusion and where they could make requests for options in the

[12] This paper assumes the desirability of an OO target implementation. Anything else will not match well with UML modeling. There are good reasons to believe that OO is a superior approach in any case. See http://www.cintegrity.com/content/Object-Orientation-Why-When-and-How for a discussion of the advantages of OO.

overall framework. One should also have a visible planned direction of development so that users can anticipate future directions of development. Such a community would also help to publicize the tool.

Frameworks
Modern development often involves development relative to a framework responsible for providing basic services. This framework may be purchased, in which case it is treated as realized code. But, in the current construct any framework would be provided with the tool and would be written in ABL. While it is possible to create such a framework by conventional coding mechanisms, given a high quality Model-to-Code translation mechanism, there is no reason not to create the framework using the same facility. Indeed, constructing the components of such a framework would provide a good test bed for the translation engine and rules. Moreover, the model used to produce the framework would provide an excellent example to potential users of the tool. In addition, using this approach would allow the framework to evolve gracefully with changes in technology.

The big danger here is similar to that discussed previously, i.e., customization of the framework by users at the same time that the core framework is evolving. Again, continuous publication can mitigate this problem. Also, development of a community of users has the potential to yield valuable contributions as users enhance the capabilities as well as create their own special versions that will not have general applicability.

The key to allowing graceful evolution will be to provide clear separation of concerns and a well-defined interface to each package in the framework. This is good OO design anyway, but will be particularly important on common components which may be subject to customization requirements. By providing a very clean isolation of the responsibility in each package and a simple, but complete interface, one separates the use from the implementation so that users can be free to use alternate components tailored to their specific needs and yet preserve the same interface and thus not be subject to version disruption.

One should note an established principle of good Object-Oriented design is that subsystems should be highly coherent, dedicated to a single purpose, and loosely coupled with each other. One is thus able to treat any subsystem as a "black box" when working in any other subsystem and such subsystems can be separately modeled, and separately created. One sees that principle here for both the visible part of the user interface and the framework(s), but in practice one would also see it in other subsystems like the data access subsystem. E.g., ideally, the data access subsystem would be the only one aware of the existence of the database and its table and field artifacts.

Source of Technology
There are a number of different options for the source of this technology. Some of it will be new and unique to this new tool and so will have to be developed. But, there are options for open source, license, and resale agreements for contributing parts of the technology which can substantially alter both the amount of work required to produce a working product and the quality and the power of the result.

For example, the apparently natural choice of focus for the UML modeling itself would seem to be Enterprise Architect (EA) because of its strong presence in OpenEdge sites now using UML. But, EA has no Action Language and its MDA translation capabilities do not appear to have been robustly tested in similar scope endeavors, although it is believed that Progress Professional Services has done some work which should be considered. Also, while EA has an Eclipse interface, it is not an Eclipse hosted tool, which limits the degree of integration with OpenEdge Architect (OEA). This suggests that an Eclipse-based tool may be worth considering since good integration may trump established market share, particularly considering that the number of existing EA users in the OpenEdge world is apparently not particularly large.

Another interesting question is whether one tries to build on the MDA capabilities in an existing UML tool or considers technology specifically directed at Model-to-Code translation. I'm not sure that any UML tool has an Action Language standard, which implies a rather large investment in new technology development up front. Whereas, if one were to use an existing translation engine one might be able to shortcut that development considerably. In addition to accelerated development, use of a developed tool might substantially enhance the quality and power of the result through such features as tools for model checking and for actual execution of models, a sort of very high level debugger.

In particular, it may be possible to further accelerate development by using one of the packages in the selected translation tools for another language such as Java. While there are clearly going to be places where the ABL implementation will differ substantially from the Java implementation and even the packaging will change, the existing implementation will provide a map on to the model elements which need consideration and guidance for how that mapping turns into language elements in a known language. In many cases, the change is likely to be more in the text produced than in the mapping.

Model-to-Code, Model-to-Model, and Code-to-Model
The primary focus for achieving higher levels of productivity in ABL is Model-to-Code. But, one has to recognize that, as such, it is primarily of use for new development, i.e., contexts in which it is possible to start with use cases and requirements and work toward code which will implement those use cases and requirements. In this process, there are some obvious opportunities for Model-to-Model transformations as well, such as converting an evolved domain model diagram into a class diagram. However, one also has to recognize that there is a great deal of ABL code already in the world and that most of that is not even object-oriented and thus has a somewhat imperfect relationship to even the most concrete aspects of UML.

