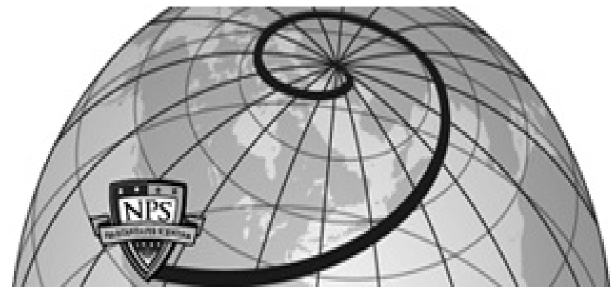
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive

DSpace Repository

Faculty and Researchers | Faculty and Researchers' Publications

1987

Rapid Prototyping of Real-Time Systems

Luqi; Berzins, Valdis

Naval Postgraduate School

Luqi and V. Berzins, Rapid Prototyping of Real-Time Systems, Technical Report NPS 52-87-005, Computer Science Department, Naval Postgraduate School, 1987.
https://hdl.handle.net/10945/65219

Downloaded from NPS Archive: Calhoun

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NPS52-87-005

NAVAL POSTGRADUATE SCHOOL
Monterey, California

RAPID PROTOTYPING OF REAL-TIME SYSTEMS

Valdis Berzins

LuQi

February 1987

Approved for public release; distribution unlimited

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

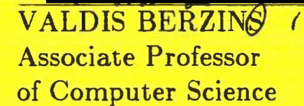
Rear Admiral R. C. Austin
Superintendent

D. A. Schrady
Provost

This report was prepared for the Naval Postgraduate School.

Reproduction of all or part of this report is authorized.

This report was prepared by:

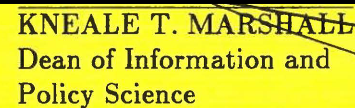


VALDIS BERZINS
Associate Professor
of Computer Science

Reviewed by:

VINCENT Y. LUM
Chairman
Department of Computer Science

Released by:



KNEALE T. MARSHALL
Dean of Information and
Policy Science

Note:

This project was supported by the NPS Foundation Research Program which was funded by the Chief of Naval Research, Arlington, VA 22217.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

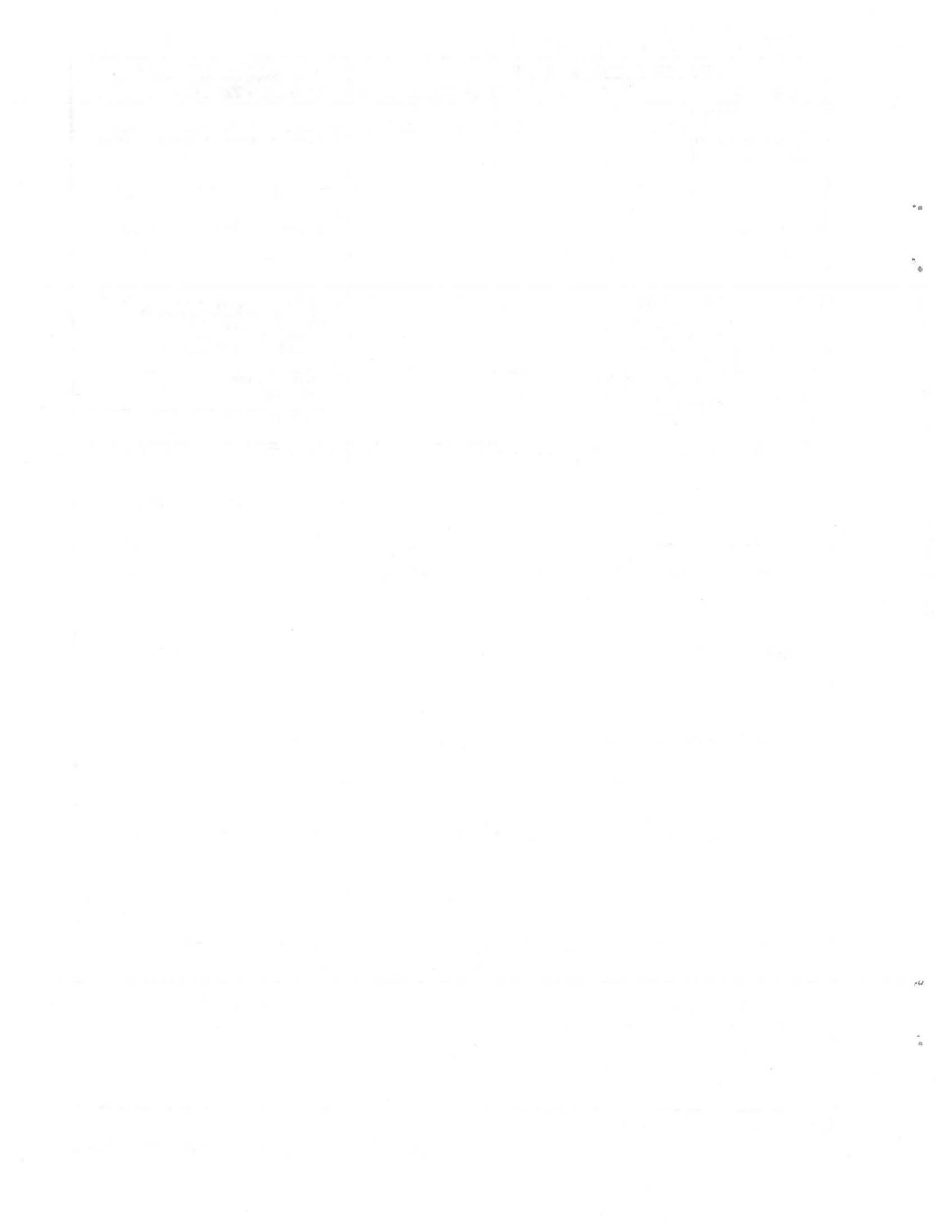
REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NPS52-87-005
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): RAPID PROTOTYPING OF REAL-TIME SYSTEMS
5. TYPE OF REPORT & PERIOD COVERED:
6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s):
Valdis Berzins
LuQi
8. CONTRACT OR GRANT NUMBER(s):
9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Naval Postgraduate School
Monterey, CA 93943-5100
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61153N : RR014-01 N0001487 WR4E011
11. CONTROLLING OFFICE NAME AND ADDRESS: Chief of Naval Research, Arlington, VA 22217
12. REPORT DATE: February 1987
13. NUMBER OF PAGES: 32
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):
15. SECURITY CLASS. (of this report): UNCLASSIFIED
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:
16. DISTRIBUTION STATEMENT (of this Report): Approved for public release; distribution unlimited
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report):
18. SUPPLEMENTARY NOTES:
19. KEY WORDS (Continue on reverse side if necessary and identify by block number):
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The main goals of the prototyping method associated with the PSDL language are to rapidly construct a prototype with a high degree of module independence. The first goal is addressed by an improved modularization technique and a hierarchical approach. The second goal is addressed by an automated environment.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601
UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)
REPRODUCED AT GOVERNMENT EXPENSE



Rapid Prototyping of Real-Time Systems

Luqi

Valdis Berzins

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

The main goals of the prototyping method associated with the PSDL language are to rapidly construct a prototype with a high degree of module independence. The first goal is addressed by an improved modularization technique and a hierarchical approach. The second goal is addressed by an automated environment.

1. Introduction

The demand for large, high quality software systems has increased to the point where a jump in software technology is needed. Rapid prototyping is one of the most promising methods proposed for solving this problem. A prototype is an executable pilot version of the intended system, which is used as an aid for analysis and design rather than for delivery to the user. Rapid prototyping is particularly effective for ensuring that the requirements accurately reflect the real needs of the user, increasing reliability and reducing costly requirements changes. This paper presents a method for rapidly constructing executable prototypes for large real-time systems with the following properties.

