NOTICES

Destroy this report when it is no longer needed. DO NOT return it to the originator.

Additional copies of this report may be obtained from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, VA 22161.

The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The use of trade names or manufacturers' names in this report does not constitute indorsement of any commercial product.
Computer-Aided Maintenance for Embedded Real-Time Software

CPT David A. Dampier, MAJ Ronald B. Byrnes, and LTC Mark R. Kindl

U.S. Army Research Laboratory
ATTN: AMSRL-SC-IS
115 O'Keefe Bldg
Georgia Institute of Technology
Atlanta, GA 30332-0800

ARL-TR-839

This paper was originally presented at the Army Science Conference in June 1994.

Approved for public release; distribution is unlimited.

Army software systems get more complex as Army hardware gets more sophisticated. Life-cycle costs for these systems are expected to exceed $35 billion over the life of current systems. Current software development and maintenance practice will soon be insufficient; hence, to keep pace, computer-aided methods must be adopted.

One such method, computer-aided prototyping, improves software development and benefits costly software maintenance, by taking advantage of automation and decreasing costly human involvement. In computer-aided prototyping, software prototypes written in a specification language are translated into some high-level programming language, like Ada, compiled, and demonstrated to the customer. Based on customer comments, the prototype is quickly updated and demonstrated to the customer again. This iterative process continues until the customer and designer agree on the prototype design. The prototype is then used as the baseline version of the final system.

Change-merging is a formal method which allows multiple design teams to work on different enhancements to the same prototype. These enhancements can be made independently and combined automatically using our change-merging algorithm. As long as the independent enhancements do not conflict with one another, the result of the change-merge is a prototype with the capability of all the enhancements. This method is applied to the maintenance of different versions of existing systems by making enhancements to the baseline version and automatically integrating these changes into each fielded version. This will drastically reduce the time required to update software systems in the field.

software, automation, computer-aided prototyping, maintenance, formal models, software engineering, software merging, change integration, slicing

UNCLASSIFIED UNCLASSIFIED UNCLASSIFIED

UL
INTENTIONALLY LEFT BLANK.
TABLE OF CONTENTS

| LIST OF FIGURES | ................................................................. | v |
| 1. INTRODUCTION | ................................................................. | 1 |
| 2. CHANGE-MERGING | ................................................................. | 2 |
| 3. SLICING OF PROTOTYPES | ................................................................. | 3 |
| 4. SLICING METHOD FOR CHANGE-MERGING OF PROTOTYPES | ................................................................. | 6 |
| 5. SUMMARY | ................................................................. | 9 |
| 6. REFERENCES | ................................................................. | 11 |
| DISTRIBUTION LIST | ................................................................. | 13 |

Distribution List

| Accession For | |
| NTIS CRA&I | ☑ |
| DTIC TAB | ☐ |
| Unannounced | ☐ |

Justification

By ____________________________  
Distribution/Availability Codes

<table>
<thead>
<tr>
<th>Dist</th>
<th>Avail and/or Special</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-1</td>
<td></td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1.</td>
<td>Change merge algorithm</td>
</tr>
<tr>
<td>2.</td>
<td>Change propagation through multiple versions of a software system</td>
</tr>
<tr>
<td>3.</td>
<td>PDG for the Patriot missile defense system, version 1.1</td>
</tr>
<tr>
<td>4.</td>
<td>The slice of Patriot’s PDG with respect to the stream tactical_status</td>
</tr>
<tr>
<td>5.</td>
<td>PDG for the Patriot missile defense system, version 1.2</td>
</tr>
<tr>
<td>6.</td>
<td>PDG for the Patriot missile defense system, version 2.2</td>
</tr>
<tr>
<td>7.</td>
<td>Preserved part of the PDG for versions 1.1, 1.2, and 2.2 of the Patriot system</td>
</tr>
<tr>
<td>8.</td>
<td>Affected part of the PDG for version 1.2 of the Patriot system</td>
</tr>
<tr>
<td>9.</td>
<td>Affected part of the PDG for version 2.2 of the Patriot system</td>
</tr>
<tr>
<td>10.</td>
<td>Result of change-merging versions 1.1, 1.2, and 2.2 of the Patriot system</td>
</tr>
</tbody>
</table>
INTENTIONALLY LEFT BLANK.
1. INTRODUCTION

Future Army software systems will continue to increase in complexity as long as Army hardware continues to get more sophisticated and mission needs more challenging. Life-cycle costs for these systems will continue to rise. It is estimated that life-cycle costs of the over 350 software systems that the U.S. Army Materiel Command is currently responsible for will exceed $35 billion. The maintenance costs for these systems is estimated at between $17–25 billion (Piersall 1994). Current software development and maintenance practice will soon become insufficient to handle the evolution of this software at a reasonable cost; hence, more reliable and efficient, computer-aided methods must be adopted in order to keep pace.