There are essentially two different routes one could take in trying to deal with existing code and a migration toward a Model-to-Code environment. One route is to approach the problem as a modernization effort, i.e., one in which one will attempt to create a model from the existing code, modify that model to make it suitable for modern development, and then generate code from that model to create the new application. This approach will produce the best ultimate code and the best environment for on-going development, but it is also labor intensive and thus expensive.

Creating a model from code is as difficult as it is, not only because most of the code is not OO and is often not well structured, but because there is an inherent loss of information when one moves from abstract to concrete. The concrete code which comes from a Model-to-Code translation doesn't include the abstract information about use cases, requirements, and intent which are in the model.

Neither does legacy code and such information may not even have been explicitly expressed either when the code was originally written or at any time since.

There have been efforts such as my ABL2UML[13] and the derivative work ABLaUML[14] which will build UML models from existing ABL code, regardless of programming model or style. These models are quite extensive and are very powerful analytic tools, but do not currently include the actual ABL code from the source, although this is a planned extension. This is, however, a component model, not a class model, and certainly does not have any of the abstract layers one associates with a full Model-to-Code project. It is likely, however, that this is a valuable analytic step even if one cannot directly create an analysis model from this import. Moreover, it is a valuable analysis model even without utilizing any Model-to-Code features.

Thus, it seems valuable to consider how this work can be extended, both to include more information in the model for analysis and to create Model-to-Model translations which can assist in building the kind of analysis model one would like for Model-to-Code work. While considerable human effort is certain to remain, it seems likely that tools can be created which will facilitate the process and reduce the work. For example, the ABL2UML component model includes links between each program and subprogram component and any data elements referred to in that component including the fields involved, read/write access, where clauses, etc. This means that one can start with any given table and quickly identify any code component which uses that table, the mode in which it is used, and thus also any business logic which might be applied to that table. Since tables are often considered as possible first pass candidates for classes in the class model, this provides a semi-automated way to identify all the properties and behavior associated with that class. Since the links are quite fine grained, if one decides to analyze a given table into a super class with multiple subclasses, a common refactoring, one could rapidly identify which components related to which properties of which subclass. There are undoubtedly other ways in which either reports or actual Model-to-Model translations could assist in building a base model from which to create Model-to-Code translations.

The other approach for dealing with existing code would be to provide Model-to-Code translations for the component model itself. This would allow someone who was not interested in making the transition to full Model-to-Code (or unwilling to make the investment) to still analyze the component model, make changes, and output revised code. Some research is needed to see how much interest there would be in such a capability. Given that we add code to the model, it is probably not difficult and would facilitate some kinds of structural refactoring, but it may be that the interest is too low to justify the effort, even though I don't believe the task is particularly difficult.

Recommended Approach for PSC

Providing a complete MDA solution is obviously a non-trivial endeavor, so it is important to both pick good goals and to identify useful delivery stages[15] in order to insure that meaningfully useful utilities are delivered to developers with each release. Thus, the first phase of this project should be to determine these goals and stages. To this end it is recommended that an evaluation and catalog be made of both the PPS CloudPoint offering and, if possible, the iMo offering to identify what each has accomplished, the shortcomings of each, and the degree to which the tools involved are a sound

[13] See http://www.oehive.org/ABL2UML
[14] See http://www.oehive.org/node/1821
[15] T4BL might be an example of a well-intentioned idea that was not packaged into a useful delivery stage.

basis for further work. This review should also include any related products and efforts such as ABL2UML and the publicly released tools by Phillip Magnay.

In addition to reviewing existing ABL-specific tools, a review should be made of possible sources of technology including both UML tools and translation tools. The review of UML tools would clearly include Enterprise Architect for reasons mentioned above as well as its rôle in some of the existing ABL tools. The review should also include UML tools which are built on Eclipse, including any which are compatible with identified translation tools. The survey of translation tools can be limited to those which have the potential for adaptation for use with ABL[16].

