


# STRUCTURED ANALYSIS TOOL INTERFACE TO THE STRATEGIC DEFENSE INITIATIVE ARCHITECTURE DATAFLOW MODELING TECHNIQUE

THESIS

Kenneth Albert Austin
Captain, USAF

AFIT/GCS/ENG/89D-1

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/89D-1

# STRUCTURED ANALYSIS TOOL INTERFACE TO THE STRATEGIC DEFENSE INITIATIVE ARCHITECTURE DATAFLOW MODELING TECHNIQUE

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

Kenneth Albert Austin, BSEE
Captain, USAF

December, 1989

## *Acknowledgments*

There were many people behind the scenes of this research effort, who played key roles in the development of my thesis. First and foremost is my wife Brenda. With her complete understanding and confidence, my educational goals were realized almost entirely at her expense. Thank you Bern, I love you.

I extend my gratitude to the members of my committee for their contributions. I thank my advisor, Dr. Thomas C. Hartrum, for his guidance and support throughout the research effort. Also, I thank the two other members of my committee, Major Howatt and Major Umphress, for their assistance and expertise.

Finally, I would like to thank fellow students, Capt Morris and Capt Smith, for the insight and assistance they gave me in accomplishing this effort. Their efforts made the most complex issues ahead of me seem simple. I would also like to thank Capt March for his willingness to help me with the Ada programming involved in this research effort. His expertise with Ada was invaluable.

Kenneth Albert Austin

# *Table of Contents*


Page

Acknowledgments . . . . . ii

Table of Contents . . . . . iii

List of Figures . . . . . vii

List of Tables . . . . . viii

Abstract . . . . . ix

I. Introduction . . . . . 1-1

1.1 Problem . . . . . 1-2

1.2 Summary of Current Knowledge . . . . . 1-2

1.2.1 SAtool . . . . . 1-2

1.2.2 SADMT . . . . . 1-5

1.3 Scope . . . . . 1-5

1.4 Assumptions . . . . . 1-6

1.5 Approach . . . . . 1-6

1.6 Equipment and Software Support . . . . . 1-7

1.7 Sequence of Presentation . . . . . 1-7

II. Background Research . . . . . 2-1

2.1 SAtool . . . . . 2-1

2.1.1 Structured Analysis . . . . . 2-1

2.1.2 Relationship between SA and $IDEF_0$ . . . . . 2-5

2.2 SADMT . . . . . 2-6

Page

2.2.1 Platforms . . . . . 2-7
2.2.2 Cones . . . . . 2-8
2.2.3 Processes . . . . . 2-8
2.2.4 Technology Modules . . . . . 2-9
2.2.5 Ports . . . . . 2-12
2.2.6 Links . . . . . 2-12
2.3 SAGEN . . . . . 2-13
2.4 Requirements Analysis Methodology . . . . . 2-14
2.5 Summary . . . . . 2-15

III. Preliminary Design . . . . . 3-1
3.1 Model Development . . . . . 3-1
3.1.1 Abstract Data Model for $IDEF_0$ . . . . . 3-1
3.1.2 Abstract Data Model for SADMT . . . . . 3-7
3.2 Mapping SAtool to SAGEN Development . . . . . 3-7
3.2.1 General Mapping Issues . . . . . 3-11
3.2.2 Mapping Starting Point . . . . . 3-12
3.2.3 Mapping at A-0 Level . . . . . 3-12
3.2.4 Mapping at A0 Level . . . . . 3-13
3.2.5 Mapping at all other Levels . . . . . 3-16
3.3 Summary . . . . . 3-20

IV. Detailed Design . . . . . 4-1
4.1 Overall Approach . . . . . 4-1
4.2 Description of Smith's SAtool . . . . . 4-2
4.3 Prototype Software Interface . . . . . 4-4
4.3.1 Process Hierarchy, Ports, and Data Types . . 4-5
4.3.2 Port Linkages . . . . . 4-7

Page

4.3.3 Port Semantics . . . . . 4-7

4.4 Summary . . . . . 4-7

V. Testing Approach . . . . . 5-1

5.1 General . . . . . 5-1

5.2 Methodology . . . . . 5-1

5.3 Test Description and Results . . . . . 5-2

5.4 Summary . . . . . 5-3

VI. Conclusions and Recommendations . . . . . 6-1

6.1 Summary . . . . . 6-1

6.2 Conclusions . . . . . 6-1

6.3 Recommendations for Future Research . . . . . 6-2

Appendix A. E-R Model of $IDEF_0$ . . . . . A-1

A.1 ACTIVITY Essential Data Model . . . . . A-1

A.2 DATA ELEMENT Essential Data Model . . . . . A-3

Appendix B. SAGEN Syntax . . . . . B-1

B.1 Notational Conventions . . . . . B-1

B.2 Process Hierarchy, Ports, and Data Types . . . . . B-1

B.3 Port Linkages . . . . . B-2

B.4 Process Semantics . . . . . B-3

B.5 Sample SAGEN Source Specification . . . . . B-3

B.6 Sample SAGEN Specification Generated by Interface . B-11

Appendix C. User's Manual . . . . . C-1

C.1 Introduction . . . . . C-1

C.2 General . . . . . C-2

C.3 A-0 Level . . . . . C-3

Page

C.4 A0 Level . . . . . C-4

C.5 Levels of Decomposition below A0 . . . . . C-6

C.6 Invoking SAIS and SAGEN . . . . . C-8

Appendix D. Technology Modules . . . . . D-1

Appendix E. Configuration Guide . . . . . E-1

Appendix F. Summary Paper . . . . . F-1

F.1 Introduction . . . . . F-1

F.2 Background . . . . . F-1

F.2.1 SAtool . . . . . F-1

F.2.2 SADMT . . . . . F-2

F.3 Preliminary Design . . . . . F-3

F.4 Detailed Design . . . . . F-3

F.5 Testing Approach . . . . . F-4

F.6 Test Description and Results . . . . . F-5

F.7 Conclusions . . . . . F-5

Bibliography . . . . . BIB-1

Vita . . . . . VITA-1

## *List of Figures*


| Figure | | Page |
|---|---|---|
| 1.1. | Example SA diagram generated by SAtool | 1-3 |
| 1.2. | SAtool Products | 1-4 |
| 2.1. | SA Box | 2-2 |
| 2.2. | Structured Decomposition | 2-4 |
| 2.3. | Internal View of a Platform | 2-10 |
| 2.4. | Platform with Dynamic Technology Modules | 2-11 |
| 3.1. | Relationships between SAtool and SADMT | 3-1 |
| 3.2. | Activity Data Dictionary Entry Format | 3-3 |
| 3.3. | Data Element Dictionary Entry Format | 3-4 |
| 3.4. | $IDEF_0$ Activity Data Model | 3-5 |
| 3.5. | $IDEF_0$ Data Element Data Model | 3-6 |
| 3.6. | External View of Platform | 3-8 |
| 3.7. | Internal View of Platform | 3-9 |
| 3.8. | SADMT Process Model | 3-10 |
| 3.9. | Sample of A–0 Diagram | 3-14 |
| 3.10. | Sample of A0 Diagram | 3-17 |
| 3.11. | Sample Diagram below A0 level | 3-19 |
| C.1. | Sample of A–0 Diagram | C-3 |
| C.2. | Sample of A0 Diagram | C-5 |
| C.3. | Sample of Leaf Processes | C-7 |
| F.1. | Relationships between SAtool and SADMT | F-3 |

## *List of Tables*


| Table | | Page |
|---|---|---|
| 3.1. | Mapping at A–0 Level | 3-14 |
| 3.2. | Mapping at A0 Level | 3-16 |
| 3.3. | Mapping at all other Levels | 3-17 |
| B.1. | Notational Conventions for SAGEN Syntactical Specification | B-1 |
| B.2. | Variable Definitions for SAGEN Syntactical Specification | B-2 |

## *Abstract*

A software interface was designed and implemented that extends the use of Structured Analysis (SA) Tool (SAtool) as a graphical front-end to the Strategic Defense Initiative Architecture Dataflow Modeling Technique (SADMT). SAtool is a computer-aided software engineering tool developed at the Air Force Institute of Technology that automates the requirements analysis phase of software development using a graphics editor. The tool automates two approaches for documenting software requirements analysis: SA diagrams and data dictionaries. SADMT is an Ada based simulation framework that enables users to model real-world architectures for simulation purposes.

This research was accomplished in three phases. During the first phase, entity-relationship (E-R) models of each software package were developed. From these E-R models, relationships between the two software packages were identified and used to develop a mapping from SAtool to SADMT.

The next phase of the research was the development of a software interface in Ada based on the mapping developed in the first phase. A combination of a top-down and a bottom-up approach was used in developing the software. A summary of the design decisions made in developing the software is presented.

The final stage of the research was an evaluation of the interface using known SADMT architecture descriptions. This technique compared the outputs of the software interface against a known SADMT description. A summary of the results is presented.

# STRUCTURED ANALYSIS TOOL INTERFACE TO THE STRATEGIC DEFENSE INITIATIVE ARCHITECTURE DATAFLOW MODELING TECHNIQUE

## *I. Introduction*

According to Pressman, the ultimate goal for automation in software engineering would be for the computer–aided software engineering tool to cover the entire spectrum of software development (14:192). The user would enter the requirements for a certain problem, and the computer tool would analyze them and design the needed software.

The present status of computer–aided software engineering tools in the software engineering world is far from that ultimate goal. Tools are available to assist in individual parts of software development, but none cover the entire spectrum.

This research effort investigates interfacing two of these tools, Structured Analysis Tool (SAtool) and the Strategic Defense Initiative Architecture Dataflow Modeling Technique (SADMT).

SAtool is a computer-aided software engineering tool that automates the requirements analysis phase of software development using a graphics editor.

SADMT is an Ada based simulation framework that enables users to model real-world architectures for simulation purposes (11). One problem with SADMT is that SADMT models are described using a very complex Ada template to simulate architectures. Because of this complexity, a less verbose language and accompanying translator called SAGEN (SADMT GENerator) was developed to more easily

generate SADMT descriptions. This translator accepts a simpler specification of an architecture and automatically generates the complex SADMT template (9).

The development of an interface between SAtool and SADMT using the SAGEN translator will bring the discipline of software engineering one step closer to the 'ultimate goal' by allowing a model created with SAtool to be simulated by SADMT.

## *1.1 Problem*

The main purpose of this effort is to investigate the usability of SAtool as a graphical front-end to SADMT.

To this end, this research develops a mapping between SAtool and SADMT. A *full mapping* would allow the user to go from SAtool to SADMT or from SADMT to SAtool. A *one-way mapping* enables the user to go from one tool to another. The primary objective of this effort is the development of a one-way mapping from SAtool to SADMT.

## *1.2 Summary of Current Knowledge*

The following sections summarize the current knowledge of SAtool and SADMT.

*1.2.1 SAtool.* In 1987, an Air Force Institute of Technology (AFIT) thesis by Steven E. Johnson (8) resulted in the development of SAtool, a computer-aided software engineering tool based upon Structured Analysis (SA) (8). This tool automated two approaches for documenting software requirements analysis: SA diagrams and data dictionaries. An example of an SA diagram generated by SAtool is shown in Figure 1.1. SAtool was implemented in the C programming language and is currently running on SUN-3 workstations.

SAtool is a graphics editor which allows the analyst to draw SA diagrams and enter portions of the data dictionary for the requirements analysis phase of software development. The remaining elements of the data dictionary are automatically derived from the diagram.

