A Code Synthesis Experiment

Henson Graves, Joe Louie, Tracy Mullen

Lockheed Palo Alto Research Labs O/96–10 B/254E

Abstract

The Lockheed Palo Alto Research Labs has an active research program and long term commitment to software synthesis. We have developed a prototype synthesis system, The Lockheed Environment for Automatic Programming (LEAP). The LEAP synthesis approach depends on using software assets that are built during the course of application development. We have used LEAP to develop significant applications in several domains including target tracking, autonomous underwater vehicles, and sensor resource management. In the largest application built so far the synthesized software consists of 65,000 lines of Ada code and 10,000 lines of C code. This paper describes how we used LEAP to construct two simple application examples.

1: Introduction

The Lockheed Palo Alto Research Labs is a participant in the DARPA Domain-Specific Software Architecture (DSSA) program. Our inclusion in this program is based on Lockheed's active research program and long term commitment to software synthesis. Our approach to synthesis depends heavily on using software assets that are built during the course of application development. The assets include architectural descriptions and templates that describe how to generate code for system components. The architectural descriptions and templates can be used to represent domain specific information.

We have built a prototype software synthesis system, the Lockheed Environment for Automatic Programming (LEAP) that employs the asset based approach [Graves 1991]. LEAP allows software production to be incrementally automated and in large part produced by system engineers rather than software engineers. LEAP fits into the Knowledge Based Software Development paradigm outlined in [Green 1986] and to work by [Rich 1988], [Mark 1991] and [Smith 1990]. We have used LEAP to develop significant applications in several domains including target tracking, autonomous underwater vehicles, and sensor resource management. In the largest application built so far the synthesized software consists of 65,000 lines of Ada code and 10,000 lines of C code.

Dr. Marc Graham of Software Engineering Institute (SEI) at Carnegie Mellon University visited the research lab on February 17 and 18, 1992. SEI is under contract to DARPA to be technical coordinators for the DSSA program. Dr. Graham wanted to find out in some detail how well LEAP worked. He presented us with two test problems for LEAP synthesis three days before his visit. The SEI experiment consisted of a description of a phone book management system and a paper that described variants of the "implicit invocation" programming cliche. The challenge was to see if we could capture this descriptive information and have LEAP, using its asset base, synthesize software implementations for the descriptions. We were able to synthesize code for the applications from the graphical descriptions that we constructed using several pre-existing templates. The template assets were usable without change. The templates are parameterized by several data structures and functions. Thus, a small amount of coding was necessary to produce these parameters. The application was working when he arrived and was modified according to his direction during his visit.

2: The LEAP Development Paradigm

To understand how the synthesis experiment worked, we will outline the LEAP application development process. System engineers graphically compose descriptions of their application by specifying components and inter-relationships. For different applications it may be convenient to display information with different graphical conventions and in varying levels of detail and therefore display conventions can be tailored for specific application domains. The engineer may introduce components using existing descriptions or construct the descriptions directly. Some of the pre-existing descriptions have synthesis rules that can be used to generate implementations. We call these descriptions templates. The LEAP application development process is illustrated schematically in Figure 1. In the diagram the shaded components represent pre-existing descriptions. The engineer specializes one of the de-
The engineer uses the graphical editor to compose an application system description by specifying component descriptions and interrelationships using both pre-existing and newly developed descriptions.

**APPLICATION REQUIREMENTS**

"... system will have microwave radar sensor...
... target type is ballistic missile...
... expected target density is medium..."

![Diagram of application requirements](image)

Describing an application figure 1

scriptions by specifying that the sensor type is to be microwave radar. LEAP uses this additional information to compute a more detailed description. When the descriptions are sufficiently detailed LEAP can synthesize an implementation. As the description evolves, the implementation is resynthesized. Thus, consistency between the description and the implementation is automatically maintained. Both the descriptions constructed by the engineer and the synthesized code are deliverables of the development process.

Descriptions are hierarchical in that description components may also have descriptions. The interrelationships and other constraints are expressed as predicates on the components. The predicates can be used to express both high-level computing abstractions such as pipes and client-server relationships, and low-level more directly implementable relationships such as synchronous and asynchronous communication. Generic architectures and the initial design descriptions typically express high-level relationships. Much of the design process is concerned with refining these abstractions into low-level "executable" descriptions. Application development leverage occurs when the asset base contains architectural descriptions that can be used as the starting point for application development and when an application description can be constructed from pre-existing synthesizable component descriptions.