These reviews will determine what is potentially available to help kickstart or accelerate the effort and what needs to be done from scratch. As discussed above, this should be a two pronged review – one branch exploring 100% Model-to-Code translation as the deliverable for the first release and the other exploring the possibility of partial solutions. This review needs to both include technical possibilities and potential as well as market acceptability and interest.

From this review we will construct a phased delivery schedule and goals along with addressing such issues as packaging. For example, if one were to decide not to use existing translation tools and a partial translation solution were desired or acceptable, one might arrive at a schedule such as:

- Phase one might be to provide an OEA-based equivalent to ABL2UML including code plus MDA from class diagrams to create object shells and reverse engineering to read code from filled in shells to the model.
- The second phase might include some model-to-model transformations to build analytic models from the ABL2UML component model, some model-to-model translations for analytical to design models, and extension of the MDA to include Activity and Sequence diagrams in the MDA with some configuration options as to how to treat code in reverse engineering.
- The third phase might then fill out these transformations and add an Action Language.

Alternatively, if one of the translation engines was sufficiently promising, one might aim for an initial release which was more of a 100% Model-to-Code tool and subsequent releases might be focused on enhancing integration and features. One casual estimate is that basic 100% Model-to-Code translation could be accomplished in as little as 5 person months of effort[17]. More detailed analysis and prioritization, as well as determining the resources available, will need to occur to determine what is feasible and appropriate in each stage.

I believe that a critical element in the acceptance and perceived utility of this capability will depend on its configurability. While it is certainly true that there are many in the ABL community who could use strong guidance on best practice, e.g., with respect to good OERA architecture, this tool will not be used if a company cannot easily personalize it to suit local needs and tastes. Not only is this essential in the initial release, but subsequent releases must support earlier choices and work, i.e., it is important to avoid a scenario such as we have seen with ADM and ADM2 in which a shop that does significant customization then places themselves in a compromised position for accepting new versions of ABL, often having to go through lengthy test and conversion process to resolve the old version, new version, and customizations. One possible option toward this goal might be to place the transformations themselves in an open source community where they could continuously evolve.

[16] Many translation tools are not template based and thus would be difficult to adapt to ABL since the changes would require altering the actual translation engine itself.
[17] Personal communication from Peter Fontana, CEO of Pathfinder, the vendor of one of the translation tools discussed.

PSC could contribute to that effort, but then always use transformations from that community and support all variations in the community. As noted above, this approach might also be used for framework components, both to ease version transitions and to encourage contributions from the user community.

Conclusion

It is proposed that Model-to-Code translation represents a major opportunity for increased productivity in ABL and one that is very much in tune with developments for other languages. Implementing such translation and making it available to ABL developers can boost productivity and increase quality and nimbleness, improving competitiveness with other development tools and enhancing the fit of OpenEdge to the rapidly changing requirements of modern business. It is suggested that the project begin with a review of existing tools, both within and outside the ABL community along with market surveys of acceptable and desirable options. A combination of technical feasibility and market acceptance will determine whether the initial target should be 100% Model-to-Code translation or a more modest target of partial generation and reverse engineering. This review will also help to determine issues related to the choice of Action Language. Good OO principles will serve to guide the handling of realized code, user interfaces, and frameworks. Proposals have been offered to optimize configurability and support of alternate models while minimizing the potential for version lock-in.

With a largely dedicated resource and good access to sources, it is likely that the review phase of this project would only take 2-3 months. If the option selected was to proceed with the use of an existing translation engine, it might be only 5 months from then before a workable 100% Model-to-Code translation map could be established for ABL. This would probably need to be accompanied by 2-3 months of associated framework development. Work on integration and packaging could probably proceed in parallel. Thus, it is feasible that a working product could be ready for beta test within a year.

Given the potential for as much as 10X gains in productivity for creating new applications while simultaneously improving quality and fit to requirement, the argument for pursuing Model-to-Code translation for ABL is not only compelling, but highly consistent with Progress' historical goals. Even greater potential exists for nimble responsiveness to changing business requirements, a critical attribute in the contemporary market. Possible Code-to-Model tools have the potential to round out the offering and assist users in analyzing and migrating existing ABL applications.