(1) The prototype must satisfy and be traceable to its requirements. Iterated prototype construction is used to analyze and firm up the requirements for the intended system.

(2) The prototype must be easy to modify. The prototype will be subject to many revisions before the user is satisfied with the requirements as reflected by the behavior of the prototype.

(3) The prototype must be easy to read and analyze. The prototype serves to support analysis of the intended system and to document an initial design. Clarity and simple high-level structures allow designers to answer questions easily about the properties and feasibility of the system.

Our method was developed together with a Prototype-System Description Language (PSDL) [7] and an automated prototyping environment [5].

The goal in constructing a prototype is different than in constructing a production quality software system. Efficient use of designer time and rapid feedback for the user are more important than robust operation, efficient use of machine resources, or completeness.

We use an integrated approach to prototyping that combines a computational model tailored for describing real-time systems with a high-level prototyping language, a systematic design method for rapid construction of prototypes, and an automated prototyping environment that should enable the effective use of a software base containing reusable components. The computational model prevents hidden interactions between system components and encourages designs with good module independence. The language supports the model, and combines it with a powerful set of data and control abstractions, to make it easy to describe systems at a high level. The automated environment relies on a software base management system for retrieving and adapting reusable software components [10].

A problem-oriented top-down strategy is used to focus the prototyping effort on a critical problem or on selected attributes of the entire system. The major system attributes that must be demonstrated to the user usually appear in a critical subsystem. It is necessary to create a quick sketch of the skeleton of the intended system, because the environment of the critical subsystem must be at least partially simulated to demonstrate the behavior of the prototype and an initial description of the intended system is needed as a basis for discussion. This quick sketch can be built rapidly and understood quickly by means of a interactive graphics editor for PSDL. During the iterations of the prototyping effort, our environment enables each update to the prototype to be made quickly and easily. The prototype system gradually fulfills the requirements in this process, which is discussed in more detail in Section 3.

The PSDL prototyping method results in a hierarchically structured prototype. The method provides a decomposition strategy for filling in more details at any level of the prototype design. It helps to focus on the critical subsystems that must be refined to resolve the problems that motivated the rapid prototyping effort.

The prototype is designed based on abstract functions, abstract data, and abstract control. This high-level view emphasizes the overall configuration at each level without getting embroiled in low-level

details. The design is refined by decomposing abstract functions and data types into lower-level ones in a process of stepwise refinement. Functional, data, and control abstractions are supported by PSDL and are used to hide lower-level details in the prototyping method. Only the concepts directly used at each level appear in the abstractions of the data flow diagram at that level.

1.1. Prototyping Language

We recognize that a good language for expressing design thoughts in terms of a precise model is essential for rapid prototyping. PSDL [7] was designed to serve as an executable prototyping language at a specification or a design level.

The prototyping method produces a PSDL description of the prototype. PSDL provides sufficient structures and descriptive ability to express the internal and external situation for the modules comprising the system. A clear and powerful modularization model (see Section 2.2) is introduced in PSDL for building and describing the prototype. The model is based on dataflow under real-time constraints. The decomposition of a composite operator is described in PSDL by an enhanced data flow diagram that includes non-procedural control constraints and timing constraints. PSDL and its prototyping method are concerned primarily with hard real-time systems. A hard real-time constraint is a bound on the response time of a process or the period between invocations that must be satisfied under all operating conditions. A hard real-time system [8] has hard real-time constraints as part of its requirements.

1.2. Previous Work

Two kinds of software system decompositions have been identified [4], one based on dataflow and the other based on control flow. There is no previous work providing a syntactic and semantic means of combining dataflow and control flow for software system design.

A number of non-procedural programming languages have been proposed in recent years. These languages have the advantage of being easy to analyze, and of exposing the natural parallelism in an algorithm. The design of the non-procedural control constraints of PSDL owes much to these ideas. One difference between our work and previous approaches to rapid prototyping using applicative languages is

that we provide a black-box specification for each component in addition to an implementation. Black-box specifications state which properties of the prototype are required in the intended system and can also be used for retrieving reusable components from a software base.

Our approach to execution support for PSDL is based on previous work on scheduling tasks with real-time constraints [8], which handles asynchronous tasks in terms of equivalent synchronous structures. The application of these ideas to PSDL is described in [6].

There has been a fair amount of work on machine-aided rapid prototyping for systems without hard real-time constraints. A system for prototyping user interfaces for interactive systems is described in [9]. Petri nets are used in [2] to prototype the synchronization and interprocess communication aspects of process control systems. While the notation is not very easy to read, it does support automated deadlock detection and performance evaluation in terms of steady-state probabilities for graph markings. A technique for modeling real world systems which is appropriate for typical data-processing applications is described in [3]. This method does not address real-time constraints and is weak on data abstractions.

Many informal versions of data flow diagrams have been used extensively to model the data transformation aspects of software systems. Data flow diagrams are easy to read, revealing the internal structure of a process and the potential parallelism inherent in a design, making dataflow attractive to designers. We believe an automated prototyping environment should provide graphical capabilities for displaying and updating the system structure of the prototype. However, these informal notations do not provide a unified mechanism to represent all of the relevant attributes of software systems (e.g. timing and control) and are not sufficiently formal to be executable. A more precise model of a dataflow computation has been developed in the context of hardware design. We have extended the model and the notation to include control aspects and critical timing constraints in a two dimensional data flow diagram without losing its natural benefits. These extensions are needed for the design of systems with hard real-time constraints.

2. Modularization

Problem decomposition is a central issue in the design of any large system. Two well known decomposition methods, based on dataflow and control flow, are considered below. We propose a single uniform

decomposition method that combines the advantages of both alternatives, and is applicable to systems with hard real-time constraints.

2.1. Combining Data Flow and Control Flow

Each criterion for decomposing a software system is based on some computational model. Dataflow and control flow are two popular decomposition criteria. The components of a dataflow decomposition are independent sequential processes that communicate by means of buffered data streams, while the components of a control flow decomposition are procedures that are called by and return to a main procedure with a single thread of control. Good modularity is one of the key factors for increasing productivity, since it reduces the debugging effort for producing a correct executable system, and improves the understandability, reliability, and maintainability of the developed system. These features are especially important in rapid prototyping. Iwamoto et al [4] suggest circumstances in which each of the two kinds of decomposition is preferable and give some restrictions sufficient to guarantee that the computed results are independent of scheduling decisions. Their system does not address real-time constraints and is a relatively low-level extension to FORTRAN applications which is subject to many confusing restrictions. They use a dataflow decomposition in cases where there is a mismatch between the structures of the input data stream and the output data stream of an operator, introducing an intermediate data stream of lower level data elements to resolve the structure clash. They also use a control flow decomposition in cases where the data stream forks into several branches and is rejoined, or where the operators on the branches influence each other's results by means of state changes, because in these cases a dataflow decomposition will result in computations whose results can depend on the unpredictable behavior of the process scheduler. An example of the first case is a decomposition with a dispatch operator that recognizes several alternative kinds of inputs and routes them to the appropriate special purpose operator. A dataflow decomposition for such a structure requires extra sequencing information in the data elements to make sure that the result streams do not get out of order when they are merged, since the relative speeds of independent processes are not predictable under the usual interpretation of dataflow. An example of the second case is a transaction with multiple updates to a shared database, where the final state of the database may depend on the arbitrary

order of the updates performed by operators on parallel branches of the data flow graph.