The initial target for synthesis in LEAP is a high level executable design language, called the Common Intermediate Design Language (CIDL) [Graves 1992]. CIDL is a typed language, designed for describing large distributed systems, with higher-order functions, polymorphism, and concurrency constructs. Synthesizing code in a language such as CIDL greatly simplifies the synthesis problem. The synthesis process is not dependent on the specifics of a programming language such as Ada. Compiler technology can be used to translate the intermediate form into Ada or other programming languages. LEAP currently translates CIDL code into Common LISP, Ada, and C.

The synthesis procedure uses a combination of instantiating parameterized CIDL code, constructing new code using synthesis rules, and directly calling foreign code. Since CIDL supports advanced object oriented features, the use of type parameters and subtyping, and polymorphism, much less code construction is needed than if the synthesis target were directly Ada. However, because CIDL has these features not present in conventional languages the translation of CIDL into these languages is non-trivial. The CIDL code is analyzed by the translator to produce an internal form that eliminates the higher order
constructions to produce a form that can be used to generate code in conventional programming languages relatively easily. The Ada code produced by the code generator can be incorporated into larger Ada implementations and can use pre-existing Ada code.

The LEAP-generated Ada executes efficiently. Maintenance is preferably done by re-synthesis which has the advantage that the implementation is consistent with the system description. However, for some projects this process is not an option and therefore we devoted considerable effort to produce readable, maintainable, deliverable quality Ada code.

3: The SEI Experiment

The experiment was to graphically construct system descriptions to represent generic forms of the two examples provided to us, and have LEAP synthesize Ada code using its template asset base. One example was an Event System that represented the "implicit invocation" cliche. The second example was a phone database manager. In both cases we first looked for templates that we could use to describe the components and graphically constructed an architecture to exploit the templates. We briefly describe the applications.

3.1: The Event System

The implicit invocation cliche consists of a collection of agents, e.g., application processes, that use services of each other. Rather than each agent calling methods of other agents directly, a broadcast server is used to maintain a register of the agents and the kinds of events that they are willing to process. The agents send their requests to the server which forwards the request to the appropriate agent. In different variants the register can be static, the registering can be done by clients themselves, or by some special agent. We interpreted the application as consisting of a collection of agents communicating through the broadcast server. Since it did not seem important for the exercise that the agents really did anything interesting, we decided to let them have a user interface that could be used to make requests and display the results of handling the requests.

3.2: The Phone Book Manager

The phone book manager example is a process that manages a phone book database. The database contains three relational tables: Person, Category, Cross-Reference. The behavior of the manager is to accept a last name, retrieve the entries with the last name, display the entries, and give the user options for inserting, deleting, modifying the entries. The manager has a user interface, a database, and control logic to perform the retrieval, display, and update functions. Our implementation did not actually have three tables but implemented equivalent functionality.

3.3: The Challenge

The synthesis of the two application implementations, without manual coding, depended on the existence of LEAP templates that could be used for the application components, and the ability of the graphical description language to describe the interconnections between the components. Specifically we needed templates for describing the agents and the broadcast manager in the event system application, and the database, control structure, and user interface in the phone book example.

Our template library has been constructed on an ad hoc basis. Thus, there was no particular reason to suppose that we had templates that could be used for the examples. However, the examples are specializations of commonly occurring situations. Thus, we happened to have templates that were usable for these examples. We had a database server template that we could use for both the broadcast server and the database manager. We used a template for a process whose behavior is described via a state transition diagram for both the agents of the event system and the control in the database example. The templates were not a perfect match, but they were sufficient to produce a good approximation of what was wanted.

4: The Templates

In this section we describe the templates that we used in the two examples. The particular templates are called DBS, AUTOMATA, and UIS.

4.1: The Database Server

The DBS template describes the type of a process which provides the services of a monitored database to its clients. A monitored database is a data store with a set of functions for performing retrieval and update operations on the data store, and for notification when the contents of specified data locations are updated.

