Knowledge-Based Constraint-Driven Software Synthesis
Tobiah E. Smith and Dorothy E. Setliff
Department of Electrical Engineering
University of Pittsburgh
Pittsburgh PA 15261

Abstract
This paper describes a constraint-driven, real-time software synthesis architecture called RT-Syn. RT-Syn formulates design space constraints for each task in the real-time software system from timing requirements and a behavior description. RT-Syn then uses these constraints within a simulated annealing-like approach, selects an abstract implementation for every data structure and algorithm required to implement the desired behavior, and transforms these selections into executable code. We present experimental results covering the synthesis of two real-time software tasks that meet the desired constraints. These results illustrate the effectiveness of the simulated annealing-like approach in searching the software design space and the high reusability and maintainability provided by the use of synthesis technology.

1 Introduction
The ability to reuse software is of increasing importance to the software community. The associated costs of tailoring software to meet the changing needs of an application are rising at an ever-increasing rate. Current methodologies range from the identification of reusable parts, to modular design. This paper presents a different approach, namely, the use of software synthesis technology to enhance software reusability. Instead of reusing software code, this paper presents an approach that synthesizes software to meet behavior specifications. Thus, this approach reuses the software behavior specifications and then synthesizes software from scratch in accordance with the demands of the application domain.

A variety of knowledge sources are required to synthesize software. Simon [17] argues that the incorporation of domain-specific knowledge is paramount to the success of software synthesis. It is our belief that domain-specific knowledge can be used to derive quantitative execution measures (e.g., time, space utilization, frequency, and precision) and that these quantitative measures are sufficient to both describe the domain and to codify the effects of the domain on software implementations. This belief is manifested in RT-Syn, a software synthesis architecture targeting real-time software. RT-Syn uses these constraints to formulate a predictive cost function that quantifies the 'goodness' of a particular implementation. This cost function is used within a simulated annealing-like approach to choose abstract implementations for each data structure and algorithm making up the software task.

The remainder of this paper is organized as follows. Section 2 reviews some immediate predecessors to this work, characterizes real-time software, and reviews the basic simulated annealing approach. Section 3 describes the architecture of RT-Syn. This section presents our approach to such issues as algorithm analysis, domain representation, measure generation, implementation selection, and code generation. Specific synthesis examples provide a framework for this discussion and illustrate salient points. Specific attention is paid to the use of quantitative measures of time and space utilization in the synthesis process. Finally, Section 4 presents an overview of our progress and a plan for future work.

2 Background
This section reviews several predecessors to this work, presents a brief characterization of the real-time software domain, and briefly describes the basic simulated annealing approach.

2.1 Software Synthesis
Software synthesis seeks to generate code from abstract behavioral descriptions. Work on software reusability encompasses software synthesis by reusing behavioral descriptions from one domain to another domain. A complete survey of software synthesis is beyond the scope of this paper. See [8, 10, 11, 18] for a survey of recent automatic software synthesis papers.

The direct predecessor of this work is ELF [14, 15, 16], an automatic program synthesis prototype that integrates domain-specific knowledge of VLSI CAD routers with a variety of synthesis techniques such as algorithm selection, data structure selection, and transformation. ELF divides the synthesis process into three distinct stages. Each stage is successively less abstract and has distinct synthesis goals. The first stage focuses on system-level concerns by partitioning the problem into specific chores based on the application requirements. Templates describing partitioning possibilities capture system-level domain-specific knowledge while simultaneously providing structure and strategy to the problem solving process. The output of the first stage is a fully-partitioned problem, meaning all system-level domain-specific knowl-
edge has been applied. The second stage focuses first on inter-chore synthesis requirements, then on intra-chore synthesis requirements. An example of inter-chore synthesis requirements are data structures and algorithms used within multiple chores. These represent chore-to-chore communication requirements. An example of intra-chore synthesis requirements are data structures used within the implementation of a single chore. The design space at this point is very large. Cost evaluations guide synthesis by pruning the large design space. Some sample cost evaluations are best and worst anticipated timing requirements, as well as best and worst anticipated space utilization requirements. Each data structure and algorithm (both statically and dynamically via their interaction) contributes to these cost evaluations. For example, an array implementation for a data structure provides fast and orderly access within an algorithm, but consumes a lot of space which may be a problem for sparse data structures. The outputs of this stage are implementation selections for each algorithm and data structure. The third, and final, stage transforms the implementation selections from the second stage to executable C code. This process is based on a transformational approach, in which a sequence of stepwise refinements ultimately leads to code. ELF successfully generated many different routers, ranging from 1000 to 2500 lines of code, within two hours on a VAX 8800. Each router successfully performed its application and compared favorably with industrial routers.