To avoid the problems with dataflow decompositions mentioned above we have developed a new underlying model of computation for PSDL [7], which is based on dataflow and guarantees that the results of a computation do not depend on undetermined properties of the schedulers. Control constraints are combined with the dataflow model to achieve the best modularity with sufficient control information. Dataflow is used to simplify the interactions between modules, eliminating direct external references and communication by means of side effects. The first problem with dataflow decompositions mentioned above does not arise in our model because of a rule in PSDL which says that a composite operator cannot fire again until all of the internal activity associated with the previous firing is complete. This rule provides a kind of mutual exclusion that prevents interference between successive actions by the same operator without preventing concurrent execution of the components of a composite operator. The second problem with dataflow decompositions does not arise in PSDL prototypes because there is no implicitly shared mutable data.

2.2. The PSDL Computational Model

The PSDL computational model is based on enhanced data flow diagrams [7]. An enhanced data flow diagram is a directed graph with associated timing and control constraints. The nodes of the graph are operators and the arcs are data flow paths.

Operators are either ***functions*** (without an internal state) or ***state machines*** (with an internal state). When an operator fires, it reads one input value from each incoming arc, and puts at most one computed output value on each outgoing arc. The firing of an operator can be triggered either by the arrival of a specified set of input data values or by a periodic timing constraint. The firing of an operator and the production of an output value can also be subject to conditional control constraints that depend on locally available data values. This limited facility for interconnecting operators is well matched to the needs of real-time systems, in which each operator must complete its task within a fixed time limit.

A data stream carries values of an abstract data type. Both the built-in and user-definable data types of PSDL are immutable. An immutable type has no operations for changing the state of a data

object, so that all changes appear as newly generated data values rather than as updates to existing data objects. The generic built-in types of PSDL include tuples (records), one_ofs (tagged variants), sets, sequences, maps (lookup tables) and relations. These types provide a powerful facility for defining finite collections of any type of value, making it easy to construct a wide variety of user-defined abstract data types. Conventional data types for numbers, strings, and truth values are also available in PSDL.

Each data stream is either a *dataflow stream*, which guarantees each data element that enters is delivered exactly once, or a *sampled stream*, which guarantees a data element can always be entered into or delivered from the stream on demand, at the cost of replicating elements or discarding older values. A dataflow stream acts like a fifo queue whose length is bounded by 1. A sampled stream acts like a memory cell which always contains the most recent data value that has been put into the stream, and which can be updated at any time. In PSDL the control and timing constraints of the operator receiving a stream determine whether the stream is of the dataflow or sampled variety, in a way that guarantees there will be data values present on all of the input streams of an operator whenever it fires [7]. Dataflow streams are discrete data flows, and sampled streams are continuous data flows. Exceptions are treated as data values of a special data type, which flow down data streams subject to the same rules as ordinary data values.

In PSDL each operator can have a maximum execution time and a maximum response time, which are treated as hard real-time constraints. Operators with real-time constraints are either periodic (synchronous) or sporadic (asynchronous). The firing frequency of each synchronous operator is specified by giving its period. The minimum period between firings is also specified for each sporadic operator, recording the necessary assumptions about worst case operating conditions with respect to asynchronous external events. The individual timing constraints of a real-time system are relatively easy to describe using the facilities described above. However, large real-time systems often contain a mixture of periodic and sporadic operators, with many different frequencies. The interactions between such timing constraints can be quite complex and very difficult to analyze without computer aid. Control constraints can also be associated with operators. These include conditions that act as guards for firing an operator or passing an output value to a data stream, and can control exception conditions or timers.

2.3. Hierarchical Decomposition

The PSDL prototyping method develops a hierarchically structured design by a process of stepwise refinement, guided by the computational model and the software base. At each level, the system at the center of attention is modeled as an enhanced data flow diagram. While the model is created mostly in a top-down fashion, the process is guided by a tool for browsing through the reusable components in the software base. Each of the operators and each of the data types associated with the data streams is given a black box specification and subjected to further refinement. There are three possibilities at this point.

(1) A search of the software base succeeds in retrieving a reusable component whose specifications match those of the required operator or type. The usefulness of partial matches is enhanced by facilities for instantiating generic reusable components and by PSDL control constraints, notably conditional guards that can limit the execution of a reusable component to the cases where its behavior satisfies the requirements of the needed prototype component. In this case all required lower level details are supplied from the software base, and no further effort is required of the prototype designer.

(2) No match is found in the software base, and the behavior of the needed component is sufficiently complex to be decomposable into a network of simpler types and operators. In this case, the component is refined as a lower level dataflow model, and the process is repeated recursively. As in any design process, the skill and experience of the designer are important factors determining the speed and quality of the decomposition. PSDL contains a powerful set of built-in data types and control constraints to aid in this process [7].

(3) No match is found in the software base, and the behavior of the needed component is so simple that further decomposition would not be useful. This case corresponds to an incomplete software base, and should be infrequent for applications with a mature software base. In such a case, a small special purpose module is coded in the underlying programming language (e.g. Ada), and is added to the design database containing the prototype. If future developments in the same problem area are anticipated, then the new module is also marked for addition to the software base. The actual

extension of the software base is done after the rapid prototyping effort is completed, because the process of generalizing the module, developing a good specification for it, and certifying its correctness are time consuming prerequisites for adding a component to the software base.

2.4. Locality and Component Scoping

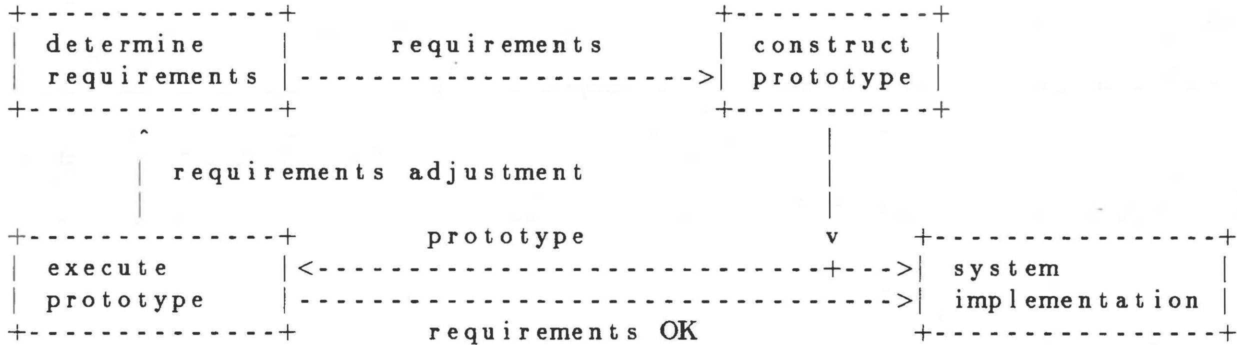
PSDL has been designed to prevent implicit interactions between operators, thereby encouraging model independence. Since there is no global data in PSDL, operators must rely on incoming data streams for all input. Since all PSDL data values are immutable, operators cannot interact by means of state changes in a shared mutable data object. Two PSDL operators cannot interact by means of state changes unless both have explicit data flow connections to the same state machine.

The state of a state machine operator is purely local in PSDL, and can be influenced only by sending data values to one of the input streams of the operator. This is achieved by means of a local name scoping rule for for composite operators. Access to state machines must be local with respect to the design hierarchy: it is not possible to send a data stream directly to a component of a composite operator, since the names of the components are not visible outside the implementation part of the composite. It is also not possible for two composite operators to share the same instance of a state machine as a subcomponent for the same reason (although they could both use different instances of the same generic state machine).

This locality property facilitates the modification of PSDL prototypes, because the number of modules impacted by a change is limited and can be determined by a relatively shallow mechanical analysis of the dataflow structure of the prototype. It also makes it easier to distribute the parts of the computation among several processors, because implicit interactions are difficult to implement in a loosely coupled architecture. The localized nature of the PSDL computational model encourages the prototype designer to first specify abstract state machines or functions, and then to decompose them into loosely coupled networks of independent operators, which is the preferred structure for distributed software.