A monitored database (Monitored_DB) may be described as an instance of a parameterized abstract data type. The parameters are a primary key type (P_Item), a secondary key type (S_Item) and the type of data to be stored (Data). While CIDL abstract data types can include axioms, since the current technology to automatically verify a formal axiomatic specification is not practical in large real-world systems, it is not used except for "special cases". However, should the technology improve in power and speed in the future, CIDL has the capability to support it. The CIDL syntax for this is:
Event Manager System Architecture

Figure 2

Monitored_DB(P_Key:Type,
S_Key:Type,
Data : Type): Type =

Theory
update : map P_key, Data to Action
retrieve:map P_Key,S_Key to seq(Data)
remove : map P_Key to Action
set_trigger : map P_Key, S_Key,
Trigger_Type(Data) to Action
unset_trigger : map P_Key, S_Key,
Trigger_Type(Data) to Action
unset_triggers : map P_Key to action

A monitored database instance provides the actual operations. Informally, the set-trigger operation takes as arguments an event and a data location. When the value of the data location changes (via the update function), then the specified event will be notified of the new value.

If a DBS component, which acts as a server, is linked to other components via the DB_ACCESS relation, the client is provided with a functional interface for the monitored database of the server. In other words, a DBS component acts as a kind of remote procedure call facility for the monitored database. The DBS's functional interface is a set of functions for performing retrieval and update operations, and for event notification when the contents of specified data locations are updated. Synthesis rules ensure that these interface functions are available inside any component that is linked to the DBS with the DB_ACCESS relation. So the meaning of the DB_ACCESS relation is that it allows one component to call the interface functions of a DBS.

Figure 2 displays a schematic example of a DBS that is connected to three clients. The boxes are separate components where each one is labeled with the name of the component and its type. For example, Event_Manager : DBS means that Event_Manager is a component which is made
from a DBS template. Note that in figure 2, the DB_ACCESS relations are called Event_Manager_Access. Generally for relations, we don't display the type information since it is implied by graphical attributes of the relation lines such as thickness, dashes or color. If we did include this information, you would see the relations between the agents and the DBS labelled "Event_Manager_Access : DB_ACCESS" instead of just "Event_Manager_Access".

The Agent component template in figure 2 is built from two subcomponents, one a user interface component (UIS_Interface : POKE_M) and one a finite state automata component (CNTRL : EVENT_CONTROLLER). The UIS_Interface component is made from another template, POKE_M, similar to DBS which will be discussed in section 4.3. We were able to use this component without any modification. The EVENT_CONTROLLER template, used for the CNTRL component, was built up using finite state automata, which can be automatically synthesized as described in section 4.2. The triangles at the lower part of the diagram are the parameters and internal data stores of the Event Manager System. Their values can be changed by choosing the "Edit" menu option. For example, the server_port may be changed to a different number by editing the value of the triangle labelled "SERVER_PORT".

4.2: Automata

The automata template defines a class of processes whose behavior is specified as a finite state automaton. The minimum required components are:

- one or more out_ports,
- one or more in_ports,
- a start state and another state,
- transition relations between the states.

Figures 4 is an example of an automata template. The states in the transition diagram are represented by circles and the transitions are represented by connecting lines with arrows going from state to state. The communication in-ports and out-ports are represented by circles located at the left and right side of the diagram, respectively. Not shown in the diagram are the condition and actions associated with each transition. In the diagrams they can be entered and edited by selecting a transition and opening it.

A transition is a relation whose value is a sequence of expressions that specifies a test and zero or more actions. The first expression is the test and the rest are actions. The syntax can be seen in the following example:

```
[ GET_TIME( ) - STARTTIME > RUN_DURATION;
  SEND(FALSE, A1);
  SEND(FALSE, A2) ]
```

Any in_port name referenced in the test condition stands for the data read from the in_port. The test condition associated with a transition is satisfied if and only if all the data are available to be read from the in_ports referenced and the test expression is evaluated to be true. Any in_port referenced in the actions will then refer to the data which has been read off the in_port.

An automata process starts in its start state and transits through its states until it reaches an end state. At any state other than an end state, each transition leading from the state is tested. A transition whose associated test condition is satisfied is selected, any specified actions are performed and the automation advances to the next state to repeat the state transition cycle again.

The AUTOMATA template is somewhat different from the DBS or POKE_M templates, in that it will be applied to any modules containing states and transitions in their behavior.