Other systems also use domain-specific knowledge to synthesize code. Some examples are: Psi/Syn [1], Libra [6], LEAP [4], and Sinapse [9]. Psi/Syn is a transformation-based system that incrementally evolves a high-level specification into executable code. Libra chooses data structure representations by first identifying all plausible options, and then searching for some 'best' implementation option space. Efficiency analysis guides this search. The Lockheed Environment for Automatic Programming (LEAP) is a software development environment that specifically incorporates the use of domain knowledge to interactively design and synthesize Ada and Lisp code. Knowledge is represented within component design templates. LEAP synthesizes code from graphical high level specifications. Sinapse focuses on the interaction between domain-specific representation and application. The Sinapse program synthesis system automatically synthesizes large Fortran mathematical modeling programs. Sinapse has built a variety of real-world application software programs in the domain of oilfield analysis and seismic wave propagation. Sinapse represents a real-world software synthesis application.

2.2 Domain: Real-Time Software

Software for real-time applications can be characterized as a set of time-constrained tasks. Failure is defined as when any task misses its hard deadline. Software development and subsequent redevelopments are one of the major bottlenecks of real-time software design and maintenance. Real-time software is a particularly good domain for research on constraint-driven synthesis for the following reasons: First, timing requirements provide constraints for constraint-driven software synthesis. Second, real-time software is technology dependent. Changes in the underlying hardware platform directly affect the timing and space characteristics. Any change of any task's timing and space characteristics may cause the entire real-time software system to miss a deadline, and thus cause failure. Third, real-time software is typically stable and straightforward, ensuring that the research can focus entirely on constraint-driven synthesis, rather than algorithm design.

2.3 Simulated Annealing

Annealing seeks to remove defects from a solid by first heating the solid to a high temperature, then slowly cooling the solid until the material 'freezes' into a crystal. This 'frozen' material represents the minimum energy state for the solid material. Simulated annealing [12] is a technique for solving combinatorial optimization problems. The motivation for simulated annealing is rooted in the physical properties inherent in annealing solid mechanics [8]. Simulated annealing is not an algorithm, rather, it is more of a methodology or protocol for solving a problem. Simulated annealing is an iterative improvement-type approach, but it has the nice property of not getting 'stuck' in local optima. For the purposes of this paper, it is only necessary to understand several characteristics of simulated annealing. First, there must exist a model of feasible configurations. Second, simulated annealing is an iterative-improvement approach. Perturbations are randomly proposed to the current feasible configuration. A perturbation is termed a move. All allowable moves are termed a move set. A move set must be able to reach all feasible solutions. Third, there must be a definable cost (or objective) function. This function measures the 'goodness' of a solution and is typically quite complex. Finally, there must be a cooling schedule which mimics the physical cooling process. At high temperatures, the material is at a high energy state, thus most moves (even if they result in a worse, and thus less ordered, solution) are accepted. As the material cools and becomes more ordered, it becomes more and more difficult to accept moves that result in a worse solution. All moves that result in a decrease in the cost function at any temperature are always accepted. Moves that cause an increase in the cost function are conditionally accepted based on a decreasing thermal equation. This equation decreases slowly, thus mimicking the cool-down process of physical annealing. The end result (if the system is cooled-down slowly and the cost function is accurate), though computationally expensive, is a near-optimal solution to the problem.

This section reviewed several immediate predecessors to this work and discussed the basic simulated annealing approach. The next section introduces the RT-Syn architecture, an automatic synthesis architecture that uses a simulated annealing-like approach to search the implementation design space.

3 RT-Syn Synthesis System

This section describes the RT-Syn system, a synthesis tool for generating real-time software. RT-Syn
transforms a behavioral specification into a C program that can then be compiled and executed. The execution characteristics meet the user-specified timing and space utilization criteria. This section first summarizes the overall architecture of the RT-Syn system, then details the operation of its various knowledge bases, analysis tools, and synthesis tools. The use of a simulated annealing-like approach to guide the design space search is explored. A set of two detailed synthesis examples (both signal processing algorithms) illustrates issues in implementation selection, and code generation. The synthesized code is shown to meet the timing and space utilization requirements.