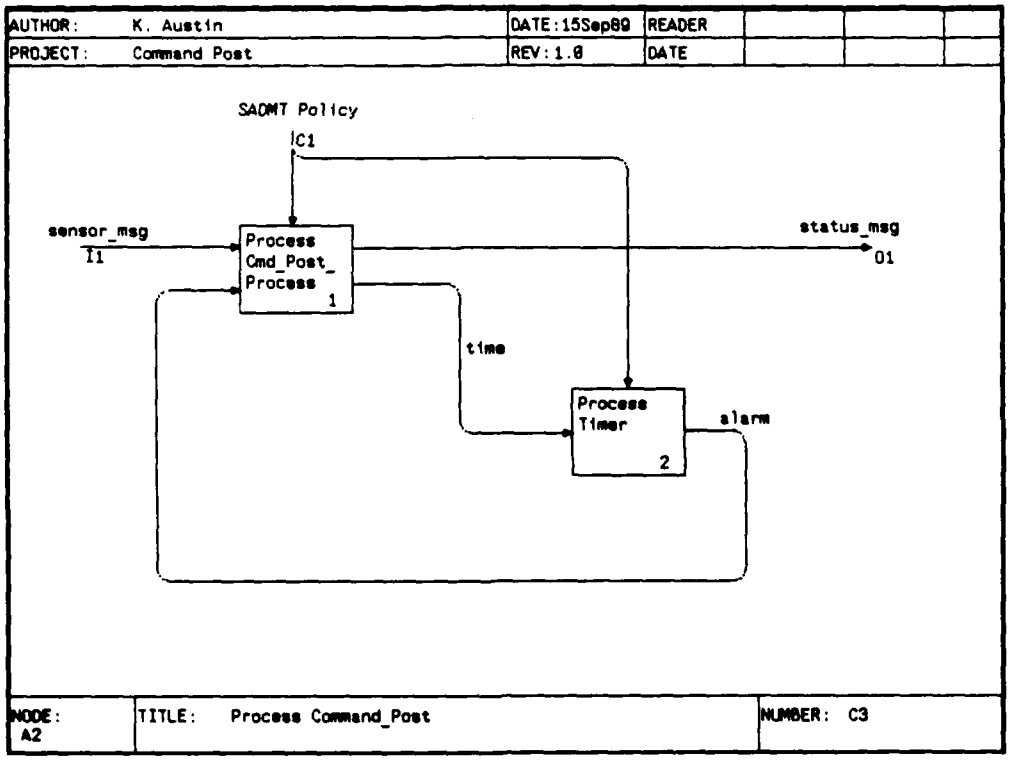


Figure 1.1. Example SA diagram generated by SAtool

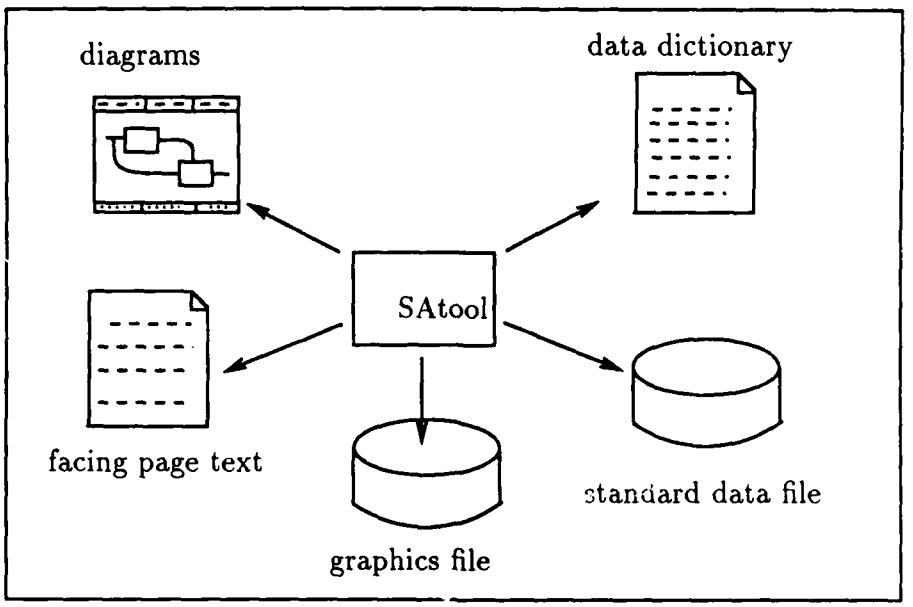


Figure 1.2. SAtool Products

SAtool captures the information entered by the user in a standard data file and stores it in a database. The format for the standard data file is the result of another 1987 AFIT thesis by Ted D. Connally (5). This thesis developed a database interface that integrates the software engineering tools available within AFIT's software engineering laboratory.

The SAtool user can generate a printout of the SA diagram, a facing-page text printout, and a printout of the data dictionary. The analysis results can be saved in a standard data file for uploading into a common database, and the tool saves the graphical drawing information so the user can recall the diagram for editing. Figure 1.2 illustrates the products generated by SAtool.

Further research by Nealon F. Smith (18), in conjunction with this research effort, is enhancing SAtool and reimplementing the tool in the Ada programming language. Although both versions perform the same function, there is one major difference. Smith's version can develop and store multiple levels of SA diagrams in

a single session. Johnson's original version only develops one level SA diagrams per session. Since the function is the same for both, this research effort supports both versions of SAtool.

*1.2.2 SADMT.* SADMT is a system developed for the Strategic Defense Initiative Organization (SDIO) by the Institute for Defense Analyses (IDA) as a method for describing system architectures. SADMT specifically supports Battle Management/$C^3$ and SDI architectures using Ada syntax. The main purpose of SADMT is to capture architectures in early development stages for simulation and evaluation purposes (11). The architectural descriptions use the standard syntax and semantics of Ada to capture the information needed to simulate the system through the SADMT Simulation Framework (SADMT/SF).

As previously mentioned, SADMT descriptions use a complex Ada template to simulate SDI architectures. Because of this complexity, the SAGEN translator was implemented to more easily facilitate SADMT descriptions. The tool accepts a simpler specification of the architecture and automatically generates the required SADMT template (9).

SAGEN and SADMT are both available at AFIT on an Encore Multimax parallel computer, and can also be configured for SUN workstations and other UNIX computers.

## *1.3 Scope*

The intent of this research effort was to show that a mapping between SAtool and SADMT exists, to develop that mapping, and to design an interface between both software packages. For this effort, the interface was developed to translate SAtool output files into SAGEN source files which can be translated to SADMT by the SAGEN translator. The prototype software developed is independent of both SAtool and SADMT. The main objective was to determine the mapping between

the two software packages.

### *1.4 Assumptions*

For the purposes of this research effort, three assumptions were made:

1. The current software for SAtool and SAGEN operates correctly.
2. The users of the independent interface are familiar with SAtool and the structured analysis methodology.
3. The users of the independent interface are familiar with the use of SAGEN and SADMT.

These assumptions limit the research to the development of the independent interface between SAtool and SADMT.

### *1.5 Approach*

This thesis was accomplished in four phases. First, a thorough requirements analysis of the problem was conducted by reviewing the documentation for both software packages.

The second stage of the research was the mapping development. During this stage the mapping between SAtool and SADMT was determined. To accomplish this task, a data modeling technique was used to obtain abstract models of SAtool and SADMT. These abstract models were used to determine the relationship that exists between the two packages.

The third stage of the research was the design and implementation of the interface in software using Ada. All software developed conforms to the standards set forth in AFIT's *System Development Documentation Guidelines and Standards* (7).

The final stage of the research was an evaluation of the interface using examples provided by IDA.

## *1.6 Equipment and Software Support*

The equipment and software for this thesis are currently available at AFIT in the software engineering laboratory. The SUN–3 workstation was used to develop the software interface. The workstation runs Berkeley UNIX and features Suncore graphics and the Sunwindow environment.

## *1.7 Sequence of Presentation*

*Chapter 2* consists of background material for SAtool, SADMT, and SAGEN and is intended to lay a foundation for the remainder of the research.

*Chapter 3* presents the preliminary design of the research effort. First the entity-relationship models developed for both SAtool and SADMT are presented. The chapter concludes by presenting the mapping between the two models.

*Chapter 4* describes the detailed design decisions that were implemented when developing the software for this research.

*Chapter 5* discusses the testing approach used to validate and verify the software interface.

*Chapter 6* presents the research results in terms of the objectives described earlier and draws conclusions based on these results. Several recommendations for further research are also presented.

# *II. Background Research*

The main purpose of this research was to investigate the mapping, if any, that exists between SAtool and SADMT. The purpose of this chapter is to review both packages and summarize the main features of each that must be considered and modeled in order to develop an interface between the two packages. Section 2.1 summarizes the features of SAtool and Section 2.2 summarizes the features of SADMT. Section 2.3 summarizes the features of SAGEN and Section 2.4 presents the modeling technique used in this research effort.

## *2.1 SAtool*

In order to build an interface between SAtool and SADMT, the underlying language that SAtool is automating must be understood. The underlying language is based on Integrated Computer Aided Manufacturing (ICAM) Definition Method zero ($IDEF_0$). $IDEF_0$ is a graphical language which can be used during the requirements analysis phase of software development. In order to discuss $IDEF_0$, it will be necessary to also discuss SA, since $IDEF_0$ is a subset of SA. The following sections give a brief overview of this language.

*2.1.1 Structured Analysis.* In (15), Ross introduced SADT[1] as a generalized language, which allows a complex idea to be represented in a hierarchical, top-down representation. When applied to a team project, SADT increases the team productivity and effectiveness while reducing the project complexity. Specifically, SADT provides methods for:

1. thinking in a structured way about large and complex problems;
2. working as a team with effective division and coordination of effort;

[1]Structured Analysis and Design Technique (SADT) is SofTech's name for SA

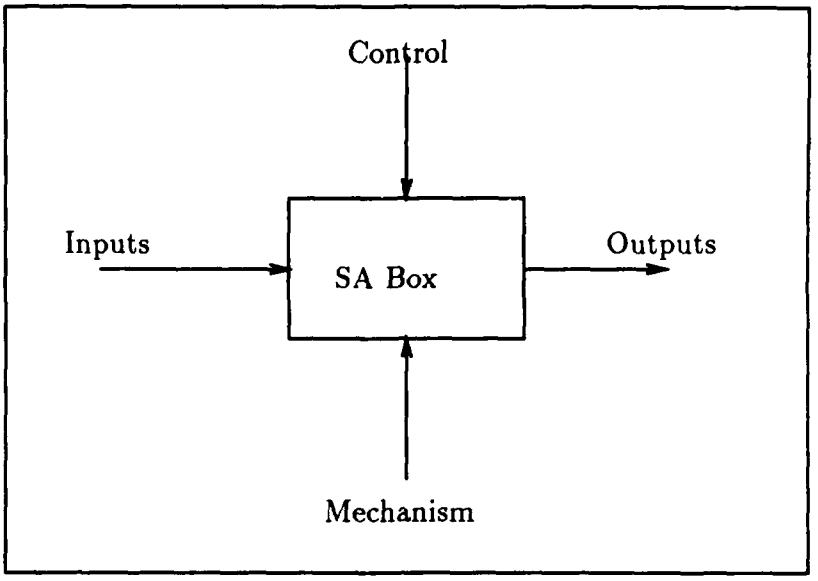


Figure 2.1. SA Box

3. communicating interview analysis and design results in clear, precise notation;
4. documenting current results and decisions in a way which provides a complete audit of history;
5. controlling accuracy, completeness, and quality through frequent review and approval procedure; and
6. planning, managing, and assessing progress of the team effort. (17:11)

In order to understand how SADT works, the graphical language it uses has to be understood. According to Ross, the graphic language notation of SA begins with the SA box. Each side of the SA box represents a specific function: *input* (from the left), *control* (from the top), *output* (from the right), and *mechanism* (from the bottom). The input for each activity is transformed into the output, using the control, by the mechanism (16:26). Figure 2.1 illustrates the SA box.

A control arrow (top of the box) describes the condition or circumstance that governs the transformation. It is important to note the difference between a control arrow and an input arrow. Any incoming arrow is a control arrow unless it only

serves as an input to the SA box (12:3-4). In $IDEF_0$ and SADT syntax, every box must have at least one control arrow and one output arrow.

Incoming arrows to the SA box (left of the box) show the data needed to perform the transformation. Outgoing arrows (right of box) show the data created when the transformation is complete. These terms, input and output, convey the idea that the SA box represents a transition from one state to another (12:3-4).