4.3: The User Interface System

The UIS template describes a process that provides a simple, X-window based user interface. In the diagrams in this paper, it is called POKE_M. The user can enter in string messages, and press the send button. The user interface displays "SENT: message" and the user message string is then sent to whatever module is connected to its out_port. Any strings arriving on the POKE_M in_port are also displayed to the user. The figure below is the user interface for agents in the Event Manager example.

```
UIS Display Window
Figure 3
```

The communication from the C-portion of the code (running the X-window display) to the CIDL-portion (handling the communication with other CIDL processes) is via an ipc server-client model. A server runs in the background, and each POKE_M template connects to it using the values of its required parameters SERVER_HOST, SERVER_PORT, etc.

5: The Event Management System

For the event system we used DBS as the broadcast server. The server maintains a register database of the events
to be notified when a notification request is made. To represent a generic example of this we graphically constructed an architecture diagram that included the broadcast manager and several agents. A schematic version of this is displayed in Figure 2. The Event Manager component is made from a DBS template, the UIS Interface component is made from a POKE_M template and the CNTRL component is of EVENT CONTROLLER type, which is made from finite state automata.

In our initial variant of the event manager system, we decided to have multiple identical agents. Each agent would have its own user interface where the user could do one of four things: register (declare) a "legal" event, request or unrequest to receive the event, or signal that the event had occurred. It turned out that the DBS's interface provided all the functionality needed to implement this. Table 1 shows how the event manager system commands translated into the DBS ones.

Whenever the control component receives a user message on UIS_IN, it checks to see if it is one of the known commands (register, request, unrequest or signal). If it is, the command is executed. If not, a message is sent back to the user saying that an unknown command has been received. Whenever the control component receives a notification on SIGNAL_EVENT, it sends a message to the user that a "SIGNAL" was received and the value provided by the signal (i.e. "event-1 occurred").

The transitions for the AUTOMATA which accomplished this were written at the CIDL level. Because of current limitations of the graphical development environment (GDE), we needed to write three extra functions for the transition relations: assign, make-trigger and get-tokens. Assign is equivalent to the assignment operator, Make-trigger creates a complex data type containing a reference to the SIGNAL_EVENT and Get_tokens tokenizes a string, returning a sequence of string tokens. Thus "DECLARE even-name" will become ["DECLARE" even-name]. The other functions were all available from the DBS interface or are CIDL primitives (send, first, second).

5.2: Describing the Agent

Once these primitive components were made, the system could be constructed from them. The first step was to make an agent template. This was straightforward, simply build a composite module from an AUTOMATA component and a POKE_M component and attach their imports and exports appropriately: the out_port of the POKE_M module to the in_port of the control module and vice versa. We used an asynchronous communication relation for this whose data type was string. Eventually the data type would be inferred from its connections but currently it is necessary to specify it. Parameters of any subcomponent that would need to be filled out when a new agent module was created were made into parameters for the agent. This way the values can be set just once at the top level and will be inherited automatically by the subcomponent.
5.3: Building the Event Manager System

Once the agent template was made, all the necessary components were available and the entire system could be built. We initially built the system with three agents and then demonstrated adding another agent. Each agent was connected to the DBS in turn with a DB_ACCESS relation, and its parameters, such as WINDOW_TITLE, SERVER_HOST, SERVER_PORT were filled in. The system description was fully synthesizable and executable at this point.