3.1 RT-Syn Architecture

RT-Syn synthesizes a real-time software task to meet a behavior specification. A behavior specification has three sections. The first section details the type of processing to be accomplished. Within the real-time software domain, such processing types may be signal processing algorithms, with behavior classes such as Fourier processing, filters, or signal compression. The second section details the required inputs and outputs of the real-time software task. Example inputs and outputs are: integers, biased integers, real numbers, characters, as well as sets of these types. The third section details the required task periodic cycle time. It is this time that acts as the upper bound on single task synthesis.

The following is a broad overview of the synthesis steps used in the RT-Syn synthesis architecture. First, a visual graphical user interface captures application algorithms without implementation specifications. Second, RT-Syn analyzes the task-level data and control flows to produce more exact timing and space predictions. Third, RT-Syn uses these predictions within a simulated annealing approach to select abstract representations of data structures and algorithm implementations to meet required timing and space constraints. Fourth, RT-Syn synthesizes C code from the selected implementations.

3.2 Algorithm Analysis: Garth

Garth takes as input the behavior processing class, finds all algorithms that implement the behavior, quantifies implementation ranges for each algorithm, and then uses a greedy approach to select a particular
algorithm for further synthesis. Interestingly, Garth uses synthesis operations (i.e., the Yardstick and Optimus tools) found in lower layers to formulate implementation ranges.

Garth initially analyzes all algorithmic possibilities and produces a control and data flow graph for each. The data flow graph for the DFT algorithm processing type for the SP1 experiment is displayed in Figure 2. Gross implementation ranges are formulated by noting that timing limitations generally adversely affect space limitations (less time requires greater space requirements). To discover the minimum speed characteristic for any algorithm, Garth synthesizes the algorithm (without actually producing code) while attempting to reach a ‘time = 0’ constraint. Of course, reaching a ‘time = 0’ constraint is impossible so synthesis will fail, but the resultant implementation specification will have the minimum time specification. Garth assumes that this specification will also produce the maximum memory specification. Similarly, setting a ‘space = 0’ constraint and synthesizing (again, without producing code) to meet that constraint find the implementation that uses the least memory and maximum time specifications.

In effect, Garth runs Yardstick and Optimus twice (without allowing Coda to produce the actual code) for each algorithm under consideration to formulate the best timing implementation, the worst timing implementation, the best space utilization, and the worst space utilization measures. These measures are the upper and lower bounds on all possible implementations of that algorithm. Garth then uses a greedy approach in selecting the algorithm whose upper bounds closest to, but not under, the user’s input requirements. This algorithm (in the very high level graphic specification format) is given to the task level synthesis tools (Yardstick and Optimus) for further consideration.

The upper and lower bounds are returned to the Algorithm knowledge base (information on the organization of the Algorithm knowledge based may be found in [20]) to be stored in the resource constraint ranges characteristic for that algorithm. The formulation of any algorithm’s implementation boundaries need only be performed once for a given algorithm and platform combination. Future synthesis of that algorithm need only retrieve the information directly from the Algorithm knowledge base. The greedy approach is not guaranteed to choose an algorithm that will meet the requirements, thus, a backtracking capability must be included. Backtracking occurs when an algorithm fails to meet the initial design requirements and another algorithm must be selected. An algorithm will fail to meet the initial design requirements if the design space for that algorithm is not convex and there exist points in the design space that are not reachable. Garth discards the original algorithm, continues the greedy paradigm in choosing the second closest algorithm to the user’s input requirements. This process continues until the set of algorithms with the desired behavior is exhausted or task level synthesis is successful.

\[ \text{Figure 2: Garth Data Flow Graph} \]

In Figure 2, the data flow graph is shown for the DFT algorithm. The graph illustrates the flow of data and control between different operations such as loop, cos, sin, and sum. Each node represents an operation, and the edges represent the data flow between them.

\[ \text{Figure 3: SP1 Design Space} \]