3. Constructing a PSDL Prototype in the Prototyping Life Cycle

PSDL was designed together with the prototyping method presented in this paper. The steps for constructing a PSDL prototype are described in this section. A diagram which shows the prototyping life cycle and the steps for updating requirements is given below:



We have chosen a real world example, a hyperthermia system, to demonstrate our design method. The reasons for choosing this example are:

(1) The software system used for temperature control in the hyperthermia system is a typical embedded system with hard real-time constraints.

(2) The application is significant and realistic. It is large enough to demonstrate the essential features of large scale prototype programming, but not too large to publish.

The prototype shown in this section was constructed rapidly. The example is used at the end of each subsection to illustrate the decisions and implementation details in the construction of a PSDL prototype.

3.1. Hyperthermia Example

The explanation of the problem addressed in the example consists of two parts. First, we give a general description of a hyperthermia system, where a hyperthermia system is used, and how to separate the prototype of the software subsystem from the whole system. Second, the requirements for the software component of a hyperthermia system are discussed.

Cancer specialists have been trying to selectively destroy tumor cells with heat. A hyperthermia system is a medical device for treating tumors based on this idea, which uses a microwave generator connected to a fine tuner and matching control system. The hyperthermia system uses microwave energy to produce and deliver controlled local therapeutic heating directly to tumors for effective and safe treatment of cancer. A computerized control system is to adjust power output automatically to maintain the therapeutic temperature in accordance with the established patient treatment plan.

The hyperthermia treatment system consists of four subsystems: a computer system, an operator's panel, a microwave generator, and a temperature sensor. The critical subsystem is the computer software which receives input from the temperature sensor and produces control commands to operate the whole system. The computer software controls the rest of the system, which is typical of real-time embedded systems. In order to demonstrate the behavior of the prototype, it is necessary to simulate the properties of the rest of the system that are relevant to the software subsystem.

The behavior of the software subsystem is described by the following informal requirements, which are rough and typical of the initial requirements supplied by a user.

(1) Accept input tumor data in the patient's medical record from an existing source.

(2) Prepare the probes and their corresponding structures in the microwave and temperature sensing systems.

(3) After the preparation is completed, the power generator starts generating microwaves and then the software control system adjusts the intensity of the microwaves sent out in response to inputs from probes in the temperature sensing system. The adjustment should be made according to the data describing the microwave-temperature-time pattern.

(4) The desired hyperthermia temperature indicated for the therapeutic treatment is 42.5^o C. The system should reach the indicated temperature in less than 5 minutes in order to leave sufficient time for the patient treatment plan.

(5) After the system reaches the indicated hyperthermia temperature, it should keep the temperature stable for 45 minutes in order to kill tumor cells. During this period, the treatment system should

adjust the intensity of microwaves to keep the temperature stable with an error tolerance $< 0.1^o$ C.

(6) The software subsystem must appropriately control the other subsystems of the hyperthermia system in order to ensure their correct operation.

3.2. Initial Steps

The initial steps for constructing a PSDL prototype perform a general analysis of the given problem. The purpose of this analysis is to decide what questions the prototype is supposed to answer, to identify which parts and attributes of the system to prototype, and to get the requirements for the prototype.

3.2.1. Decide what Questions the Prototype is Supposed to Answer

The purpose of each prototype is to answer some questions about the system to be designed. The first step in our prototyping method is to determine which questions are supposed to be answered using the prototype. Typical kinds of questions that can be answered using a prototype are whether the proposed system behavior meets user needs, whether system input and output interfaces are acceptable, and whether proposed real time constraints can be satisfied. In the hyperthermia example, the questions we address are whether a real time control system satisfying the requirements is feasible, and whether the proposed control system is safe for use in hospitals.

3.2.2. Identify which Parts and Attributes of the Intended System to Prototype

The next step is to determine which part of the system must be prototyped to answer the questions identified above, which we will call the critical subsystem. In our example the critical subsystem is the software component of the hyperthermia system, because the feasibility and safety of the temperature probes and microwave generator are not in doubt. The critical subsystem has interfaces to the doctor, the temperature probes, and the microwave generator. The attributes of the system that affect safety and hence must be included in the prototype are the treatment temperature and the treatment time. The relation between the microwave power level and the treatment temperature must be determined and simulated to allow the evaluation of the proposed control algorithms for the microwave power level.

3.2.3. Form the Requirements Set for the Prototype

It is necessary to rewrite the requirements for the prototyping system into a clear and brief form because the initial English description is usually long, redundant, and imprecise. PSDL assumes that the requirements are structured as a set of named items. The PSDL facility for recording the correspondence between the requirements and the parts of the prototype works best if each item in the requirements represents a single constraint and different items represent independent constraints. The requirements can be given as shown below, or a more formal notation can be used.

(1) Shutdown -- Microwave power must drop to zero within 300 ms of turning off the treatment switch.

(2) Temperature Tolerance -- After the system stabilizes, the temperature must be kept between 42.4° C. and 42.6° C.

(3) Maximum Temperature -- The temperature must never exceed 42.6° C.

(4) Startup Time -- The system must stabilize within 5 minutes of turning on the treatment switch.

(5) Treatment Time -- The system must shut down automatically when the temperature has been above 42.4° C. for 45 minutes.

3.3. Prototype Construction in PSDL

The steps for constructing a system component in a PSDL prototype are described briefly here and then explained in terms of the hyperthermia example below. The designer first decides on the level of detail that must be represented in the prototype, and writes a PSDL specification for the component. That specification is used as a basis for searching the software base for a reusable software component that can be used to implement the prototype directly. If the search succeeds, the prototype of the component is complete, otherwise the designer attempts to decompose the component into a dataflow diagram consisting of lower level operators and data types. If the decomposition is successful, each of the lower level component is then prototyped using the same method. If a useful decomposition cannot be found, the component is coded in the underlying programming language and eventually added to the software base.

3.3.1. Decide on Level of Detail for the Prototype

One of the most important concerns is how much detail should be included in the prototype. Our guideline is to include the minimum amount of detail implied by the previous choice of what questions to answer (Section 3.2.1) and which part of the system to prototype (Section 3.2.2). At this point, the critical things to keep in mind are:

(1) A prototype system is not a production system. The purpose of a prototype is to provide answers to questions about the requirements and the properties of the proposed system. The prototype has to include only the aspects of system behavior relevant to answering the questions. It does not have to be a complete, reliable, and efficient realization of the proposed system.

(2) For the attributes and subsystem chosen to be prototyped, we do not have to design the prototype in exactly as much detail as the intended system. Functional simulation can be used to reduce the amount of detail which has to appear at a specific working level. Aspects that are not related to the questions can be left out, or represented by low-cost mockups. For example, if the purpose of the prototype is to determine the effectiveness of a proposed control algorithm, then the display formats are not critical and off-the-shelf defaults can be used. Conversely, if the central questions are related to human factors, then the format and placement of the displays may have to be represented in detail, while the data content is not critical, and can be filled in by a random data generator. Extraneous details can be treated as lower level attributes and left to be realized in lower level components if the decomposition is eventually refined further, in response to more detailed questions about the system.

There are several important considerations in determining the level of detail:

(1) Choosing a minimum set of subcomponents at each level of the decomposition hierarchy.

(2) Eliminating unnecessary decomposition if reusable components of the same or nearly the same specifications can be found.

(3) Trying functional simulation in the underlying programming language for the components whenever it can be a simpler way to implement the specification.

3.3.2. Write a PSDL Specification for the Components

PSDL prototype components are either operators or abstract data types. Every component of the prototype will eventually have both a specification and an implementation part in PSDL. Specifications are developed for all of the components at a given level of the hierarchy before any of the implementation parts are considered, and the process is repeated until no more decomposition is needed. The steps to be followed in writing a PSDL specification part for a component are described in more detail below.