One such method is computer-aided prototyping. This method not only improves software development activities but benefits software maintenance as well. The benefits provided by computer-aided prototyping to software maintenance start in the initial phases of system development. If current methods are used to develop and maintain this software, software costs will continue to rise, thus counteracting the decreasing budget trend. What we need are software development and maintenance methods which take advantage of automation and decrease costly human involvement.

Computer-aided prototyping is one such method that reduces initial development time while allowing the development software to be maintained using the same prototyping tools. In computer-aided prototyping, quickly built and iteratively updated prototypes of the intended system are demonstrated to the user. Each successive iteration of the prototype resembles more closely the final intended version of the software. The final accepted prototype is a very close approximation of the intended software system. Since the prototype is written in a specification language translatable to a high-level programming language such as Ada, the code produced by the prototyping environment can be used in the final software product.

This same prototyping environment can also be used to perform maintenance on a production version of a software system. Since translation mechanisms may also be used to translate high-level programming code into the prototyping language (Altizer and Berzins 1992), a production version of the software system can be translated into the prototyping language, loaded into the prototyping environment, updated, and translated back into the high-level programming language. This is useful because the prototype description is significantly simpler than the production code if the prototype is expressed in a notation tailored to
support modifications. In addition, the software tools in the computer-aided prototyping environment can help carry out the required modifications rapidly (Luqi 1989). This research impacts both of these roles for rapid prototyping. We concentrate on application of this technology to software maintenance, since the majority of maintenance costs are associated with enhancement and refinement rather than correcting faults (Piersall 1994).

2. CHANGE-MERGING

Change-merging is the process of automatically combining the effects of several changes to a software system (Dampier, Luqi, and Berzins 1994). Early version control systems such as the source code control system (SCCS) (Silverberg 1992) and the revision control system (RCS) (Tichy 1982) provide primitive change-merging facilities based on string editing operations on the source text without considering the effects on program behavior. However, automated tools must provide guarantees to designers regarding program behavior. Semantics-based change-merging seeks to construct a program whose behavior contains all of the significant changes made in the modified versions while maintaining any behavior common to all three versions.

We have constructed a model and algorithm for automatically updating different versions of a software prototype with changes made to the base version as shown in Figure 1. These changes are applied to the base version and propagated through all of the different production versions by using the change-merging algorithm. This algorithm accepts a base version and two modified versions of the program and uses program slicing (Weiser 1984) to find the part of the base version preserved in both of the modifications, as well as the parts of the modifications which are different from the base. The algorithm then combines these three pieces into a merged program containing the functionality peculiar to each of the modified versions. This method is applied to the change propagation problem by using the base version of the program as the base for the change-merge, the production version to be updated as one modification, and the changed base version as the other modification as shown in Figure 2. As long as the change made to the base does not conflict with the production version, the result will be an updated production version containing both its original functionality and the update.

A working version of this change-merge algorithm has been developed for the computer-aided prototyping system (CAPS) (Luqi 1989), using the prototyping system description language (PSDL) (Luqi 1988) as its base language.
Algorithm Change-Merge(BASE, A, B : psdl_program) return psdl_program is
begin
    Change-Merge the Specifications;
    --Each individual component of the specification is change-merged according to the rules
defined
    --by Dampier (1990, 1994).
    Change-Merge the Graphs;
    Build Prototype Dependence Graphs for each version;
    Calculate the Preserved Part of BASE in both A and B;
    Calculate the Affected Part of both A and B with respect to Base;
    Graph-Union the Preserved Part with the Affected Parts of both A and B;
    Check Correctness of Merge using Compare Graphs;
    Change-Merge the remainder of the Implementations;
    --The stream and timer declarations along with the control constraints are merged according to

Figure 1. Change merge algorithm.

Figure 2. Change propagation through multiple versions of a software system.