Figures 3 and 4 illustrate the design space of the two examples, SP1 and SP2. The endpoints of the design space are derived by the best/worst method detailed earlier. These design-space bounds are used...
Shown in Figures 3 and 4, the implementation that is time intensive (and most compact in space) provides the system with an upper time range of 15,000,000 cycles for the algorithm. Likewise, the algorithm most space intensive provides the system with an upper bound on the space axis (in this case, 5 kBytes). In addition, Figures 3 and 4 depict several points which correspond to actual implementation points. The major synthesis points lie along a line referred to as the optimal implementations boundary. All implementation specifications that lie above the implementation boundary (e.g., FFT4, FFT5, and FFT6) will be realized as one of the implementations lying along the boundary. This greatly simplifies the size of the design space. We make this simplification because of a simple fact. It is not necessary nor desirable to synthesize an implementation that with a simple modification will produce a virtually identical timing specification but with a much lower space specification (e.g., TSM5 in Figure 4). Figure 3 depicts the three major synthesis points lying along the implementation boundary of SP1. Figure 4 shows a similar situation for SP2 (though many more implementation options exist in this case). In addition, several points (e.g., TSM4, TSM5, and TSM6) are depicted above the optimal implementations boundary, showing that non-strict characteristics requirements will always result in one of the boundary solutions. This implementation boundary is roughly formulated by Garth when determining which implementation is most likely to suit the user's needs. The data for this formulation comes from the coarse-grained algorithm characteristics found in the Algorithm knowledge base. For example, if a user requires an implementation of SP1 with time < 10 and space < 3, Garth will immediately choose FFT3 as the most likely solution, and attempt synthesis based on that specification first.

3.3 Platform Database

As we noted earlier, the second layer, the task synthesis layer, is composed of a Platform knowledge base, a measure-generation tool called Yardstick, and a constraint-driven implementation selection tool called Optimus. This discussion describes the contents of the Platform knowledge base and the actual platform used in the experimentation, the DLX processor [5].

Synthesis of an implementation requires access to specific platform-dependent knowledge. The Platform knowledge base contains modeling information needed to generate accurate predictions about the timing and space utilization characteristics of tasks for a given platform. This knowledge base captures information on the effects of optimizing compilers (if any), special hardware capabilities, etc. Much research is being performed in the computer architecture predictability and reliability field on the quantification and use of just such this information [2, 22]. The RT-Syn knowledge base represents this information in terms of knowledge about primitives. Primitives are the lowest-level building blocks used by RT-Syn when synthesizing code. Typically, primitives represent single operations, such as addition, multiplication, and branching operations. In these experiments, RT-Syn contains knowledge about primitives on the C compiler for the DLX processor platform [5]. Information stored in the platform knowledge base includes: the timing and space characteristics of all primitives on this platform, formula to account of eccentricities in the behavior of the model of the platform, characterization of any operating system calls that will be used, and a characterization of the C compiler to be used when generating executable code. The behavior of platforms and compilers varies widely. As a basic example, some computers utilize a separate floating-point math unit, which makes floating-point math a faster alternative to integer math. At a compiler level, different compilers perform different optimization techniques, so that the same piece of C code compiled on two different compilers and then run on the same platform will have very different performance characteristics. An important feature of the RT-Syn system is that a given set of tasks can be completely re-synthesized for different platforms by changing only the platform selection. Current work in predictable platforms [21] seeks to design and develop real-world platforms which exhibit the similar predictable characteristics exhibited by the DLX processor.

3.4 Measure Generation: Yardstick

The section describes the mathematical equation required to generate the software metric characteristics within Yardstick. Yardstick is primarily a set of functions describing execution time and space utilization given descriptive information found in the Platform knowledge base. Yardstick performs the 'cost function' capability as required in the simulated annealing approach.

Yardstick takes as input the current data structure that is the focus of design, a proposed selection for that data structure, and all previous design selections.
These previous design selections represent the current state of the design. Yardstick returns the cost of a particular selection (called W), the cost has three different factors: execution time of primitives using that data structure implementation selection (called T), memory requirement of the data structure implementation on the chosen platform (called S), and precision (called P) which measures the ability of the data structure to represent the information through all uses in the data flow graph. Currently, time and space are considered equal in importance to the design. Since the values for time and space differ by orders of magnitude, the weighting of the time and space characteristics is first normalized by calculating the worst-case time/space characteristics for the program to be synthesized (as described in the previous section) and using these resulting values to compute a normalization factor \( \Omega \). This factor is then applied to all space weights, so the true features of a primitive are \( T \) and \( S \times \Omega \). Precision is presently unweighted.