The mechanism arrow indicates the device that will perform the transformation. The key idea to a mechanism arrow is that the transformation in the SA box is accomplished by the mechanism, which is defined somewhere else (12:3-5).

SA combines graphic features such as lines and boxes with standard written language to create the SA model. This SA model is a hierarchically organized structure of separate diagrams. Each diagram exposes only a limited part of the subject to view, so that very complex subjects can be easily understood. Any part that is not easily understood can be further broken down (15:16). According to Ross, "The human mind can accommodate any amount of complexity as long as it is presented in easy-to-grasp chunks that are structured together to make the whole." (15:17)

Bachert further describes the SA model as an organized sequence of diagrams consisting of SA boxes defining system activities and data arrows defining relationships among the activities. These relationships, shown by placement of the boxes and arrows, conveys a description of the system to the reader. The first diagram in a model is a single box that is a general description of the whole system. The general activity is further defined on the next diagram as three to six detailed activity boxes connected by arrows representing system data. This decomposition process continues until the system is described at the desired level of detail (2:5).

Figure 2.2 illustrates the basic idea behind this structured decomposition. At each level, only the details essential for that level are given. Further details are exposed by moving down in the hierarchy. At the highest level, the A-0 diagram

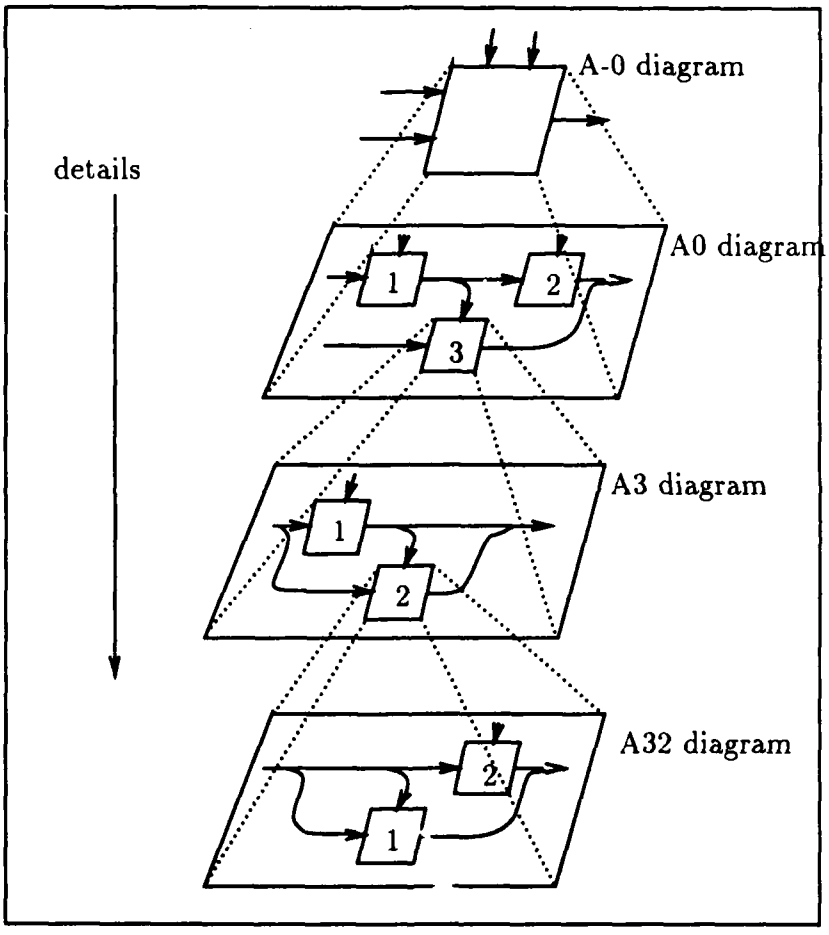


Figure 2.2. Structured Decomposition [Based on (15:18)]

presents a general description of the whole system. The A-0 diagram also limits the context of the system being described. The next level, A0, describes the A-0 diagram in further detail with regard to what activities are done by this system. Finally, each successive level of decomposition further defines its parent activity until an acceptable level of detail is achieved.

SADT provides for two kinds of decomposition, an activity decomposition, and a data decomposition. In the activity decomposition, activities (verbs) are represented by rectangular boxes, and the data (nouns) are represented by arrows flowing into and out of the boxes. In the data decomposition, the boxes represent data (nouns), and the arrows represent activities flowing between the boxes.

*2.1.2 Relationship between SA and IDEF$_0$.* A full implementation of SA includes 40 different language features and the dual decomposition. The United States Air Force Program for ICAM, which is directed toward increasing manufacturing productivity via computer technology, defined a subset of Ross' Structured Analysis language called IDEF$_0$ (12). This function modeling language eliminates some of Ross' language, as well as the data decomposition. Only the activity decomposition is employed. According to the IDEF$_0$ manual:

> The ICAM program approach is to develop structured methods for applying computer technology to manufacturing and to use those methods to better understand how best to improve manufacturing productivity. ... IDEF$_0$ is used to produce a *function* model which is a structured representation of the functions of a manufacturing system or environment, and of the information and objects which interrelate those functions. (12:1-1)

The original intent of the IDEF$_0$ language was to provide a structured approach for computerizing manufacturing processes; however, the language also provides an excellent methodology for performing requirements phase analysis for system development projects. That is precisely why IDEF$_0$ was selected as the language of

choice for performing requirements phase analysis in AFIT's *System Development Documentation Guidelines and Standards* (7).

For further background information on SADT and $IDEF_0$, the reader should refer to (12), (15), (16), and (17).

### *2.2 SADMT*

According to Linn, "SADMT is a technique for thinking about and describing arbitrary systems composed of intercommunicating processes that uses the strong typing and functional facilities of the Ada programming language (11:1)." This formal description ability was needed to facilitate:

1. evaluation and comparison of architectures and development of standard tools for the analysis of architectures,
2. standardization of architectural specification for simulation purposes, and
3. development of new architectures based on characteristics found to be desirable in previous architectural studies. (11:1)

As previously stated, this technique was developed by IDA for SDIO to be used for describing SDI (space based) architectures; however, it is not limited to the problem of describing SDI architectures (11:33). SADMT can be used to describe architectures that are not space based.

> The SADMT description of an architecture is completely represented by an Ada program and this Ada program, in conjunction with the SADMT/SF, may be executed to simulate the architecture. (11:3)

Therefore, SADMT and the SADMT/SF enables the user to describe a system architecture for simulation. According to Linn, the major thrusts of SADMT/SF are:

1. to implement the semantics of the SADMT process model and the associated underlying system of (simulated) time and space,
2. to structure a simulation so as to effect a clean separation of the architectural representation from the simulated environment, and
3. to structure the simulation so as to cleanly separate the processes that deal directly with the environment from the BM/$C^3$ processes that do not. (11:33)

To formally describe the features of SADMT, the entities that it employs must be understood. Specifically, SADMT employs the following entities:

- Platforms
- Cones
- SADMT Processes
- Technology Modules
- Ports
- Links

The remainder of this section will present the features of SADMT that must be incorporated into a formal model in order to determine the relationships that exist with $IDEF_0$, the underlying language upon which SAtool is based.

*2.2.1 Platforms.* An architectural description is represented by *platforms* that are moving in space. In this sense, a platform can be described as any physical entity such as a satellite, weapon system, or any thing of substance (11:4). As a physical entity, a platform has physical characteristics that can be modeled in Ada.

One such characteristic is that a platform can create a platform. This is an important concept captured by SADMT for modeling SDI architectures. For example, a space-based weapons platform can fire (*create*) a missile (*platform*).

*2.2.2 Cones.* The platforms in any single architectural description communicate via *cones.* In this sense, a cone can be described as communication or any *wave-like entity.* Therefore, a platform moving in space becomes aware of other platforms via the cones that each platform *emits* or outputs. A cone has various attributes to describe the information it is emitting and the direction in which it is sent. If a platform falls within a cone's spread, that platform will receive the cone. SADMT refers to the act of a platform receiving a cone as *beaming* (11:33).

A cone can also be used to model a *wave-like* entity as stated above. This is another important concept captured by SADMT for modeling SDI architectures. For example, a space-based weapons platform can fire (*emit*) a laser-beam (*wave-like entity*).

*2.2.3 Processes.* To further describe a platform, SADMT defines an abstract entity called a *process* and a mechanism for specifying a process as a hierarchy of communicating subprocesses (11:7). Therefore, a platform consists of several processes; each of these processes may consist of several other processes; and so on. At the lowest level of decomposition, an SADMT platform is a group of leaf processes (processes that are not decomposed) and a network of communication links between these processes. The leaf processes represent the logical processing associated with a platform and as such, contain all the semantic processing that each platform conducts. These leaf processes are described in Ada according to a very specific template using Ada tasks.

Associated with each platform are some number (possibly zero) of SADMT processes that represent the logical processing that the platform is modeling. These are called the logical processes of the platform.

In addition, each platform contains a set of special SADMT processes called *technology modules* that interface directly with the simulated environment (11:33). The technology modules receive all their SADMT/SF inputs from the PIG (Platform

InterfacinG) process. Every platform must contain a PIG process since this is the platform's interface to SADMT's simulated environment. The PIG process informs a technology module that:

1. the platform has collided,
2. the platform has received a beaming, and/or
3. an event of interest to the platform has occurred (11:44).

Figure 2.3 illustrates a platform, and how a platform is further described by SADMT processes. This illustration contains two logical ($BM/C^3$) processes, $k$ technology modules, and the required PIG process.

*2.2.4 Technology Modules.* As stated previously, the technology modules of a platform are represented as SADMT processes. The technology modules are the only processes that may interface directly with the environment (SADMT/SF). Their responsibility is to manage the interaction between the logical processes of the platform and the simulated physical environment (11:44). Therefore, a platform's technology modules are the actual SADMT processes within the platform that *emit/receive* cones to/from other platforms within the system being modeled.

Since technology modules model technology, they are intended to be reusable across different architectures (11:33). A list of available technology modules provided by IDA is shown in Appendix D.

SADMT also defines a special technology module called a *dynamic technology module* that is automatically placed on every platform in an architectural description. SADMT automatically inserts any defined dynamic technology modules on each platform as they are created, unless a platform specifically excludes the dynamic technology module (11:45).