5.4: Variations

Simple variations, such as changing the command names, can be done one of two ways depending on what is needed. For example, if we wanted to rename the REGISTER command to the DECLARE command in one of the agents, we would just change the test condition for the transition from Read-UIS-Cmd to Register-Event-Cmd (see Table 2) in that agent's control component. Instead of CMD = "REGISTER" it would be CMD = "DECLARE". However, if we wanted to have this change for all the agents (and didn't want to change them one by one) we would need to modify the agent template. The first step would be to make a new variation of EVENT_CONTROLLER, with the transition test condition changed. The second step would be to replace the old EVENT_CONTROLLER component in the agent template with the new one, say NEW_EVENT_CONTROLLER. Once this was done, then all the existing agents would have the new control component instead of the old one (i.e. CNTRL : NEW_EVENT_CONTROLLER).
<table>
<thead>
<tr>
<th>FROM</th>
<th>TO</th>
<th>CONDITION</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Start-Loop</td>
<td>True</td>
<td>assign(event-trigger, make_trigger(signal_event))</td>
</tr>
<tr>
<td>Start-Loop</td>
<td>Receive-Event</td>
<td>SIGNAL_EVENT</td>
<td>send(&quot;SIGNAL&quot;, UIS_IN); send(SIGNAL_EVENT, UIS_OUT)</td>
</tr>
<tr>
<td>Receive-Event</td>
<td>End-Loop</td>
<td>True</td>
<td>assign(uis_tokens, get_tokens(UIS_IN)); assign(cmd, first(uis_tokens)); assign(args, rest(uis_tokens)) unset_triggers(first(args)); update(first(args), &quot;default-value&quot;); set_trigger(first(args), &quot;,&quot;,&quot;event-trigger); update(first(args), &quot;,&quot;,second(args)); send(&quot;Unknown cmd&quot;,UIS_OUT); send(cmd, UIS_OUT)</td>
</tr>
<tr>
<td>Start-Loop</td>
<td>Read-UIS-Cmd</td>
<td>UIS_IN</td>
<td>make_trigger(signa1Levent))</td>
</tr>
<tr>
<td>Read-UIS-Cmd</td>
<td>Register-Event-Cmd</td>
<td>cmd = &quot;REGISTER&quot;</td>
<td>make_trigger(signa1Levent)</td>
</tr>
<tr>
<td>Read-UIS-Cmd</td>
<td>Request-Event-Cmd</td>
<td>cmd = &quot;REQUEST&quot;</td>
<td>make_trigger(signa1Levent)</td>
</tr>
<tr>
<td>Read-UIS-Cmd</td>
<td>Unrequest-Event-Cmd</td>
<td>cmd = &quot;UNREQUEST&quot;</td>
<td>make_trigger(signa1Levent)</td>
</tr>
<tr>
<td>Read-UIS-Cmd</td>
<td>Unknown-Event-Cmd</td>
<td>not cmd=&quot;REGISTER&quot; and not cmd = ...</td>
<td>make_trigger(signa1Levent)</td>
</tr>
<tr>
<td>*-Event-Cmd</td>
<td>End-Loop</td>
<td>True</td>
<td>make_trigger(signa1Levent)</td>
</tr>
<tr>
<td>End-Loop</td>
<td>Start-Loop</td>
<td>True</td>
<td>make_trigger(signa1Levent)</td>
</tr>
</tbody>
</table>

Transitions for The Agent's Control AUTOMATA Component

Table 2

A second variation we did, upon the request of Dr. Graham, was to have a single distinguished agent that handled the event registration. The other agents could only signal events. We did this by creating two different agent templates with two different types of control automata components (i.e. EVENT_CONTROLLER1, EVENT_CONTROLLER2).

6: The Phone Book Manager

The primary element of the Phone Book Manager is the database. We used the DBS template for the phone book database. The application needed a user interface and an executive. We used the generic POKE_M template for the user interface, and automata for the executive.

6.1: The main program

The phone book manager accesses the phone book database in response to queries it received from users. This overall structure of the main program is reflected in the graphical system description shown in Figure 5.

In this diagram, the CONTROL component is the phone book database manager, the DBS is a database server which contains a database component, and the UIS is the user interface. The database manager is connected to the database server via DBS_INTERFACE, an instance of the DB_ACCESS relation. This enables the database manager to make function calls to access the database. At the front end, the database manager is connected to the UIS for inputs and outputs via communication events C1 and C2.

6.2: The Manager

The phone book manager accepts a last name from the user, retrieves all the personal records with that last name and displays them to the user. The user can then either modify one of the existing records or insert a new one. These operations are reflected by the graphical description of the CONTROL unit shown in Figure 6. Although the whole CONTROL module could have been created as one large automata, a certain amount of modularization made it easier to build and understand. Currently we do not have hierarchical AUTOMATA templates, but we created the same functionality by having separate processes PROCESS_LAST_NAME, INSERT and MODIFY. Each of these processes is built from automata.