3. SLICING OF PROTOTYPES

The change-merging algorithm is based on the ability to partition a portion of the prototype's
functionality using prototype slicing. The slice is constructed by analyzing the prototype's dependence
A prototype dependence graph (PDG) for a prototype $P$ is an augmented, fully expanded, PSDL implementation graph $G_P = (V, E, C)$. The set of vertices, $V$, has been augmented with an external vertex, $EXT$. The set of edges, $E$, has been augmented with an edge from each vertex without an outgoing edge to the $EXT$ vertex. Furthermore, a timer dependency edge is added from $v_i$ to $v_j$ when $v_i, v_j \in V$ and $v_i$ contains timer operations that affect the state of a PSDL timer read by $v_j$. A slice of a PSDL prototype $P$ with respect to a set of streams $X$, $S_P(X) = (V, E, C)$, is a subgraph of the PDG, $G_P$, which includes the part of $P$ affecting the values written to that set of data streams. A slice is constructed as follows:

1. $V$ is the smallest set that contains all vertices $v_i \in G_P$ that satisfy at least one of the following conditions:
   
   (a) $v_i$ writes to one of the data streams in $X$.
   
   (b) $v_i$ precedes $v_j$ in $G_P$, and $v_j \in V$.

2. $E$ is the set that contains all of the data streams $x_k \in G_P$ which satisfy one of the following conditions:
   
   (a) $x_k \in X$.
   
   (b) $x_k$ is directed to some $v_i \in V$.

3. $C$ is the set that contains all of the timing and control constraints associated with each operator in $V$ and each data stream in $E$.

An example of a PDG is shown in Figure 3. This graph is the implementation graph for a prototype of the Patriot missile defense system. This prototype has five operators and eight streams. A slice of this operator taken with respect to one of those streams will contain everything in the prototype that affects the values written to that stream. An example of the slice of this prototype with respect the stream `tactical_status` is shown in Figure 4.
Slices are valuable in change-merging because they partition the prototype into disjoint parts that can be considered in isolation without worrying about the rest of the prototype. This guarantees that a slice taken from one prototype and placed in another where that slice is still well-formed will behave in precisely the same way as it did in the first prototype (Dampier, June 1994).
4. SLICING METHOD FOR CHANGE-MERGING OF PROTOTYPES

We use prototype slicing for change-merging in much the same way program slicing was used in Horwitz integration method (Horwitz, Prins, and Reps 1988). We use slicing to calculate the preserved part of the base version and affected parts of each modified version when performing a semantics-based change-merge of three versions of a prototype.

Consider two modified versions of the base Patriot prototype, Version 1.1, shown in Figure 3. The first modified version, 1.2, shown in Figure 5, is one in which a modified version of the control_Patriot module has been installed in a working system. The second modified version, 2.2, shown in Figure 6, is a copy of the base in which the display_status module has been updated. The preserved part of the base version of the prototype is determined by finding the largest slice common to all three versions. In the Patriot system example, this is the slice of Patriot's PDG with respect to the following set of streams: {radar_mode, track_id, missile_track, intercept_angle, target_range, tactical_status, launch_angle, track_file, launch_Patriot_ext}. A picture of this slice is shown in Figure 7.
The affected parts of the two modified versions are computed by comparing the base version, Figure 3, with the modifications, Figures 5 and 6. This comparison is done using prototype slicing. If the slice of a modified version with respect to a set of streams is different than the same slice of the base version, then those streams are affected parts. Any change between the base and the modified versions is significant and must be included in the affected part for that modification. The affected parts of the two modified versions of the Patriot prototype are shown in Figures 8 and 9.
In our algorithm, the change-merged version is built by combining the preserved part with each of the affected parts using a graph-union operation. The result of applying the change-merge operation to the three versions, 1.1, 1.2, and 2.2, of the Patriot prototype is shown in Figure 10. This is a change-merged version of the graph but is not yet guaranteed to be semantically correct. The semantic correctness is guaranteed only after the change-merged version is compared with each of the modified versions to ensure that any changes made in the modified versions are preserved during the change-merge operation.

We check for correctness by comparing the slices of both the modified version and the change-merged version with respect to the streams in the affected part for that modified version. If the slices are the same, then we are guaranteed to have preserved the semantic meaning significant to the modified version. If the slices are different, then a conflict has occurred between the two modified versions that must be resolved by the engineer.
5. SUMMARY

We have described a formal method that can significantly improve both the software development and software maintenance processes. The method provides the ability for software development teams to work on independent parts of a software project and use a computer-aided tool to automatically integrate their respective updates. This method can significantly decrease development time, thereby improving customer satisfaction with the delivered system. This method can also be used to update different production versions of a software system by allowing changes to be made to the base version and automatically change-merged with each of the versions in the field. Use of this method will decrease both software development and maintenance costs by providing the software engineer with a reliable computer-aided tool to decrease their workload.