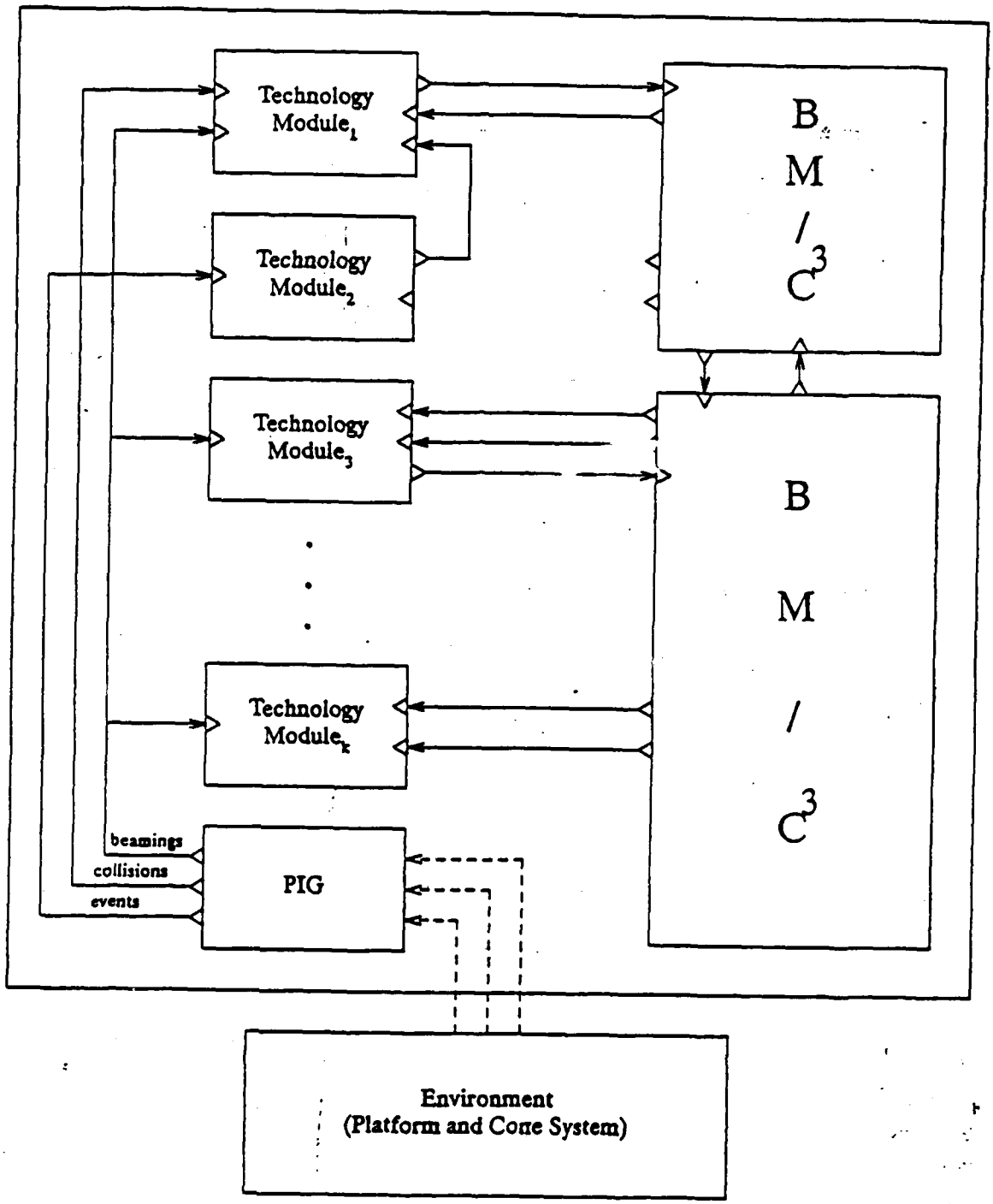


Figure 2.3. Internal View of a Platform (11:34)

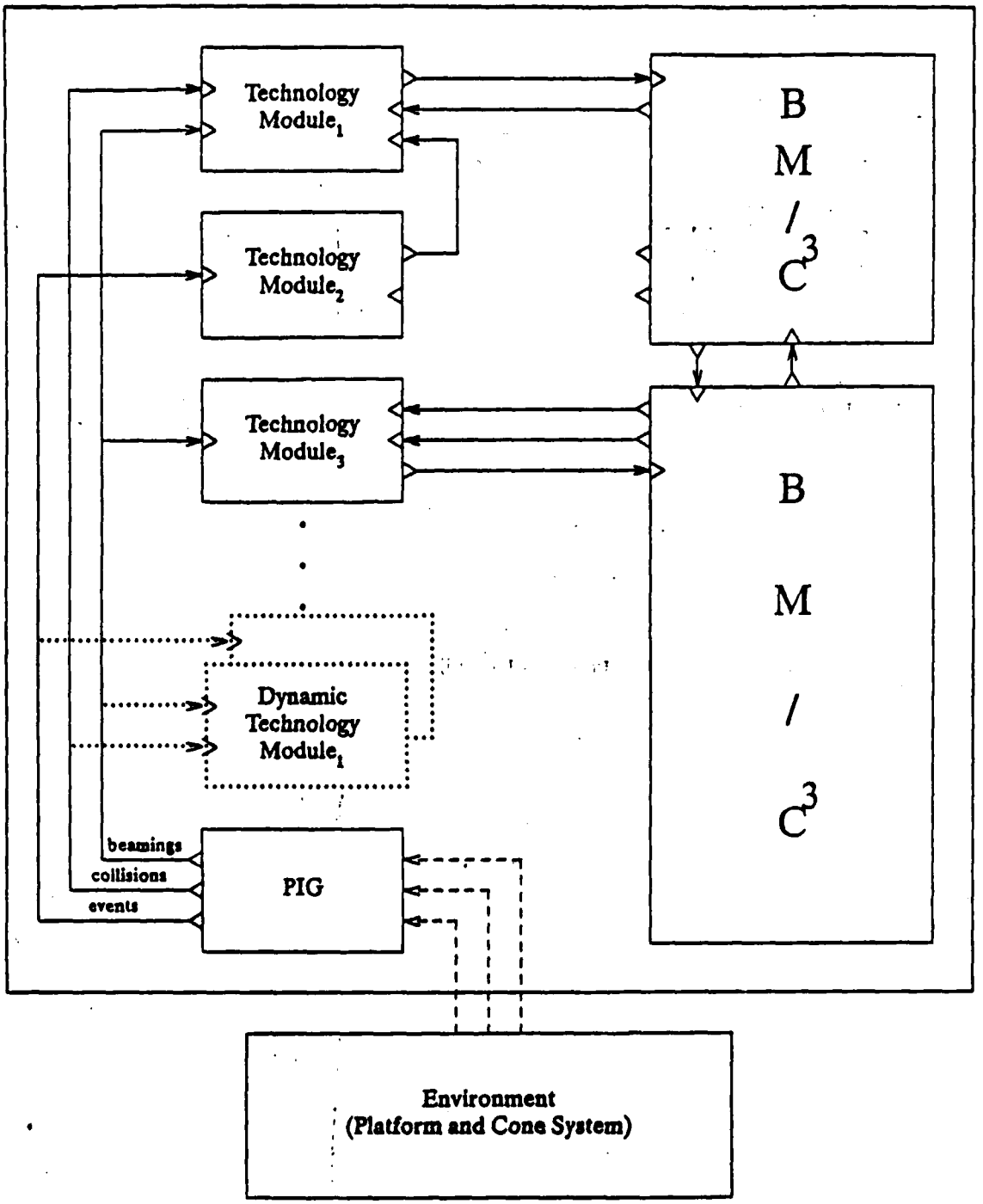


Figure 2.4. Platform with Dynamic Technology Modules (11:46)

A dynamic technology module enables the user to model such things as 'radar sensor returns' without explicitly placing a technology module for 'sensor returns' on each platform. The SADMT/SF automatically does that for the user (11:45).

Figure 2.4 illustrates the internal view of a platform that contains dynamic technology modules.

*2.2.5 Ports.* The SADMT model of processes requires that processes operate by sending and receiving messages to other processes and that there be no interprocess communications except via this message interface. The message interface between any two processes is strongly typed with respect to the domain of message values and to the windows, or *ports*, through which messages are passed (i.e. each port has a specific data type with which it is associated and only data of that type may enter or leave a process through that port). All interprocess communication is accomplished via these ports. A port is either an input port or an output port; SADMT makes no provision for bidirectional ports. In addition, a port can be selectable, allowing the user to specify to/from which port to send/receive a message (11:7).

*2.2.6 Links.* SADMT also defines how these interprocess connections, or *links*, are specified. SADMT defines three types of interprocess communication links:

- *internal link* – connects an output port of a subprocess to an input port of another subprocess of the same parent process;
- *input-inherited link* – connects an input port on a parent process to an input port on one of its child subprocesses; and
- *output-inherited link* – connects an output port on a child subprocess to an output port on parent process.

In each case, the data type of the connected ports must be the same (9:2). SADMT allows for multiple links to be connected to a single input port or output port as

long as the link's data type matches the port type. Since more than one link can come into a port or go out of a port, SADMT defines selectable ports to allow the user to specify exactly which link is to be used when exiting/entering a port.

For further background information on SADMT, the reader should refer to (1), (9), and (11).

### *2.3 SAGEN*

As discussed in Section 1.2.2, SAGEN is a less verbose language than SADMT that can be used to generate the complex Ada template for SADMT. The SAGEN translator can be used to create individual platform descriptions, individual process descriptions, individual technology module descriptions, and individual dynamic technology module descriptions. Therefore, to create a system description, SAGEN could be invoked once for each one of the above entities as needed until all were modeled.

According to (9), when using SAGEN an architecture is described in three parts:

1. Process hierarchy, ports, and data types,
2. Port linkages, and
3. Process Semantics.

The correct syntax for specifying each of these parts is specified in detail in Appendix B. After the correct syntax is presented, several examples are presented.

Whether SADMT or SAGEN is used to create the SADMT description of the system being simulated, a *main program* must be written in addition to the SADMT descriptions. This main program creates and initializes each platform, initializes the simulation's initial configuration, and starts the simulation (11:45-51). Generation of this *main program* is beyond the scope of this effort.

## *2.4 Requirements Analysis Methodology*

When designing software systems, the software engineer must use a requirements analysis methodology to develop a description of the software being designed. According to Pressman,

> A requirements analysis methodology combines systematic procedures and a unique notation to analyze the informational and functional domain of a software problem; it provides a set of heuristics for partitioning the problem and defines a mode of representation for both logical and physical views. (14:163)

For this research effort, the Entity-Relationship (E-R) Model will be used to model $IDEF_0$ and SADMT. The E-R model is most frequently used for database design; however, it also serves well as a data model for requirements analysis. According to Chen:

> The entity-relationship model adopts the more natural view that the real world consists of entities and relationships. It incorporates some of the important semantic information about the real world. (4:9)

It is precisely for this reason, the E-R model is the requirements analysis methodology used to model SAtool and SADMT in this research effort. The E-R model can capture some of the semantic information of the real world it is modeling.

Korth further defines the entity-relationship model:

> The entity-relationship (E-R) data model is based on a perception of a real world which consists of a collection of basic objects called *entities*, and *relationships* among these objects. An entity is an object that exists and is distinguishable from other objects. The distinction is accomplished by associating with each entity a set of attributes which describes the objects. ... A relationship is an association among several entities. (10:6)

Once an E-R model is developed, it can be graphically represented by an *E-R diagram.* The following components are used in an E-R diagram:

- *Rectangles* – represent entities and entity sets,
- *Ellipses* – represent attributes of entities,
- *Diamonds* – represent relationships between entities, and finally
- *Lines* – which link the attributes to entities and entities to relationships (10:7).

Using these graphical constructs, the E-R diagram illustrates the entities and their relationships to the reader. Numerical descriptions are also used on E-R diagrams to specify numerical relationships between entities. These numerical relationships enable the E-R diagram to represent a hierarchy between the entities. The next chapter will present several E-R diagrams to model SAtool and SADMT.

### *2.5 Summary*

This chapter has presented background material for SAtool, SADMT, and SAGEN. This review has summarized the main features of each that must be considered and modeled in order to develop an interface between the two packages. Additionally, this chapter presented the E-R model as the requirements analysis methodology used in this effort.

SAtool is a computer-aided software engineering tool based on $IDEF_0$. $IDEF_0$ is a graphical language which can be used during the requirements analysis phase of software development. This language combines graphic features such as lines and boxes with standard written language to create a model. This model is a hierarchically organized structure of separate diagrams, each exposing only a limited view of the subject. This limited view enables the user to easily understand complex subjects. Any part that is not easily understood can be further broken down.

SADMT is a modeling technique for thinking about and describing systems composed of intercommunicating processes. This technique was developed by IDA

for SDIO to be used for describing and simulating SDI architectures. The technique uses the strong typing and functional facilities of the Ada programming language to describe these SDI architectures. These architectural descriptions are represented by *platforms* moving in space communicating by emitting and receiving *cones*. To further describe a platform, SADMT defines an abstract entity called a *process* and a mechanism for specifying a process as a hierarchy of communicating subprocesses.

SAGEN is a less verbose language than SADMT that can be used to generate the complex Ada template for SADMT. SAGEN can be used to create individual platform descriptions, and individual process descriptions. When using SAGEN, an architecture is described in three parts: process hierarchy, ports, and data types; port linkages; and process semantics.

The E-R model is based on a perception of a real world which consists of a set of basic objects called *entities* and *relationships* among these objects. The E-R model is most frequently used for database design; however, it also serves well as a data model for requirements analysis. Once an E-R model is developed, it can be graphically represented by an E-R diagram. The E-R diagram illustrates the entities and their relationships in the E-R model.