If we adopt the following notation, the Yardstick cost function may be expressed as:

\[
W(A) = (\Omega \times S(A)) + \Gamma(A) + \sum_{\text{all primitives } Y \text{ that use } A} T(Y, A)
\]

Yardstick accesses the Platform database to compute \( \Gamma(A) \) and \( T(Y, A) \). The cost of a data structure is dependent upon the operational platform as well as upon its use within an algorithm. This function captures these characteristics. It is necessary to have a predictable platform for these equations to be useful. As work progresses in the design of predictable systems, this restriction will assume less and less importance.

### 3.5 Simulated Annealing: Optimus

This section describes Optimus by first describing the approach used to order the selection process and then characterizing the annealing process to select individual implementations. This section concludes with a description of how this tool successfully selects a design for the SP1 and SP2 experiments.

Optimus uses the data and control flow graphs build by Garth and the cost functions from Yardstick to select data structure implementations. The selection process is ordered by a bottom-up (or correspondingly output-back-to-input) traversal of the data flow graph from Garth. Design first focuses on the data structures required to process the outputs, then back through all computations to those data structures required to process the inputs to the algorithm. This graph-walking approach specifically acknowledges the impact of external requirements (the required outputs) on the implementation selections. In this way, the external requirements are applied as early as possible and are used to reject portions of the design space that are unworkable. This analysis/selection loop continues until the task implementation is fully specified. At this point, the implementation code is constructed. The actual selection process for each iteration in the analysis/selection loop uses a simulated annealing-like approach to evaluate the costs as derived by Yardstick for specific implementation possibilities in coordination with the current design state.

As detailed in Section 2.3, there are four identifying characteristics of a simulated annealing approach. These are: a configuration space, a set of moves which can reach all feasible solutions, a cost function, and a cooling schedule. The following details these characteristics as exemplified in Optimus.

The configuration space is modeled as sets of current implementation selections for all required data structures along with control flow selections within each algorithm. Each implementation is represented as a hierarchical composition of ‘basic’ data structures. Some examples of basic data structures are: integers, real numbers, characters, records, pointers, arrays, lists, and strings. Control flow selection possibilities and their characteristics are stored in the primitives database. Example characteristics are: data structure suitability and requisite data structure support required to implement the control operation. The move sets are modifications (whether addition, deletion, or actual modification of type) within the hierarchical composition of any data structure. At first glance, the design space is incalculably large. The design space becomes manageable through the use of the primitive-based cost function in Yardstick. Thus, the design space is quickly pruned through an analysis of the actual use of a data structure. Possible implementations (such as an array of integers) that are patently absurd when considering the actual use are never considered.

The final characteristic of the Optimus simulated annealing-like approach is the cooling schedule. The cooling schedule must accept all good moves and conditionally accept bad moves. In addition, the cooling schedule modifies the conditions under which bad moves are accepted until virtually no bad moves can be accepted. At that point, the assignments are considered frozen. The remainder of this discussion reviews specific characteristics of the configuration space that must be included in the cooling schedule.

The Optimus approach differs from typical simulated annealing approaches by the imposition of a limit on the number of times that a particular data structure can be modified. The ‘freezing’ factor applies to the number of changes possible, not the entire state of the design. This is to avoid cycling between implementations due to a cyclical data flow graph. In addition, this allows the annealing process to converge more quickly. The freezing factor is applied to each calculation of the cost function for a particular data structure variable:

\[
W_i(A) = W_i(A) + 0.1i \cdot W_{i-1}(A)
\]