The function of each component is clarified by writing its formal description and attribute specifications. Informal English descriptions are written for each component as design documentation if the formal descriptions and the attributes specifications are considered insufficient to describe the components or the design. The specification part of the top level operator in the hyperthermia example is shown below:

```
OPERATOR brain_tumor_treatment_system
SPECIFICATION
 INPUT patient_chart: medical_history, treatment_switch: boolean
 OUTPUT treatment_finished: boolean
 STATES temperature: real INITIALLY 37.0
 DESCRIPTION
 { The brain tumor treatment system kills tumor cells using hyperthermia induced by microwaves. }
END
```

This operator is a state machine. The only component of the state that is needed for the purposes of the prototype is the temperature of the tumor, which is specified to be normal body temperature in the initial state.

The medical_history is an abstract data type appearing as an external input to the system. A partial PSDL specification for this data type is given below. The complete data type has many other operations, but only those related to the brain tumor treatment system are included in the prototype. This illustrates the principle of including only those details needed for the purposes of the prototype.

```
TYPE medical_history
SPECIFICATION
  OPERATOR get_tumor_diameter
    SPECIFICATION
      INPUTS patient_chart: medical_history, tumor_location: string
```

```
    OUTPUTS diameter: real
    EXCEPTIONS no_tumor
    MAXIMUM EXECUTION TIME 5 ms
    DESCRIPTION
    { Returns the diameter of the tumor at a given location,
      produces an exception if no tumor at that location. }
  END
  KEYWORDS patient_chart, medical_record, treatment_record, lab_record
  DESCRIPTION
  { The medical history contains all of the disease and treatment information for one patient. }
END
```

3.3.3. Decompose Components not in the Software Base

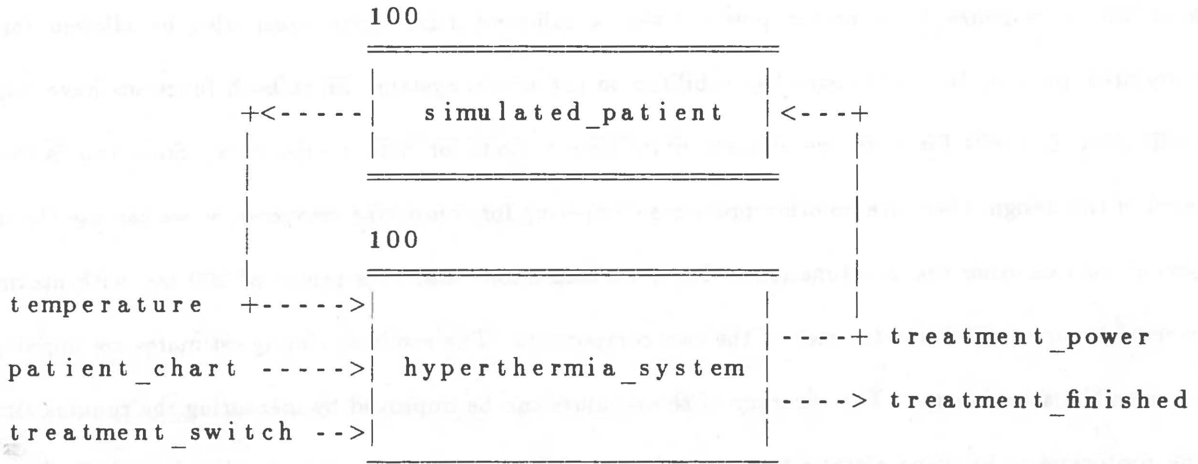
After the components have been identified and specified, the software base is searched to determine if they match existing reusable components. If the retrieval results in a reusable component with sufficiently close specifications, then the implementation is finished. Otherwise complex components are decomposed into more primitive parts, and simple ones are coded in the underlying programming language.

An abstract data type is decomposed by giving a representation for the values of the type in terms of other simpler types and then decomposing the operators of the type. An operator is decomposed by expressing it as a dataflow decomposition involving simpler types and operators. This is done by identifying some useful lower level operators, guided by the concepts to be found in the problem description and requirements, and by considering the operators available in the software base. Such operators can be found by means of the browsing tool, and by retrieving inexact matches to a PSDL specification from the software base. The interconnections of the operators and the types of the data streams are determined next. Control constraints are added where needed, and the timing constraints of the composite component are allocated to its parts at the next lower level.

The operator for the brain_tumor_treatment_system is not available in the software base. A further decomposition is chosen to implement this operator because it is not a simple one. In order to check that the requirements for maintaining the treatment temperature are met, it is useful to divide the prototype into a control system and a simulation of the system to be controlled. In this case the system to be controlled consists of the patient, together with the microwave generator, and temperature measurement system. The only attributes of this system that are needed for the purposes of the prototype are the

temperature reading and the desired power level for the microwave generator. The lower level operators and the data streams connecting them are shown in the PSDL implementation part below.

IMPLEMENTATION
GRAPH



DATA STREAM treatment_power: real

CONTROL CONSTRAINTS
OPERATOR hyperthermia_system
PERIOD 200 BY REQUIREMENTS shutdown
OPERATOR simulated_patient
PERIOD 200
DESCRIPTION { mechanically generated description } END

The brain_tumor_treatment_system is described as a periodically executing feedback loop, which implements a state machine. Each cycle in a PSDL data flow diagram must be cut by a state variable. In the example, the only cycle is cut by the state variable *temperature*. The period of the brain_tumor_treatment_system is chosen to meet the emergency shutdown requirement, which requires the system to set the power to zero within 300 ms of the time the treatment switch is turned off. This requirement will be met if the sum of the period and the maximum execution time of the hyperthermia system do not exceed 300 ms. Since the treatment switch can change at an unpredictable moment, it can be almost a full period before the system samples the value of the switch. The hyperthermia_system must then be executed before a response to the changed input signal can be generated. A tighter time bound cannot be established without looking inside the hyperthermia system, which we want to avoid to preserve the

hierarchical structure of the prototype.

The control function is very important for the safety of the patient, and the temperature tolerances are tight compared with the accuracy of commonly available thermometers, so that ample time must be allocated for the accurate computation of the treatment_power. Since it takes some time for the tumor to heat up in response to a higher power level, a sufficient time delay must also be allowed for the simulated_patient, to avoid control instabilities in the actual system. Since both functions have roughly equivalent demands for time, we allocate equal time periods for both components. Since this is the top level of the design, there are no other processes competing for computing resources, so we can use the entire period for executing the two functions. These considerations lead to a period of 200 ms, with maximum execution times of 100 ms for each of the two components. The resulting timing estimates are approximations to the actual times. The accuracy of these values can be improved by measuring the running time of the prototype or by using a static timing analysis tool for calculating worst case time bounds for loop free code, using the instruction times of a particular compiler and machine, and by constructing an accurate simulation of the power/temperature/time relationship for the human brain, to determine the delay time that must be allowed for the simulated_patient.

The period must also allow the system to make adjustments to the power level fast enough to guarantee that the temperature remains in the allowable range. The correspondence between temperature tolerances and required response times would be determined in practice by means of experiments using the prototype. These experiments are likely to spark changes to the timing requirements as well as to the control algorithms. An important reason for building prototypes of real-time systems is to help determine the exact timing requirements that will suffice to guarantee functional properties of the system such as the temperature tolerance requirement.