# *III. Preliminary Design*

The previous chapter presented background material describing SAtool, SADMT, and SAGEN. The purpose of this chapter is to assimilate the features of SAtool and SADMT into formal models, and to develop a mapping between those models. This chapter presents an E-R model of $IDEF_0$, an E-R model of SADMT, and finally, develops a preliminary design for the mapping between SAtool and SADMT.

## *3.1 Model Development*

In order to develop an interface between SAtool and SADMT, the underlying entities of each package have to be modeled to determine the relationships, if any, that exist between the two languages. Figure 3.1 illustrates this idea.

This section presents E-R models for both $IDEF_0$ and SADMT. As shown in Figure 3.1, once the entities from both packages are identified, the relationships between the two can be identified and developed.

*3.1.1 Abstract Data Model for $IDEF_0$.* The abstract data model for $IDEF_0$ presented here was developed by a collaborative effort with fellow AFIT students

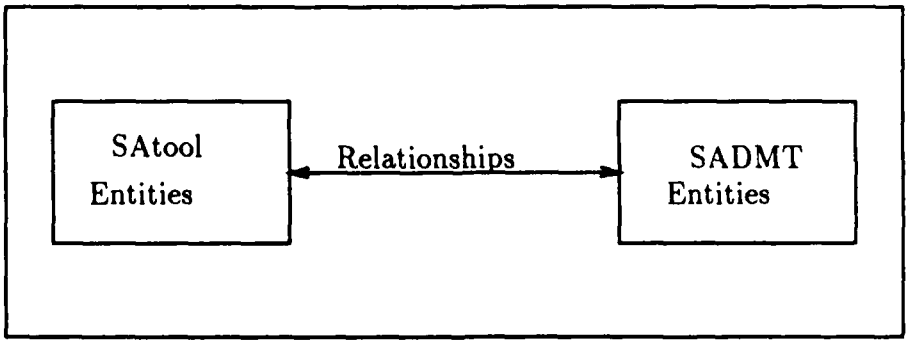


Figure 3.1. Relationships between SAtool and SADMT

whose theses were being developed in conjunction with this effort. Specifically, Morris (13) and Smith (18) were also developing research that required a formal model for $IDEF_0$.

The basic premise in the development of this abstract data model is that a given $IDEF_0$ diagram is actually just a graphic representation of a more fundamental underlying model, which can be represented by any number of diagrams. Therefore, two models were developed: the *abstract data model*, which is the underlying fundamental model, and the *drawing model* which defines one of many possible graphical representations of the abstract model.

Only the abstract data model is needed to define the SADMT input model; therefore, only the abstract data model is presented here. The reader should refer to (13) and (18) for a discussion of the drawing model.

As stated in Section 1.2.1, SAtool uses data dictionaries to store information derived from the $IDEF_0$ diagrams. This provides a more structured representation of textual information than is required by SADT or $IDEF_0$. There are two types of data dictionary entries as defined in earlier AFIT research (5). The first type contains information about the SA boxes or *activities*. The format for the activity data dictionary entry is shown in Figure 3.2. The second type contains information about the arrows or *data elements* connecting the SA boxes. The format for the data element data dictionary entry is shown in Figure 3.3. Both of the data dictionaries have predefined field lengths for each entry.

In order to allow for understandable, yet complete diagrams, the E-R model was done in two figures that complement each other. Each model is based on the respective data dictionary formats mentioned above. Figure 3.4 shows the activity model with the details about data elements left out. Figure 3.5 shows the data element model while leaving out the details about the activities.

Both of these E-R models are discussed in detail in Appendix A.

NAME: (of activity) C25
TYPE: ACTIVITY
PROJECT: (Project name) C12
NUMBER: (Node number of this activity) C20
DESCRIPTION: (Multiple lines allowed) C60
INPUTS: (Multiple entries on separate lines allowed.) C25
OUTPUTS: (Multiple entries on separate lines allowed) C25
CONTROLS: (Multiple entries on separate lines allowed) C25
MECHANISMS: (Multiple entries on separate lines allowed) C25
(NOTE: On the above four fields, the data element name should be used.)
ALIASES: (Names of aliases, multiple entries allowed) C25
COMMENT: (Why is this needed ? One line per alias) C60
PARENT ACTIVITY: (Name of parent activity) C25
REFERENCE: (Paragraph number of requirements, etc., multiple lines) C60
REF TYPE: (Type of reference) C25

VERSION: (version of this data dictionary entry) C10
VERSION CHANGES: (What's different ?) C60
DATE: (of this entry) C8
AUTHOR: (of this entry) C20

Figure 3.2. Activity Data Dictionary Entry Format (7:12)

NAME: (of this data item) C25
TYPE: DATA ELEMENT
PROJECT: (name of project) C12
DESCRIPTION: (Multiple lines allowed.) C60
DATA TYPE: (if known) C25
MIN VALUE: (if applicable) C15
MAX VALUE: (if applicable) C15
RANGE: (if applicable) C60
VALUES: (allowable values, if appropriate. Multiple lines allowed.) C15
PART OF: (parent data element) C25
COMPOSITION: (sub elements, if any. Multiple lines allowed) C25
ALIASES: (Names of aliases, multiple lines allowed) C25
    WHERE USED: (Where does this alias occur ?) C25
    COMMENT: (Why is this alias needed ?) C60
SOURCES: ($IDEF_0$ activity names) C25
DESTINATIONS: ($IDEF_0$ activity names)
    INPUT: (Where this data item is an input) C25
    CONTROL: (Where this data item is a control) C25
REFERENCE: (Paragraph number of textual requirements statement, etc.) C60
    REF TYPE: (Type of reference) C25

VERSION: (version of this data dictionary entry) C10
VERSION CHANGES: (What's different ?) C60
DATE: (of this entry) C8
AUTHOR: (of this entry) C20

Figure 3.3. Data Element Dictionary Entry Format (7:14)

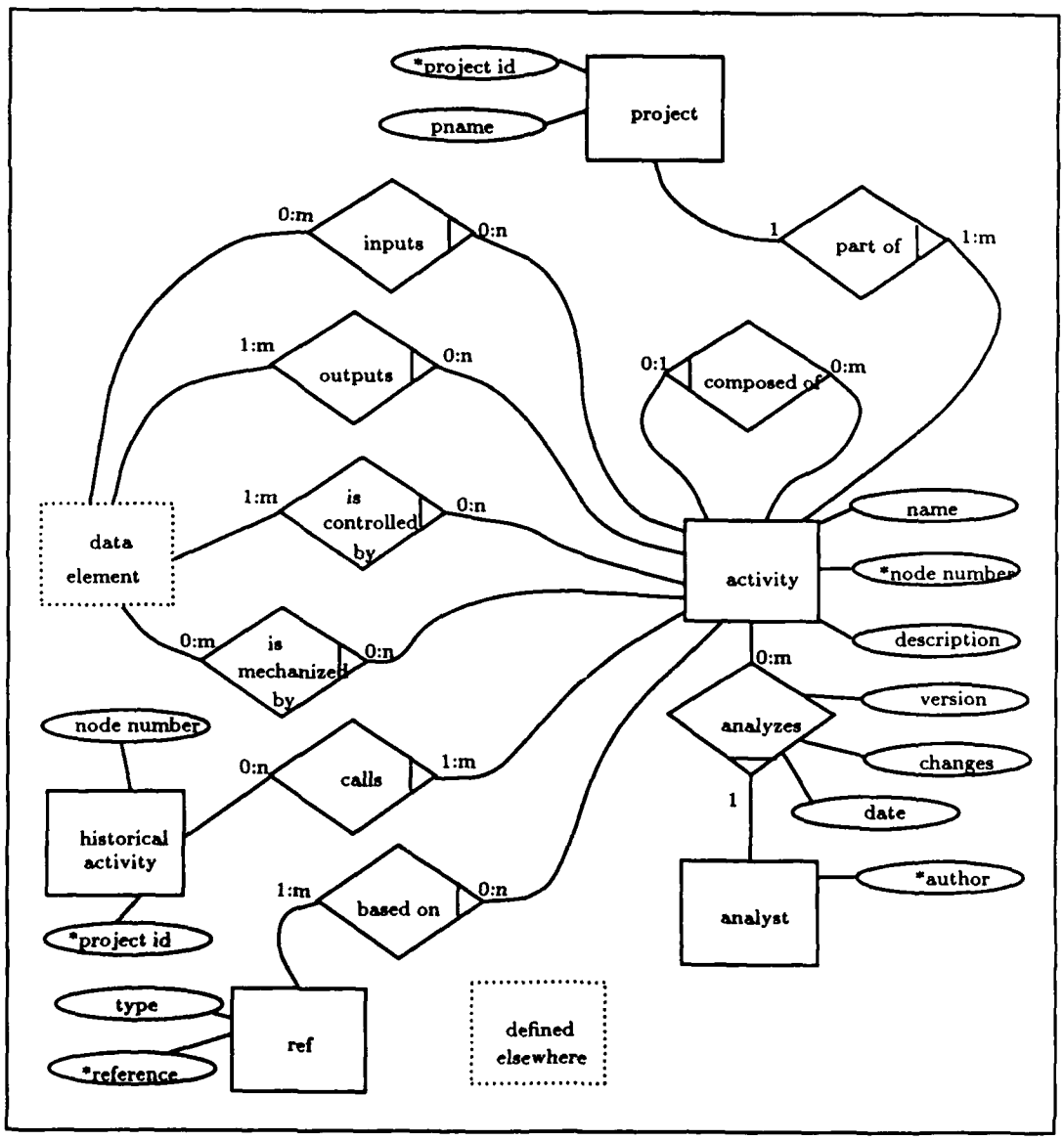


Figure 3.4. $IDEF_0$ Activity Data Model

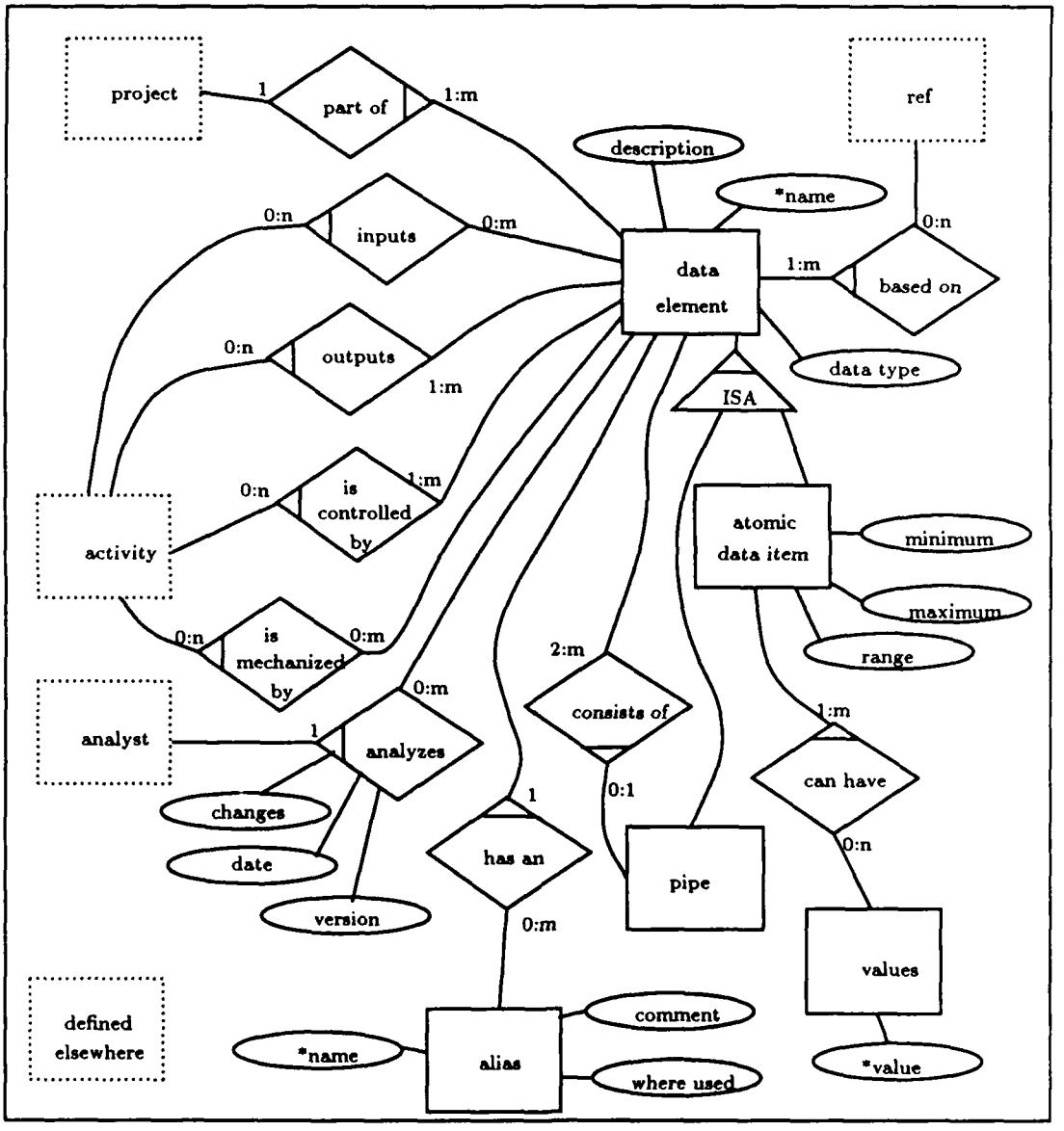


Figure 3.5. IDEF$_0$ Data Element Data Model

*3.1.2 Abstract Data Model for SADMT.* Based on the information presented in Section 2.2, the data model for SADMT has three different views:

- *External View of a Platform.* An SADMT architectural description defines a configuration of platforms moving in space, communicating via cones. Therefore, this external view of a platform encompasses the arbitrary system being described. Figure 3.6 illustrates the E-R diagram for this external view of the abstract data model for SADMT.
- *Internal View of a Platform.* To further describe a platform, SADMT defines an entity called a process. A process is further defined as a hierarchy of communicating processes. SADMT also defines special processes called technology modules that govern a platform's interaction with its environment. Figure 3.7 illustrates the E-R diagram for this internal view of the abstract data model for SADMT.
- *SADMT Process Model.* SADMT further defines how SADMT processes pass messages, the ports through which the messages are passed, and finally how those ports are linked together. Figure 3.8 illustrates the E-R diagram for this SADMT Process model.

## *3.2 Mapping SAtool to SAGEN Development*

The previous section presented E-R models for the general languages of both $IDEF_0$ and SADMT. The purpose of this section is to discuss the one-way mapping from SAtool to SAGEN. Since this research effort's main purpose was to map SAtool into SAGEN, the mapping development is driven by the SAGEN input requirements discussed in detail in Appendix B. Close observance of the SAGEN requirements indicates that a SAGEN specification does not require all of the features of SADMT that were modeled in the SADMT E-R diagrams shown in Figures 3.6, 3.7, and 3.8.