Optimus also slow freezes the entire design as well as individual selections. The entire design en-
...compasses control flow as well as data implementation selection. For example, if a list is chosen rather than an array, there will be a shift in the way the data is traversed from a FOR-NEXT loop to a DO-WHILE loop. Depending on the nature of the program alteration, new variables may need to be introduced or old variables may become obsolete. The first issue to be addressed is when control flow modification are necessary, since such a change can cause major re-evaluations. It is unreasonable to attempt a full re-optimization of the proposed change to determine its merit. The cooling schedule only allows modifications if the resultant change results in a state that is \(< 1.05\) times the old state (the state before the control flow change was made). Initial states have high costs so that small differences are easier to achieve than at cooler states with lower costs. When the control flow of a program changes, all affected primitives and the cost functions for their associated variables are invalidated. This effect can ripple through the data flow graph, causing a large number of variables to be invalidated. Once it has been determined that the control flow should be modified, the number of iterations \(i\) for each cost function calculation for each data structure affected by the control flow modification is reset to zero and allowed to resettle to fit the new design requirements. Thus, after a control flow change, all influenced variables are completely unfrozen. Thus, control flow modification is also a cause of implementation selection cycling. For example, a given data structure (called \(A\)) may originally be implemented as a list. A control flow modification (say to a FOR-NEXT loop) causes a modification to \(A\) to be an array. Further control flow modification may redirect the loop to a DO-WHILE loop, which may further modify \(A\) back to a list. To avoid this type of cycling, the cooling schedule keeps track of the number of resets \(v\) for each variable. As \(v\) increases, it becomes more costly to make the change as dictated by the control flow modification. The cost function then becomes

\[
W_i(A) = W_i + 0.1(i + v) \cdot W_{i-1}(A)
\]

This example illustrates how the design of data flow and control flow interacts, and how the simulated annealing approach controls the volume of modifications implicit in this interaction.

Another result of control flow modification are the introduction of phantom variables. Phantom variables are those data structures required to implement a particular control flow but are not explicit in the data flow representation of the graph. For example, browsing a linked list does not require an explicit index counter, but many algorithms use the index into the data as part of calculations. Optimus determines if this index is needed and will create new variables and code to provide this index when using data structures that have implicit indices. If at a later time this particular linked list is converted to an array, the created variable will become obsolete (the associated code primitives will disappear during conversion). The cost functions for these variables are still tracked even when they are no longer used, since a subsequent iteration might convert that array back to a linked list yet again, in which case the phantom variable is needed. If these variables were disposed of each time, the validation count information would be lost. By tracking them continuously and re-assigning them if it becomes appropriate, Optimus is better able to maintain an accurate representation of the "frozenness" of a particular implementation.

3.6 Selecting An Implementation: Experimentation

This section reviews the operation of Garth, Yardstick, and Optimus for the two experiments, SP1 and SP2.

Execution of RT-Syn consists of determining the implementation range for the algorithm in question, either by actually executing Optimus with extreme conditions or by simply looking up previous values in the Algorithm Database, choosing a likely candidate algorithm, and then attempting to implement that algorithm by making design decisions based on the given constraints. In terms of optimization and implementation selections, SP1 requires a relatively small amount of effort. Control flow issues arise in the decision of how to implement the two simple loops which constitute the majority of the algorithm. Even here, this decision is often implicitly made by the data type (be it a linked list or an array). Therefore, the remaining optimization to be performed takes place at the data-type selection level for the various internal variables used within the algorithm. During annealing, the data type of the main data-holding structure is almost always chosen early in the optimization (though a precise ordering is impossible due to the random nature of the annealing process). To a large degree, this choice alone can determine which of the three implementations boundary (see Figure 3) the algorithm will gravitate towards. Figure 5 shows the predicted and actual time characteristics for several user-specified constraints. Space constraints are not shown for brevity; RT-Syn's space predictions have consistently been very close to 100% of the actual requirements.

<table>
<thead>
<tr>
<th>Name</th>
<th>Synthesis</th>
<th>Predicted Time Characteristics</th>
<th>Actual Time Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT1</td>
<td>0</td>
<td>2.6</td>
<td>2.5</td>
</tr>
<tr>
<td>FFT2</td>
<td>0</td>
<td>14.3</td>
<td>14.0</td>
</tr>
<tr>
<td>FFT3</td>
<td>0</td>
<td>7.0</td>
<td>6.8</td>
</tr>
<tr>
<td>FFT4</td>
<td>8.4</td>
<td>2.9</td>
<td>7.0</td>
</tr>
<tr>
<td>FFT5</td>
<td>5.0</td>
<td>2.6</td>
<td>2.5</td>
</tr>
<tr>
<td>FFT6</td>
<td>15.0</td>
<td>14.3</td>
<td>14.0</td>
</tr>
</tbody>
</table>

Figure 5: Comparison of Predicted vs. Actual Characteristics for SP1

SP2 provides more variety in implementations. However, SP2 introduces data-dependent conditions, which cause the control-flow management equations to take effect. SP2 has several major selection
decisions which can affect the outcome of the synthesis. These are the data types of the input and output data, the looping styles of the two major loops, and the data changes necessary when changing control flow styles. The ordering of the selection process is different each time. To better illustrate the design process, the following briefly describes how a basically random technique effectively focuses on the critical areas of SP2, and then describes in detail the synthesis of a critical loop structure within SP2.