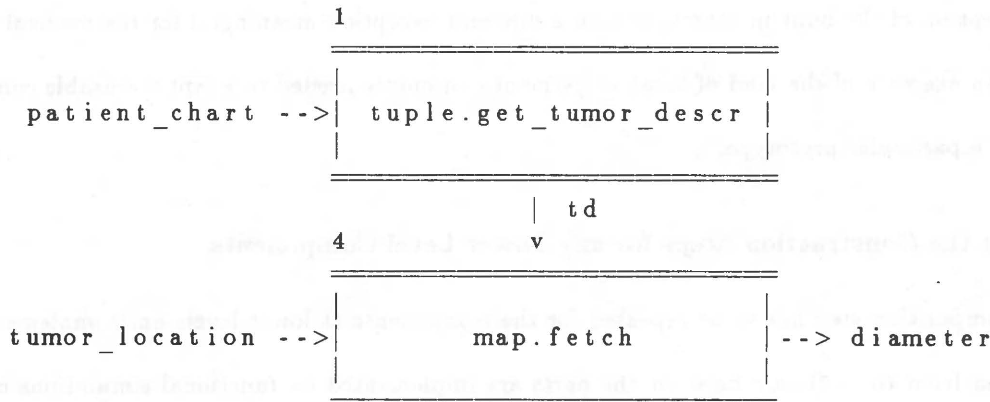
A software simulation of the patient (together with the microwave generator, tuning circuitry, antennas, and temperature sensors) is included to allow the prototype to be tested. This is typical of many real time applications, where the actual environment of the intended system is too dangerous or too expensive to risk while testing a prototype with unknown and possibly faulty properties. We believe that simulations

of the environment of the software system are an essential part of rapid prototyping, and that any language for prototyping real time systems must support the construction of such simulations.

The interface to the brain_tumor_treatment_system includes the abstract data type *medical_history*. The implementation part of medical_history is shown below.

```
IMPLEMENTATION
tuple[tumor_descr: map[from: string, to: real] ]

  OPERATOR get_tumor_diameter
  IMPLEMENTATION
   GRAPH
```




```
    DATA STREAM td: tumor_description
    CONTROL CONSTRAINTS
      OPERATOR map.fetch
        EXCEPTION no_tumor IF not(map.has(tumor_location, td))
  END
END
```

Only one operation of this type, *get_tumor_diameter*, is needed for the prototype, although additional operations for creating values of the type will be needed to exercise the prototype. The type is modeled as a tuple because a real medical history will contain many other components in addition to a tumor description. Since these attributes are not important for the prototype, they are not specified in detail. The tumor description is modeled as a mapping from tumor location to tumor diameter because a patient can have more than one tumor, and because the size of the tumor is the only attribute important for the prototype. Most of the available time is allocated to the table lookup function *map.fetch* because it involves some searching, while the extraction of a fixed component of a record can be done in a small fixed amount

of time. In a mature system, the execution times for operations of built-in types such as ***tuple.get*** would be obtained automatically from the software base.

The data types TUPLE and MAP are built into PSDL, and their implementations are retrieved from the software base. A tuple is the Cartesian product of a number of component types, with symbolic names for the components. A map is a function from a finite subset of one data type to another data type, and is similar to a lookup table. The fetch and get_tumor_description functions are primitive operations of the map and tuple types, so that they need not be refined any further. Tuple is a parameterized family of types with a get_X operation for each component name X. Note the use of the exception control constraint to turn an exception of the built-in map type into a different exception meaningful for the medical_history type. This is an example of the kind of local adjustment commonly needed to adapt a reusable component to the needs of a particular prototype.

3.3.4. Repeat the Construction Steps for any Lower Level Components

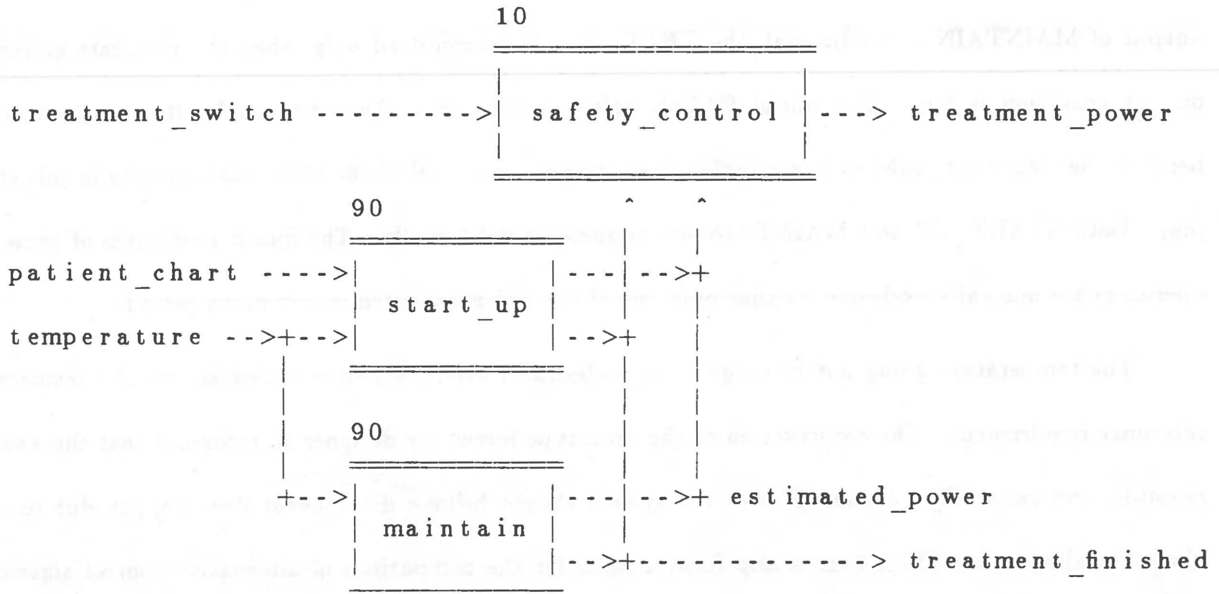
The decomposition step has to be repeated for the components at lower levels until implementations can be retrieved from the software base, or the parts are implemented by functional simulations coded in the underlying programming language.

We show the decomposition of one component at each level of the hierarchy, continuing until we hit the bottom. We have chosen a path down the hierarchy that terminates in a component implemented by a functional simulation coded in Ada. The PSDL description for the hyperthermia_system is shown below.

```
OPERATOR hyperthermia_system

SPECIFICATION
  INPUT temperature: real, patient_chart: medical_history, treatment_switch: boolean
  OUTPUT treatment_power: real, treatment_finished: boolean
  MAXIMUM EXECUTION TIME 100 ms BY REQUIREMENTS temperature_tolerance
  MAXIMUM RESPONSE TIME 300 ms BY REQUIREMENTS shutdown
  KEYWORDS medical_equipment, temperature_control, hyperthermia, brain_tumors
  DESCRIPTION
  { After the doctor turns on the treatment switch, the hyperthermia system reads the patient's medical
    record, and turns on the microwave generator to heat the tumor in the patient's brain. The system controls
    the power level to maintain the hyperthermia temperature (42.5 degrees C.) for 45 minutes to kill the
    tumor cells. When the treatment is over the system turns off the power and notifies the doctor. }
END
```

IMPLEMENTATION
GRAPH



DATA STREAM estimated_power: real
TIMER treatment_time

CONTROL CONSTRAINTS
OPERATOR start_up
TRIGGERED IF temperature < 42.4 BY REQUIREMENTS maximum_temperature
STOP TIMER treatment time
RESET TIMER treatment_time IF temperature <= 37.0

OPERATOR maintain
TRIGGERED IF temperature >= 42.4 BY REQUIREMENTS maximum_temperature
START TIMER treatment_time BY REQUIREMENTS treatment_time, temperature_tolerance
OUTPUT treatment_finished IF treatment_time >= 45 min BY REQUIREMENTS treatment_time

END

This example illustrates the use of an event controlled timer, a conditional output, and conditionally activated operators. The treatment_time timer is reset (to zero) whenever the temperature drops below body temperature (i.e. at the end of a treatment session). The timer is (re)started if the temperature is in the range for effective hyperthermia, and it is stopped if the temperature goes out of the range. The treatment_time timer is used to record the treatment time, and to control the transmission of the output treatment_finished from the MAINTAIN operator.