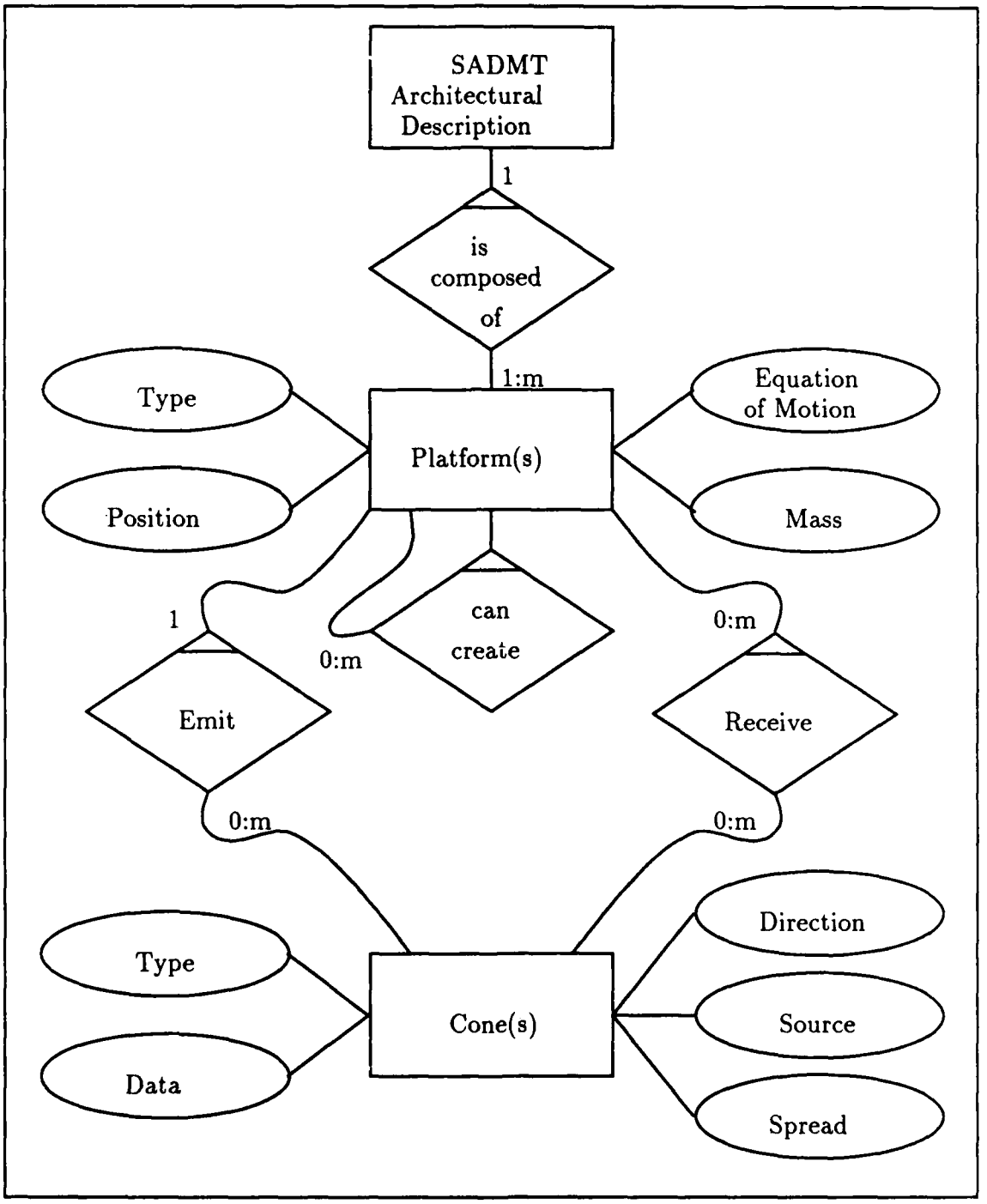


Figure 3.6. External View of Platform

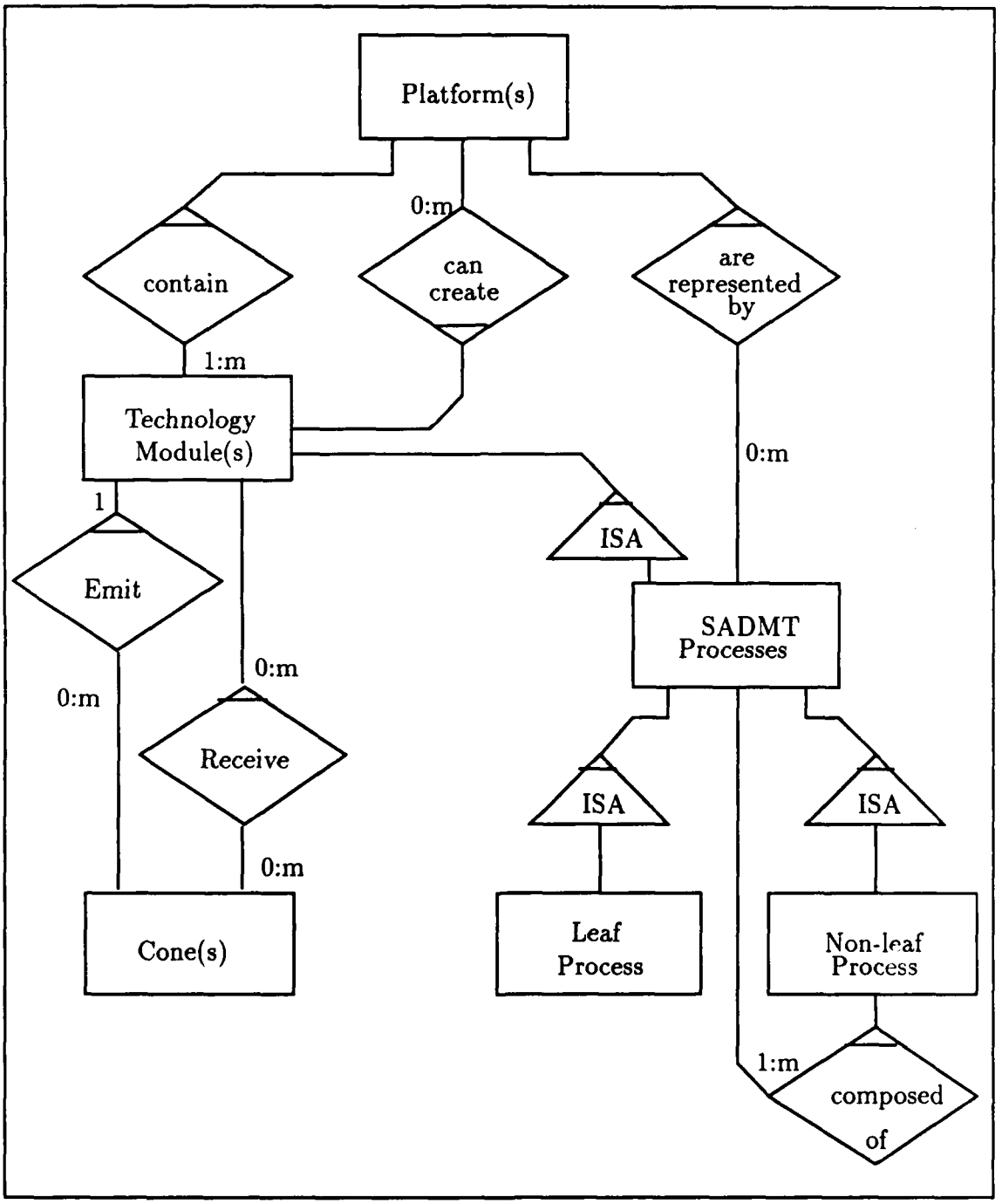


Figure 3.7. Internal View of Platform

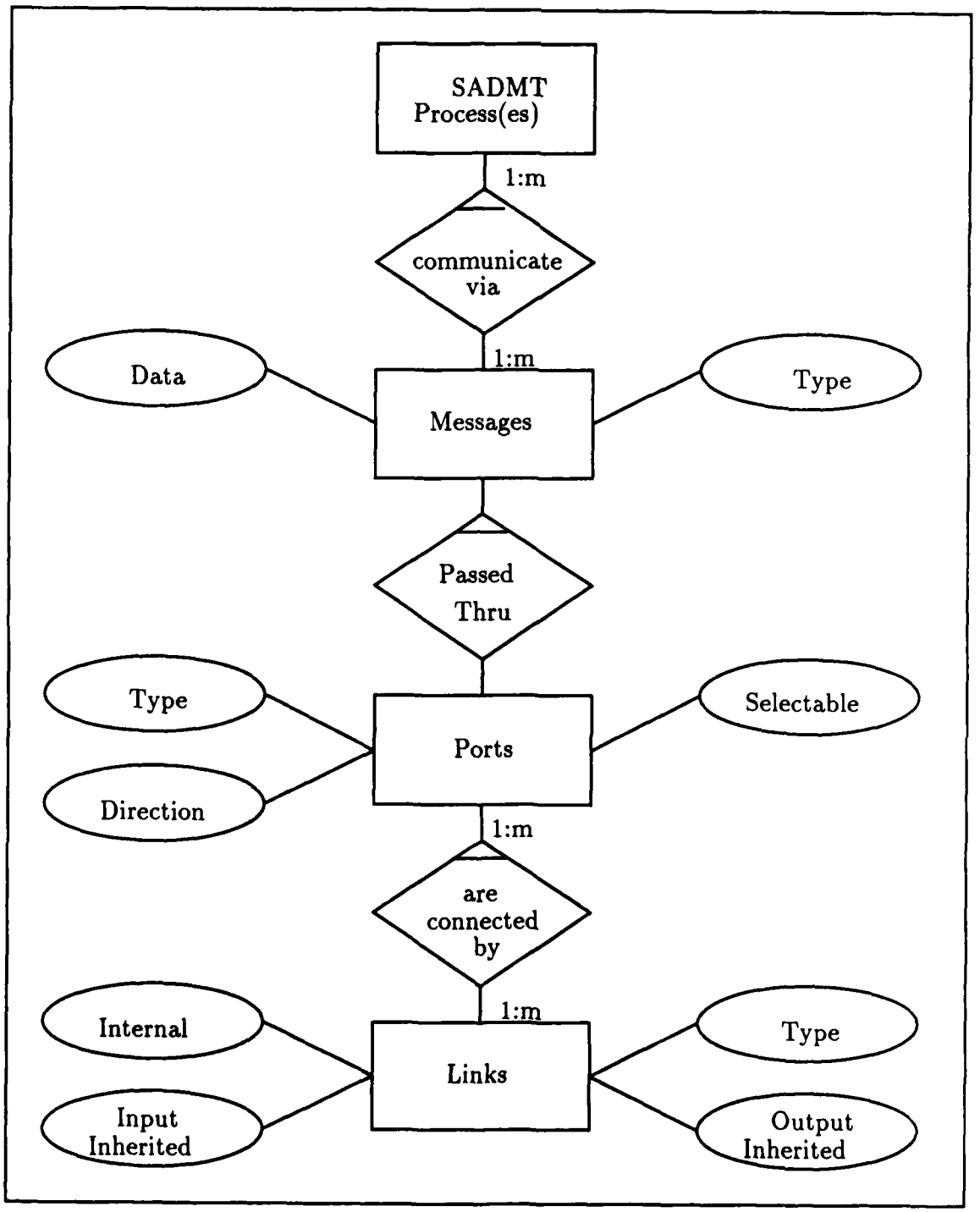


Figure 3.8. SADMT Process Model

Therefore, the approach taken was to find an $IDEF_0$ construct to represent each required SAGEN feature.

This section discusses some general mapping issues, the initial starting point of the interface, the external mapping at the A–0 level, the internal mapping at the A0 level, and finally, the mapping at all subsequent levels of decomposition.

*3.2.1 General Mapping Issues* This section describes some of the general issues that were considered when developing this mapping. These issues address $IDEF_0$ features that are not required for SADMT descriptions that can be used to describe SAGEN input requirements. Additionally, several SAtool underlying activity data dictionary entries are presented that can be used to represent SAGEN features.

The first features to be discussed are $IDEF_0$ control arrows and mechanism arrows. Since SADMT is a dataflow modeling technique, dataflow is represented by inputs and outputs and there is no need for control arrows or mechanism arrows. Therefore, this mapping does not require that $IDEF_0$ activities have a control arrow. Similarly, this mapping does not necessarily use mechanism arrows in their traditional sense. Therefore, both control arrows and mechanism arrows can be used to represent required SAGEN features in the mapping.

The second issue to be discussed is the $IDEF_0$ 'pipe' construct. The traditional use of an $IDEF_0$ pipe enables the user to perform several pipe operations. The *branch* and *join* constructs enable the user to represent two data elements as one by joining them together if they are of the same type. The *bundle* and *spread* constructs enable the user to represent two data elements as one by joining them together if they are of different types. This mapping will support the *branch* and *join* constructs but not the *bundle* and *spread* constructs. Since dataflow in SADMT is strongly typed, only pipe constructs that maintain typing will be used.

The remaining issues discuss underlying activity data dictionary entries that

can be used to represent SAGEN features. The *ALIAS* field for each activity can be used to alias that activity. Throughout this mapping, the *ALIAS* field is used to name each activity with a noun phrase.

Finally, the *DESCRIPTION* field for each activity can be used to provide a textual description of what that activity does. In this mapping, the description field is used to describe the platform or process which that activity represents. Additionally, if the activity represents a leaf process, the *DESCRIPTION* field is used to input the Ada code describing the semantics of the leaf process.

*3.2.2 Mapping Starting Point.* According to the SAGEN input requirements, a SAGEN source file can describe a platform, a process, a technology module, or a dynamic technology module.

Since an SADMT architectural description simulates the interaction between a configuration of platforms moving in space, this interface will support SADMT by creating individual SAGEN source files for platform descriptions. These platform descriptions can then be translated into SADMT descriptions by executing the SAGEN translator. Thus, a configuration of platforms can then be arrived at by invoking this interface once for each platform in the SADMT architectural description being modeled.

Therefore, the mapping starting point for this research effort is the platform level. This conclusion suggests that the interface from SAtool will always begin with "Provide <*platform_name*> Platform" at the A–0 level.

*3.2.3 Mapping at A–0 Level.* The single activity at the A–0 level in an $IDEF_0$ model describes in general what is being analyzed. As stated above, the activity at the A–0 level for this interface will always be "Provide <*platform_name*> Platform".

According to the description of SADMT, the only inputs to a platform are the PIG process inputs. The SADMT/SF informs a platform that it has received a cone

through the *beaming* port on its PIG process. The SADMT/SF informs a platform of certain events through its *event* port on its PIG process. Finally, SADMT/SF informs a platform when it collides with another platform through its *collision* port on its PIG process. The only outputs from a platform are cones to other platforms (see Section 2.2.4).

Therefore, the user of this interface is limited to A–0 level inputs of data type *beaming*, *event*, or *collision* as shown in Figure 2.3. Similarly, the user is limited to A–0 outputs of data type *cone*. Each individual output cone or received beaming, event, or collision must be explicitly represented by a unique arrow on the A–0 drawing.

The SAGEN syntax requirements shown in Appendix B show that a string is needed to be used as a platform designator. This interface limits that string to 2 characters (based on the platform examples provided by IDA). This string should be indicated at the A–0 level by a *mechanism* arrow.

Table 3.1 presents the SAGEN features for a platform and the corresponding $IDEF_0$ constructs that are used to model them at the A–0 Level.

Figure 3.9 illustrates an example of an SAtool diagram of the A-0 level.

*3.2.4 Mapping at A0 Level.* The mapping at the A0 level describes the internal view of a platform shown in Figures 2.3 and 2.4 and how that view is represented in $IDEF_0$. Therefore, this section describes the mapping to the PIG process, the SADMT processes, the technology modules, and the dynamic technology modules.

Since every platform has one PIG process, and the inputs to the PIG process from the SADMT/SF are the same as the PIG process outputs, the PIG process is hidden from the user in this interface. As long as the platform inputs are shown correctly at the A–0 level, the interface will correctly model the SAGEN file.

The description of the SADMT process mapping is the same as the mapping at all other levels, which is presented in Section 3.2.5. The technology modules and

| *SAGEN Features* | *IDEF$_0$ Constructs* |
|---|---|
| Platform | Activity |
| Beaming | Input boundary arrow (of data type *beaming*) |
| Event | Input boundary arrow (of data type *event*) |
| Collision | Input boundary arrow (of data type *collision*) |
| Cone(s) | Output boundary arrow (of data type *cone*) |
| Platform Designator | Mechanism arrow |
| Platform name | *ALIAS* field |
| Platform description | *DESCRIPTION* field |

Table 3.1. Mapping at A–0 Level

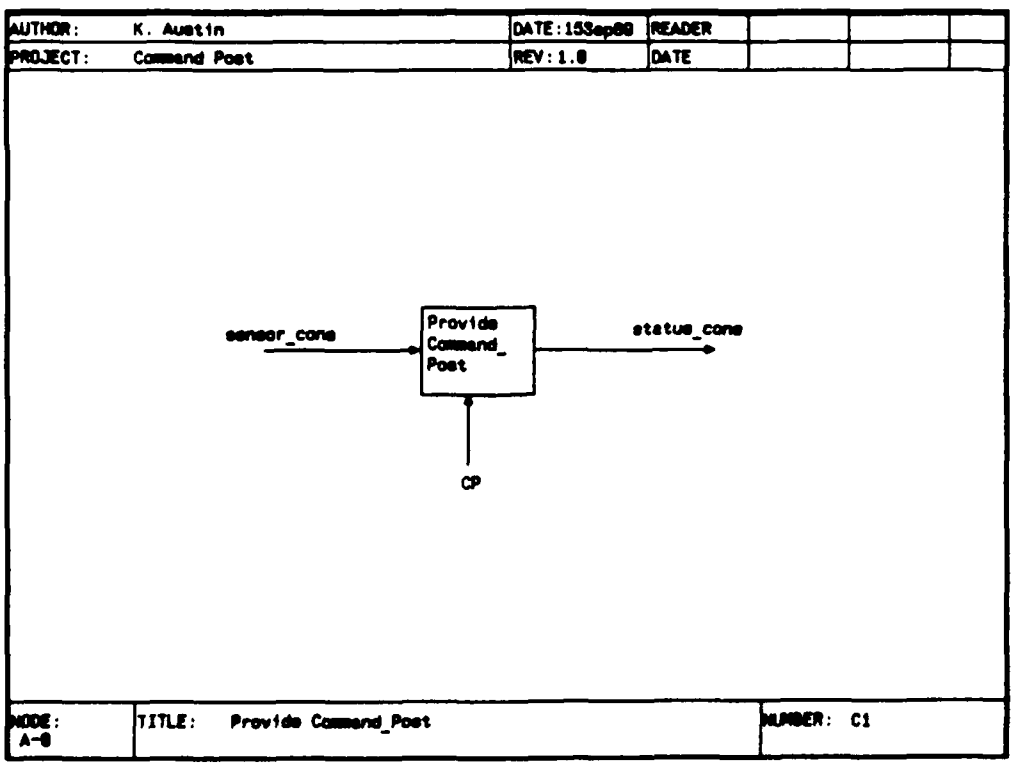


Figure 3.9. Sample of A–0 Diagram

the dynamic technology modules are discussed in the following subsections.

Table 3.2 presents the SAGEN features for a platform and the corresponding $IDEF_0$ constructs that are used to model them at the A0 Level. Figure 3.10 illustrates an example of an SAtool diagram of the A0 level.

*3.2.4.1 Technology Modules.* As previously discussed in Section 2.2.4, all cone inputs/outputs to/from a platform must be processed by a technology module. The cones enter/exit the platform via the PIG process which is the platform's connection to the SADMT/SF.

Only a technology module can emit/receive a cone; therefore, each boundary arrow at the A0 level must be an input to/output from a technology module. Since $IDEF_0$ does not have ports with port types, the data type of the boundary arrow becomes the port type. Therefore, input boundary arrows must be of type *beaming*, *event*, or *collision*. As shown in Table 3.2 and Figure 3.10, a technology module is represented at the A0 level by an activity with a boundary arrow and a mechanism. In this case, the mechanism arrow indicates that the technology module is represented elsewhere.