6.3: The control flow of the manager

The logical details involved in processing user input are specified in the PROCESS_LAST_NAME description (Figure 7). This description contains a state transition diagram where the states are visually represented as small squares instead of circles. The start state ST is distinguished by having its name labeled above the square box.
The end state ND has its name labeled below the box. The communication inports and outports are represented by circles located at the left and right side of the diagram, respectively. The triangles at the lower part of the diagram are the parameters and internal data stores of the automaton.

The automaton starts at state ST. The first transition (from ST to ST1) prompts the user for a last name, and waits for user input. If the user decides to quit the session (by typing "END" or carriage return) then the automaton terminates (state ND). If the user enters a last name, the personal records having that last name are retrieved (state SN). Depending on the number of person records with that last name, there are three different things that PROCESS-LAST-NAME can do at this point. If the number of records returned is zero, then the INSERT component will be called (state SO) and PROCESS-LAST-NAME will wait for INSERT to finish (state DN). If the number of records is one, then the user is given the choice of modifying the current record, deleting it, or inserting a new one (state S1). If the number of records is more than one, then the first names are presented to the user (state SM). The user can choose one of the existing records (state S1) or decide to insert a new one (state SO).

If the user has chosen one particular personal record (state PER), then it can either be deleted, modified, or a new record can be inserted. For deletion, the DBS remove function is called, and the session is done (state LP). For insertion, the INSERT component is called (state S0). For modification, the MODIFY component is called. For either INSERT or MODIFY, PROCESS-LAST-NAME waits for a signal on the DONE in_port, signifying completion (state DN). Once MODIFY or INSERT has sent a DONE signal, the user session is finished (state LP). A new session is automatically initiated after completing the previous one (True test condition on transition from LP to ST).

An example transition rule is given below. This rule stated that a transition from state SN to SM may occur if the number of person records is greater than one. Before the transition takes place, actions are performed which in this case are the formatting and display of information and prompt messages to the user.

\[
\text{sn_sm: transition(SN, SM) = }
\begin{cases}
    \{(\text{length(persons)} > 1)\};
    \text{send(format_string("First names}
\end{cases}
\]

...
7: Discussion

We believe that this experiment demonstrates the viability of the LEAP synthesis approach. Constructing these implementations is not programming in the traditional sense. We are not advocating any special merit to the particular design approach that we used. The approach was determined by the templates that we had available.

Success with such simple examples does not, of course, guarantee that the technology will scale up. However, we have used LEAP to generate the software for medium size applications. The process on the larger examples is not different from the process on the small examples. This is because the large examples can still be approached top-down, incrementally adding fidelity to requirements. Constructing a shallow prototype for a complex system is similar to constructing a complete implementation for a simple system. As LEAP builds up its asset-base, including architectural level descriptions (which include design rules and constraints), the scale-up to handle the complete implementation of a complex system will use the same methodology, just incorporating more higher-level abstract assets.

The use of CIDL as an intermediate level design representation has proven very robust in its expressive capability. Currently, the graphical descriptions are not nearly as
expressive. However, as we recognize new cliches, it is easy to add them as templates that are accessible at the graphical level. Our estimate based on several applications is that we have a productivity gain of 20:1 over the traditional design and coding process. This is based on the assumption that hand coded Ada on large projects is produced at the rate of a line per hour. This rate of code productivity includes the time spent in developing requirements and design specifications, and in activities such as testing and documentation. LEAP cuts down on the manual effort needed for these auxiliary activities. During the course of development we produced a number of reusable component templates and added them to our asset base. We expect to see increasingly dramatic productivity increases as LEAP is used to tackle other related applications.

Even in application domains for which there has not been any investment in knowledge engineering, LEAP provides leverage over traditional programming language environments. Developing an application by graphical layout yields a significant productivity increase over successively developing the requirements, design specifications, and implementations in separate systems. The LEAP system descriptions offer relief from the well known difficulty that users cannot write formal specifications for simple applications, let alone complex ones. Composing a description from component descriptions allows well understood components (templates) to be provided with a complete implementable specification. The user does not have to write these specifications, only modify them to accommodate the specific application needs. During the course of application development templates and design rules can be captured to provide a growing knowledge base. Design rules can be used to constrain the search space used by the synthesis engine in the same way as humans do.
Acknowledgements

We gratefully acknowledge the contributions of Wolf Polak, Alan Teague, and Nick Weyland to the LEAP effort.

References