Future enhancements to this research include considering abstract data types written in the prototyping language, finding ways to resolve more conflicts automatically, and developing methods to change-merge programs written in implementation languages like Ada and C++.
INTENTIONALLY LEFT BLANK.
6. REFERENCES


<table>
<thead>
<tr>
<th>NO. OF COPIES</th>
<th>ORGANIZATION</th>
</tr>
</thead>
</table>
| 2            | ADMINISTRATOR  
ATTN DTIC DDA  
DEFENSE TECHNICAL INFO CTR  
CAMERON STATION  
ALEXANDRIA VA 22304-6145 |
| 1            | DIRECTOR  
ATTN AMSRL OP SD TA  
US ARMY RESEARCH LAB  
2800 POWDER MILL RD  
ADELPHI MD 20783-1145 |
| 3            | DIRECTOR  
ATTN AMSRL OP SD TL  
US ARMY RESEARCH LAB  
2800 POWDER MILL RD  
ADELPHI MD 20783-1145 |
| 1            | DIRECTOR  
ATTN AMSRL OP SD TP  
US ARMY RESEARCH LAB  
2800 POWDER MILL RD  
ADELPHI MD 20783-1145 |
|              | ABERDEEN PROVING GROUND |
| 5            | DIR USARL  
ATTN AMSRL OP AP L (305) |
<table>
<thead>
<tr>
<th>NO. OF COPIES</th>
<th>ORGANIZATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>DIR USARL</td>
</tr>
<tr>
<td></td>
<td>ATTN AMSRL SC IS</td>
</tr>
<tr>
<td></td>
<td>115 OKEEFE BLDG GIT</td>
</tr>
<tr>
<td></td>
<td>ATLANTA GA 30332-0800</td>
</tr>
<tr>
<td>2</td>
<td>DIR USARL</td>
</tr>
<tr>
<td></td>
<td>ATTN AMSRL SC IS</td>
</tr>
<tr>
<td></td>
<td>CPT DAVID A DAMPIER</td>
</tr>
<tr>
<td></td>
<td>155 OKEEFE BLDG GIT</td>
</tr>
<tr>
<td></td>
<td>ATLANTA GA 30332-0800</td>
</tr>
<tr>
<td>2</td>
<td>DIR USARL</td>
</tr>
<tr>
<td></td>
<td>ATTN AMSRL SC IS</td>
</tr>
<tr>
<td></td>
<td>MAJ RONALD B BYRNES</td>
</tr>
<tr>
<td></td>
<td>115 OKEEFE BLDG GIT</td>
</tr>
<tr>
<td></td>
<td>ATLANTA GA 30332-0800</td>
</tr>
<tr>
<td>2</td>
<td>DIR USARL</td>
</tr>
<tr>
<td></td>
<td>ATTN AMSRL SC IS</td>
</tr>
<tr>
<td></td>
<td>LTC MARK KINDL</td>
</tr>
<tr>
<td></td>
<td>115 OKEEFE BLDG GIT</td>
</tr>
<tr>
<td></td>
<td>ATLANTA GA 30332-0800</td>
</tr>
<tr>
<td>1</td>
<td>DIR USARL</td>
</tr>
<tr>
<td></td>
<td>ATTN AMSRL SC I</td>
</tr>
<tr>
<td></td>
<td>115 OKEEFE BLDG GIT</td>
</tr>
<tr>
<td></td>
<td>ATLANTA GA 30332-0800</td>
</tr>
</tbody>
</table>

ABERDEEN PROVING GROUND

1            | DIR USARL    |
|              | ATTN AMSRL SC |
This Laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers to the items/questions below will aid us in our efforts.

1. ARL Report Number _ARL-TR-839_ Date of Report _August 1995_

2. Date Report Received ________________________________

3. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) ________________________________

4. Specifically, how is the report being used? (Information source, design data, procedure, source of ideas, etc.) ________________________________

5. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided, or efficiencies achieved, etc? If so, please elaborate. ________________________________

6. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.) ________________________________

7. If indicating a Change of Address or Address Correction, please provide the Current or Correct address above and the Old or Incorrect address below.

7. CURRENT ADDRESS

   Organization
   ________________________________
   Name
   ________________________________
   Street or P.O. Box No.
   ________________________________
   City, State, Zip Code

7. OLD ADDRESS

   Organization
   ________________________________
   Name
   ________________________________
   Street or P.O. Box No.
   ________________________________
   City, State, Zip Code

( Remove this sheet, fold as indicated, tape closed, and mail.)
(DO NOT STAPLE)