The simulated-annealing-like approach used within RT-Synx has already been described, but it is worth repeating here that potential moves are chosen at random, and then evaluated subject to the new projected speed/space costs and the effects of the cooling functions used with the annealing approach. An important feature of the annealing process is the natural focus on important sections of the algorithm. A significant portion of the design space for a problem the size of SP2 consists of modifications to the myriad primitives that are used only a few times in the course of a program run. These facets of the algorithm do very little to affect the speed/space characteristics of the final algorithm. The remaining portion of the design space consists of the primitives that take part in the major data structures and code routines within the algorithm. Typically, changing the status of one of these important primitives will greatly modify the characteristics of the final algorithm. The annealing process proceeds by selecting a primitive at random and performing a modification to it. The system is then re-evaluated in this new state, and if the result of the evaluation meets the criteria specified above, the move is taken. To recap, bad moves of a certain tolerance are allowed, and primitives become more frozen each time they are moved.

In a typical run, then, random selections will more often than not result in choosing one of the relatively insignificant primitives for modification, since they represent the majority of the primitives in the algorithm. It is likely that the proposed new move for such a primitive will be taken, whether good or bad, since the effect on the algorithm’s final characteristics will be relatively negligible. However, performing a move on one of these primitives will result in that primitive moving into a more frozen state, making it less likely to be moved in the future. When the annealing process happens upon one of the more important primitives, it is likely that only a relatively good move will be accepted, since in this case a blatantly bad move will have a huge impact on the outcome of the run. Following this process through many iterations, the insignificant primitives freeze quickly, following a trend of generally good moves, but with many less-than-optimal selections present, while the key primitives freeze more slowly, and the moves that are performed upon them are almost invariably globally good ones.

Consider the synthesis of a more significant code section, for example, the primitives involved in a loop through one of the algorithm’s major data structures. Assume that the speed and space constraints are both very tight, and that this loop performs an operation for which the index value of the variables is required, for example, a weighted sum of the elements in an array. Initially, the primitive representing the data structure of the values to be used within this loop will be unspecified. The data flow graphs indicate that an array or a linked list of values are possible design choices for this algorithm given the inputs required. In the case of a linked list, Optimus will create an index-tracking variable (a phantom variable) if the index of a value is required within the loop. When the annealing process focuses on this primitive, either selection will initially result in a ‘good’ choice in a single dimension (speed or space), since the linked list structure is more memory-efficient, while the array structure is faster to loop through (only the index variable needs to be incremented, whereas in the linked list case, both the index variable and the link pointer values must be updated each pass through the loop). Because of the design weights on both speed and space Yardstick will initially return similar costs for each option. For this example, assume that the linked list structure is chosen. At this point, Optimus will update the control flow of the loop structure to reflect that it is looping through a linked list. In addition, it will create a new variable to maintain a looping index for this loop, since linked lists have no implicit loop counter by default. As synthesis proceeds, this same primitive will eventually be chosen again. A move to the array state is then proposed and may be taken by the annealing process. Because of the cost ambivalence, the impact of the cooling process is less than would be expected (remember good moves are always accepted). At this point, the previously-created explicit loop counter variable, used in the linked list representation, is no longer needed. A record of its existence is kept, however, since it is possible (indeed, likely) that another type swap will occur for this primitive, so previous decisions about the primitives involved in this loop variable are maintained (this is an example of the phantom variables discussed earlier in the paper). It should be evident that this fluctuating back and forth between data structures could continue indefinitely were it not for the gradual freezing of primitives, making them successively harder to move. Finally, another move back to a linked list representation will almost certainly be made (in practice, the main data structures in SP2 tend to flip-flop about 5 times before deciding on a representation in tight constraint optimizations). When it is, the phantom variable used to maintain the loop index will be reincarnated, along with its previous data type information and state of frozenness. This is done to prevent the information for this primitive from resetting to a virgin state each time the data type of the data structure primitive flip-flops. This example illustrates the effectiveness of the cost function and the impact of the cooling schedule of different parts of the design during the synthesis of a particular experiment. Figure 6 presents data for several synthesis runs of this experiment. The corresponding data points are illustrated in Figure 4.
3.7 Building Code: The Code Generation Layer