The MAINTAIN operator always produces the value TRUE for the treatment_finished switch, while the START_UP operator always produces the value FALSE for the treatment_finished switch. Since the output of MAINTAIN is conditional, the TRUE value is transmitted only when the predicate giving the output condition is true. The initial FALSE value persists until the conditional output is transmitted because the treatment_finished is a sampled data stream (as are all of the other data streams in this example). Both START_UP and MAINTAIN are triggered conditionally. The guard predicates of these two operators are mutually exclusive, so that only one of the two is executed in any given period.

The temperature going out of range is an undesirable event, which is prohibited by the *temperature tolerance* requirement. The construction of the prototype forces the designer to recognize that this event is possible, and raises the question of how the system should behave if the event does happen due to some kind of malfunction. These events also form a basis for the comparison of alternative control algorithms for the *maintain* operator. A plausible development history involves prototyping several control algorithms, and comparing their behaviors by monitoring the frequency of *stop timer* events for *treatment_time*.

The specification for maintain operator at the next level of refinement is given below.

```
OPERATOR maintain
SPECIFICATION
  INPUT temperature: real
  OUTPUT estimated_power: real, treatment_finished: boolean
  MAXIMUM EXECUTION TIME 90ms
    BY REQUIREMENTS temperature_tolerance
  DESCRIPTION
  { The power is controlled to keep the power between 42.4 and 42.6 degrees C. }
END
```

The *maintain* operator is a specialized function which is not found in the software base. It is simple enough so that further decomposition is not useful, so we choose to realize it using a functional simulation in the underlying programming language. This decision is recorded in PSDL as follows.

```
IMPLEMENTATION Ada maintain
END
```

At the bottom level of the hierarchy, the PSDL implementation part gives the language in which the

module is implemented, and the name of the implementation module. The name of the implementation module can be different from the PSDL name in cases where a reusable component is retrieved from the software base. An Ada implementation for the *maintain* operator is shown below.

```
PROCEDURE maintain(temperature: IN real; estimated_power: OUT real;
                   treatment_finished: OUT boolean ) IS

  c: CONSTANT real := 10.0;
BEGIN
  IF temperature > 42.5 THEN estimated_power := 0.0;
  ELSE estimated_power := c * (42.5 - temperature);
  END IF;
  treatment_finished := true;
END maintain;
```

This represents a conservative first design. It is very safe in the sense that it is very unlikely for the temperature to go too high. The algorithm is based on a very simple physical model assuming the tumor has no heat loss, so that the rate of temperature increase is proportional to the applied power level. The model has the advantage of being very simple, so that it can be implemented quickly. It is not very accurate, however, since the blood flow through the brain tissue will carry away excess heat. Because of this oversimplification, the first version of the control algorithm may not ever apply enough power to reach the hyperthermia temperature, violating the ***startup time*** requirement.

This is typical of an iterated rapid prototyping effort. The initial version of the prototype will meet some but not all of the requirements. It is often fastest to construct a system by successive approximation, getting a quick skeleton in place that roughly approximates the required behavior, and then using that skeleton as a basis for planning further efforts.

3.4. Evaluate the Constructed Prototype

A prototype is evaluated by running it through the preprocessor and then executing it on sample input data using the PSDL interpreter and debugger. First the designer should test the prototype with respect to the given requirements and fix any design faults. The designer can also answer some of the questions that motivated the prototyping effort at this point. Once the designer believes the prototype behaves according to its specification, the prototype is demonstrated to the users or their representatives, who

identify faults in the observed behavior. These faults are recorded and traced back to the requirements. The behavior of the prototype can also be measured to identify performance bottlenecks.

3.4.1. Test and Debug the Prototype

The PSDL execution support system should perform various kinds of static analysis and report certain classes of design errors. The most important of these are violations of timing constraints, which are discovered by the static scheduler when it fails to find a valid schedule. Such errors can be caused either by inconsistent constraints or by insufficient resources. The first kind of violation is a design fault, while the second is an error in estimating the computing resources needed, and can often be corrected by allowing the scheduler to use more processors. Other classes of errors detectable at this point include type inconsistency of interfaces, and modules without implementations. The last kind of error should be reported but should not be fatal, triggering the automatic creation of a stub module that roughly simulates the missing component by means of randomly generated output data of the appropriate types and a time delay conforming to the specified maximum execution time, if one is given. This should make it possible to simulate important attributes of a partially completed prototype, providing feedback on critical questions early in the process as possible.

As in any other activity, human error is possible in prototype construction. The designer must execute the prototype with a at least a minimal set of test data, to make sure that the behavior of the prototype conforms to the intentions of the design. The test cases produced in this stage should be saved, to form the initial version of the demonstration to the user.

3.4.2. Answer Questions Based on Prototype Execution

The prototyping effort started with the identification of a set of questions. These questions can be answered in part by the designer by choosing relevant input values and observing the behavior of the prototype. Questions about the feasibility of performance constraints fall in this category. Sometimes the questions involve clarification of concepts that are not precisely defined in the requirements. In such a case, it is the designer's responsibility to propose a precise version of the concept, and to demonstrate to

the user how the designer's proposed definitions affect the behavior of the prototype. It becomes the user's responsibility to examine the consequences of the definitions and current interpretation of the requirements, and to judge whether the results are acceptable.

3.4.3. Identify Faults and Trace to Requirements

It is the user's responsibility to identify faults in the demonstrated behavior of the prototype, and to communicate to the designer what aspects of that behavior are not acceptable and why. It is the designer's responsibility to make sure that the full range of behavior which can be manifested by the prototype is included in the demonstration. It is also the designer's responsibility to record the faults, and to trace them to components of the prototype and to the requirements. The cross reference facilities of the design database should aid this stage of the process.

3.4.4. Evaluate the Prototype Design

The example prototype is also evaluated by the designer to identify potential difficulties in the construction of the deliverable version of the intended system. This can include gathering statistics about the firing frequencies of the components of the prototype to identify potential performance bottlenecks, and evaluating the appropriateness of the design concepts used in the prototype, taking the feedback from the users into consideration. This evaluation may lead to new questions for the next iteration of prototyping, and the identification of critical subsystems which are judged to be difficult to design or to be subject to tight performance requirements with questionable feasibility. Analysis of the prototype can also provide a cost estimate for the intended system.

In a serious prototyping effort for a hyperthermia system, it would be determined at this point that the requirements on the temperature are difficult to meet, and that a careful detailed analysis of the technical problems involved is needed. The next step in such a case would be to bring in expert consultants in the areas of biology, heat transfer, and control theory to develop an accurate simulation model of the thermal properties of the human brain, and to propose and analytically investigate the stability and effectiveness of a number of control algorithms. Several alternative versions of the control algorithm in the

maintain operator would be prototyped, and the behavior of each monitored during execution, to verify the opinions of the experts.

3.5. Update the Requirements and Modify the Prototype

The designer has to negotiate with the user a set of extensions and modifications to the requirements, based on the faults identified in the demonstration, and on cost and schedule estimates derived from the prototype. This process is coupled with reinterpretation and redefinition of some of the informal concepts appearing in the original requirements. If such redefinitions are necessary, then another iteration of the prototyping effort and another demonstration and user review is indicated.