The third layer, the code generation layer, is composed of a language knowledge base and Coda, a code generation tool. This layer takes the implementation selections made in the task synthesis layer and generates the actual code. This section briefly discusses how Coda transforms the selections into code, and then describes the salient features of the two experiments, SP1 and SP2.

The final step in the synthesis process is the generation of code to implement the selections made in the task synthesis layer. Code is generated based on declarative knowledge of the target language in the language knowledge base. The inputs to Coda are the set of implementation selections, the control flow graph, and the data flow graph formed in the first layer. The incremental code transformation process begins at the top of the control flow graph. This code transformation and generation process has been fully explored in [1, 6, 13, 14]. The code generation process interacts with the target language knowledge base to generate code to declare, allocate, access and assign values to records, arrays, lists, pointers, integers, characters, real numbers, as well as to complex organizations of these structures (e.g., dynamically allocated array of lists of records). The end result of the Coda tool is code in the target language that implements the desired behavior and exhibits the time and space characteristics specified by the user.

3.8 Building Code: Experimentation

This section reviews the final results of the two experiments, gives an example of code generated showing the design space explored, and reviews the expected characteristics and the actual characteristics attained through the RT-Syn synthesis process. As can be seen in Figures 5 and 6, the time predictions of RT-Syn match closely with the actual values. On average, the system can claim a 98% accuracy rate when modeling unoptimized C code on the DLX processor. A degradation to 95% occurs when optimization is used (using the GNU C compiler) but is not included in the platform database. When a characterization of the optimizations present in the GNU C compiler are added to the platform database, the accuracy returns to 98%. This illustrates the effectiveness of the platform database. It was mentioned above that the space predictions are always quite similar to the actual values, nearing 100%. RT-Syn's modeling techniques are conservative, meaning that they will tend to overestimate rather than underestimate. This feature is vital to a successful constraint-driven synthesis system.

The SP1 algorithm can be synthesized in approximately five seconds on a Macintosh II-class machine. The average number of lines of C code for this example is 39 lines. The SP2 algorithm takes significantly longer, on the order of twenty to thirty seconds, and generates 120-140 lines of C code. The code resulting from the synthesis of SP2 has been incorporated into an existing signal-processing application, where it provides the same functionality as previously handwritten code.

Other experimentation using RT-Syn [19, 20] demonstrates that efficient synthesis (100's of lines of code is less than 30 seconds on a MAC II) is possible and also shows the range of implementations attainable merely by modifying the task-level constraints. These experiments show the power of the simulated annealing-like process in selecting implementations to meet a set of specifications and thereby to provide for software reusability. Only knowledge of platforms and primitives particular to data structure and algorithm selection is incorporated and used. This knowledge prunes the design space, while providing solutions for time and space constrained signal processing tasks.

4 Conclusions

The real-time software development process is time consuming. This paper presents work in progress towards alleviating the costs of real-time software system design, development, and maintenance. The selection process uses a simulated annealing-like approach to search the implementation design space. This synthesis system successfully generates functional C code from high-level algorithmic descriptions. The generated code can be modeled for speed and space requirements, and these predictions prove to be reasonably accurate for a variety of platforms. Results illustrating this ability to accurately predict the time and space characteristics of a task and to synthesize an implementation which meets the predictions can be found in [19, 20]. The ability to generate predictable code is the cornerstone of the RT-Syn system. By demonstrating the validity of the synthesis system, we validate the premise of automated real-time task set synthesis. Future work includes the expansion of RT-Syn into higher abstraction layers (e.g., system-level and communication behavior-level abstraction layers) in order to synthesize multi-task real-time systems and networks, as well as increasing the breadth of the domain by expanding the algorithm or platform knowledge bases.

Acknowledgements

The authors wish to thank the reviewers for their many helpful comments.

Figure 6: Comparison of Predicted vs. Actual Characteristics for SP2
References