In the hyperthermia example, an additional requirement results from the possibility of a premature *stop timer* event for *treatment time*, indicating failure to meet the *temperature tolerance* requirement. This possibility can be detected by execution of the initial prototype, which does not deliver a treatment_finished signal in less than one hour because it cannot maintain the hyperthermia temperature at the required level. The situation is easily detected in a prototype demonstration, but it is easy to overlook in other approaches to requirements analysis that do not involve computer aid. Further consideration of the detected undesirable situation shows it could be due to either software or hardware malfunctions, and that the absence of such a failure cannot be absolutely guaranteed. Consequently a new requirement is needed to specify what "safe operation" means in the event of such a failure.

4. Supporting Environment

An automated support environment is essential for the rapid construction of prototypes. PSDL and its prototyping method have been designed for use in an environment containing a software base management system, a syntax directed editor with graphics capabilities, a design database, and execution support system.

4.1. Software Base

The purpose of the software base management system is to store and retrieve reusable software components [10]. In addition to implementation information, each component in the software base must have

a PSDL specification. The PSDL specification is organized as a set of orthogonal attributes. Component retrieval based on partial matches of these attributes must be provided by the software base management system. A browsing capability similar to the one provided by the smalltalk environment, and a set of operators for tailoring and instantiating generic components [5, 10] should also be provided.

We assume that a sufficiently large practical software base containing high quality reusable components is available. It is important to have a relatively complete set of general purpose components for performing the functions that are common to many systems, such as managing displays, sorting and searching, parsing input strings, and managing lookup tables. Many of these functions can be effectively encapsulated in a relatively small set of abstract data types. It is very important to provide generic versions of the reusable components, since it would otherwise be impossible to design with abstract data types while relying on standard reusable components for performing common utility functions.

4.2. Designer Interface

The designer interface consists of a syntax directed editor for PSDL and a graphics tool for constructing and displaying data flow diagrams. The syntax directed editor helps to speed up the process by eliminating syntax errors, automatically supplying keywords, and prompting the designer with a choice of legal syntactic alternatives at each point. The graphics tool is a part of the syntax directed editor, which provides a graphical view of the dataflow diagram part of the PSDL implementation of a composite module. It helps the designer visualize the relationships between the components of a decomposition by means of a two dimensional data flow diagram, and provides a convenient way to enter and update the decomposition information in the enhanced data flow diagram, which is part of a PSDL implementation of a component. This capability is important because the text form of a data flow diagram is harder to understand than the graphics form.

4.3. Design Database

The design database in the prototyping environment contains a PSDL design. Using a database rather than a text file simplifies job of writing programs that analyze PSDL prototypes, and helps to pro-

vide a continuous cross referencing capability, by maintaining binary relations between pairs of syntactic objects. This cross referencing capability is most important for requirements tracing, and is used mostly in updating the requirements and adjusting the prototype to match. In this case the binary relationship is *satisfies-requirement.* The design database must support retrievals of the form (1) given a requirement, find all the PSDL components that realize it, or (2) given a PSDL component, find all of the requirements it realizes.

4.4. Execution Support System

In order to construct and update a prototype rapidly, the execution support system for PSDL must be efficient. Since prototype modifications are at least as frequent as prototype runs in the expected usage pattern for the execution support system, both preprocessing time and execution time must be given roughly equal weight, making an interpretive implementation strategy preferable to compilation.

The execution support system should be able to save the state of a computation, and to run several alternative versions of a prototype from a given state without repeating the initial part of the computation. This is important because the designer will be engaged in an interactive dialogue with the user, where a given aspect of the prototypes behavior is demonstrated, criticized, and alternatives are explored interactively. Since it may have taken a long user interaction to arrive at the particular state to be examined, it is not acceptable to require the designer and the user to go through many repetitions of that dialogue, or even to incur the delay due to re-running the initial part of the dialogue from a saved script. The need for modifying the prototype in the middle of a run implies the need for a dynamic loader that can be used in the middle of a given execution of the prototype, and for some means for rapidly responding to changed specifications for a component of the prototype. We assume there is a high quality software base of sufficient size to accommodate most common variations on system behavior and a powerful software base management system capable of retrieving reusable components efficiently.

The execution support system consists of a static scheduler, a dynamic scheduler, and a debugger. An initial design for these components is described in [6]. The purpose of the static scheduler is to schedule time for the computations with hard real-time constraints in such a way that all of the timing constraints

will be guaranteed to be met. We use the standard approach of statically allocating time slots sufficient for the worst case execution times of the operators. The abstract treatment of timing information is an important property of the dataflow model since only the essential time orderings among the events in the computation are given. These time orderings act as constraints on the static scheduler, and allow the flexible exploration of schedules for multi-processor configurations. The purpose of the dynamic scheduler is to schedule the computations that do not have hard real-time constraints in time slots not used by the time critical computations. The purpose of the debugger is to exercise the prototype, to collect statistics, and to enable the designer to readily modify it to conform to new or modified requirements.

5. Conclusions

A strategy based on reusable software components is a promising practical approach to rapid prototyping. Good modularity is especially important in prototyping because of the need for making many changes in a short time. A systematic method for prototyping is necessary but not sufficient for the rapid construction of prototypes for large real-time systems. The method must be supported by a clear, simple, and expressive computational model supported by a matching language and automated prototyping environment, to make the process rapid. The same language must be used for designing the prototype and for software base retrievals to realize the benefits of reusable software components. We have designed such a model and language, as well as the kernel of such an environment. The language has been applied to a number of examples, and appears to be quite effective for designing and analyzing real-time systems. Construction of a prototype version of the environment is currently underway.

Better methods for organizing and retrieving reusable components from the software base are important for the practical realization of the prototyping method presented here. There is no previous work on retrieving components from a software base satisfying given specifications. Some promising directions include software base organizations based on adaptive generalization hierarchies, and reusable components retrieval based on specifications with a semantic canonical form. Computer-aided modification of prototype behavior is important for effective responses to user feedback during prototype demonstration sessions. Previous theoretical results on merging software versions [1] can be extended and applied to this problem.

Efficient methods for implementing flexible interpreters with restarting checkpoints is another important area for further investigation.

References

1. V. Berzins, "On Merging Software Extensions", *Acta Informatica 23*, 6 (1986), 607-619.

2. G. Bruno and G. Marchetto, "Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems", *IEEE Trans. on Software Eng. SE-12*, 2 (Feb. 1986), 346-357.

3. J. Cameron, "An Overview of JSD", *IEEE Trans. on Software Eng. SE-12*, 2 (Feb. 1986), 222-240.

4. K. Iwamoto and O. Shigo, "Unifying Data Flow and Control Flow Based Modularization Techniques", in *Proceedings of the Fall COMPCON Conference*, IEEE, 1981, 271-277.

5. Luqi, "Rapid Prototyping for Large Software System Design", Ph.D. Thesis, Computer Science Department, University of Minnesota, 1986.

6. Luqi and V. Berzins, "Execution Aspects of Prototypes in PSDL", Tech. Rep.-86-2, Computer Science Department, Univ. of Minnesota, 1986.

7. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Transactions on Software Engineering*, to appear 1987.

8. A. K. Mok, "The Decomposition of Real-Time System Requirements into Process Models", *IEEE Proc. of the 1984 Real Time Systems Symposium*, Dec. 1984, 125-133.

9. A. Wasserman, P. Pircher, D. Shewmake and M. Kersten, "Developing Interactive Information Systems with the User Software Engineering Methodology", *IEEE Trans. on Software Eng. SE-12*, 2 (Feb. 1986), 326-345.

10. R. T. Yeh, N. Roussopoulos and B. Chu, "Management of Reusable Software", *Proc. COMPCON*, Sep. 1984, 311-320.

REPRODUCED AT GOVERNMENT EXPENSE

Initial Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Center for Naval Analyses 2000 N. Beauregard Street Alexandria, VA 22311	1
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
LuQi Code 52Lq Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	35
Valdis Berzins Code 52Be Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	35
Chief of Naval Research Arlington, VA 22217	2