One of the main ideas behind technology modules is reusability (11:33). This interface assumes the existence of a library of technology modules, and that a particular technology module can be invoked by the mechanism call at the A0 level. Since the mechanism is calling a library technology module, which by definition is represented elsewhere, the decomposition of the activity representing the technology module ceases at the A0 level. A listing of the available technology modules provided by IDA is shown in Appendix D. This listing is necessary since the user of this interface must have prior knowledge of the available technology modules when designing a platform description. If the library does not contain the needed technology module, the user will have to create his own.

| *SAGEN Features* | *IDEF$_0$ Constructs* |
|---|---|
| PIG Process | None(Implicit in model) |
| SADMT Process(es) | See Table 3.3 |
| Technology Module(s) | Activity with boundary arrow and mechanism |
| Dynamic TM(s) to be excluded | Activity with no inputs, no outputs, and a mechanism |
| Cone(s) | Output boundary arrows (of data type *cone*) |
| Process names | *ALIAS* field |
| Process description | *DESCRIPTION* field |

Table 3.2. Mapping at A0 Level

*3.2.4.2 Dynamic Technology Modules.* Dynamic technology modules are different than the technology modules discussed in Section 2.2.4. The simulation will automatically place any declared dynamic technology modules on each platform in a simulation unless the user specifically excludes it from a given platform. As shown in Table 3.2, an excluded dynamic technology module is represented by an activity with no inputs, no outputs, and a mechanism. If a particular dynamic technology module is not specifically excluded at the A0 level, then the assumption is that the user wants it placed on the platform.

*3.2.5 Mapping at all other Levels.* The mapping at all other levels essentially describes the SADMT Process Model (Figure 3.8) and how that model is represented by IDEF$_0$. Table 3.3 presents the SAGEN features for a SADMT Process and the corresponding IDEF$_0$ constructs that are used to model them at all other levels of decomposition.

The SADMT Processes map to IDEF$_0$ activities. The messages passed between processes map to the data elements connecting the IDEF$_0$ activities. The data type of the data elements becomes the data type of the port that the message is passed

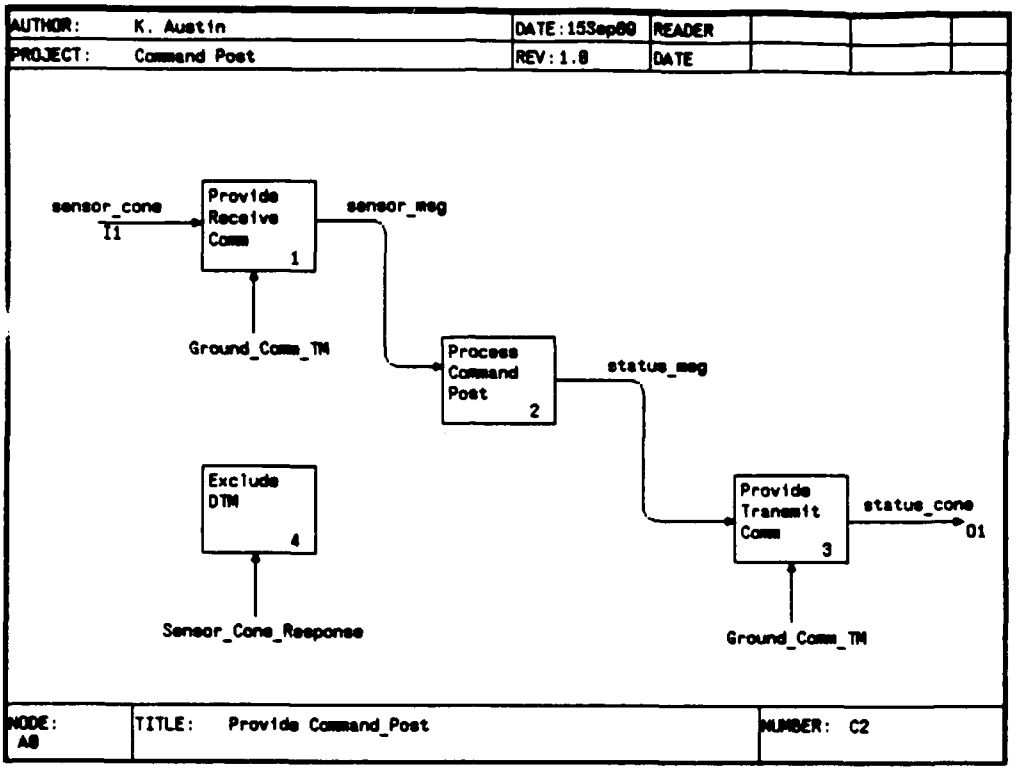


Figure 3.10. Sample of A0 Diagram

| *SAGEN Features* | *IDEF$_0$ Constructs* |
|---|---|
| Processes | Activities without mechanisms |
| Message | Data Element |
| Port | Combination of Data Element's Source, Destination, and Data Type |
| Link | Source and Destination for Data Element |
| Ada clauses | Control arrow |
| Process names | *ALIAS* field |
| Process description | *DESCRIPTION* field |
| Process Semantics (Ada pseudocode) | *DESCRIPTION* field (after Process Description) |

Table 3.3. Mapping at all other Levels

through. The direction of the port depends on whether it is the data element source or destination. The source would be an outport, while the destination would be an inport. This mapping presently does not support selectable ports. By definition a selectable port enables the user to declare what ports a message will flow through. $IDEF_0$ has no way of indicating controlled flow through certain data elements. If a data element is connected to three activities, then all three activities will receive that element.

The links of the SADMT process model map to the source and destination of a data element. An input-inherited link maps to a boundary arrow without a source, while an output-inherited link maps to a boundary arrow without a destination. The type of the link maps to the type of the data element. Since the data element's type is used as the type for both the links and the ports that it maps to, the mapping ensures that a link to a port has the same type as the port.

The SAGEN syntax requirements shown in Appendix B show that an SADMT process description may begin with Ada context (with) or visibility (use) clauses. These clauses will be represented by control arrows in this mapping. This mapping will create a "*with package_name;*" and a "*use package_name;*" for each control on a given activity describing an SADMT process. If there are no control arrows, this mapping assumes that the SADMT process represented by this activity does not have any Ada context clauses.

Finally, the process semantics are represented by an Ada task at the 'leaf' nodes in the SADMT Process Model. The 'leaf' node maps to an $IDEF_0$ activity that has no children. The semantics in Ada pseudocode are described in the *DESCRIPTION* field of the activity after the process description. The format for that entry is shown in Appendix C.

Figure 3.11 illustrates an example of a SAtool diagram below the A0 level.

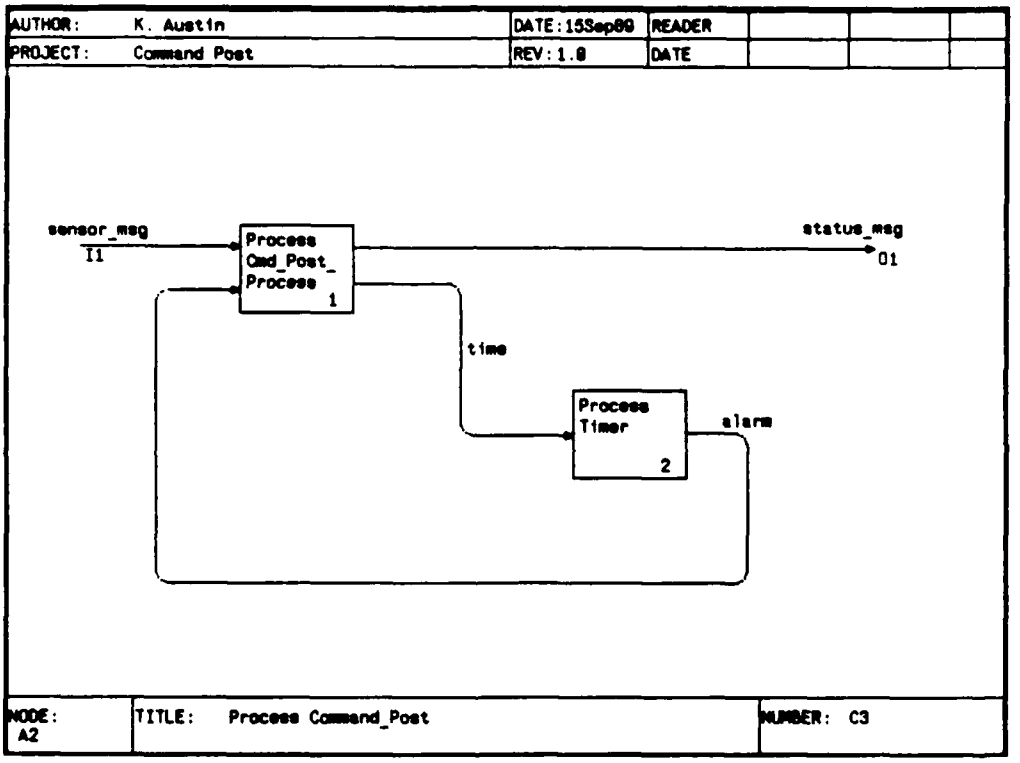


Figure 3.11. Sample Diagram below A0 level

### *3.3 Summary*

This chapter has presented E-R models for $IDEF_0$ and SADMT, and a mapping between them. The mapping provides a basis for the development of a software interface between the two packages.

The E-R model for $IDEF_0$ was based on a premise that a given $IDEF_0$ diagram is actually a graphic representation of a more fundamental underlying model, which can be represented by any number of diagrams. This underlying model has two main entities, *activities* and *data elements.* An E-R diagram can be used to illustrate these entities and the relationships that exist between them.

The E-R model for SADMT actually has three different views. The three views are the: (1) *External View of a Platform*, (2) the *Internal View of a Platform*, and (3) the *SADMT Process Model.* The external view represents an external look at the platform moving in space. The internal view represents an internal look at the SADMT processes used to further describe the platform. Finally, the SADMT process model represents the intercommunicating subprocesses used to represent each process.

The mapping development was divided into several sections. First, the mapping begins at the A–0 level, and always begins with the activity "Provide <*platform_name*> Platform". The mapping at the A0 level represents the internal view of the platform. Each of the activities represent SADMT processes which further describe the platform. Finally, all subsequent levels of decomposition represent the hierarchy of communicating subprocesses in the SADMT process model.

# *IV. Detailed Design*

The previous chapter presented the preliminary design of the interface between SAtool and SADMT. The purpose of this chapter is to discuss the detailed design decisions that were implemented in the prototype software interface. The following sections will describe:

- the overall approach to the software implementation,
- the portions of Smith's version of SAtool II (18) that are used, and
- the prototype software developed for this research effort.

## *4.1 Overall Approach*

This section describes the overall approach that was used to implement the prototype version of this software interface between SAtool and SAGEN. The approach taken was arrived at after reviewing the purpose of this effort, which was to investigate the usability of SAtool as a front end to SADMT. To do this, the interface was to operate on SAtool output files and generate SAGEN input files.

Having reviewed the purpose of this effort, a combination of a top-down and bottom-up development methodology was used to develop the prototype software. The reason a combination was used is:

1. The prototype software was to be independent of both SAtool and SADMT (SAGEN). Therefore, the main part of the program was developed top-down.
2. In designing the interface, many of the submodules needed to get information from the common database were already implemented by Smith's version of SAtool. To avoid 'reinventing the wheel', every effort was made to use as much of the existing software as possible. Since the software reused was in

submodules of Smith's version of SAtool, reusing them presented a bottom-up approach.

## *4.2 Description of Smith's SAtool*

This section describes the portions of Smith's software that are reused in developing this interface. For a detailed description of Smith's version of SAtool, refer to (18).

As discussed in Section 1.2.1, Smith's version of SAtool, which is implemented in Ada, can load, edit, and store multiple levels of $IDEF_0$ diagrams. Smith accomplishes this by creating two linked lists to store all the data dictionary information shown in Figures 3.2 and 3.3. One linked list stores all the activities created during an editing session, and the other stores all the data elements created. To create these lists, Smith instantiates a *generic* linked list presented in (3:79). The only difference between the two lists is the item that they store.

The item that the data activity list stores is a record of all the entries in the AFIT data activity data dictionary (Figure 3.2). Smith defines this record as:

```
type ACTIVITY_RECORD is
   record
      Name             :   db_Types.ACT_NAME_TYPE;
      Record_Type      :   db_Types.ENTITY_RECORD_TYPE;
      Project          :   db_Types.PROJECT_NAME_TYPE;
      Node_Num         :   db_Types.NODE_NUM_TYPE;
      Act_Description  :   db_Types.Text_List.List;
      Inputs           :   ACT_ARROW_TYPE;
      Outputs          :   ACT_ARROW_TYPE;
      Controls         :   ACT_ARROW_TYPE;
      Mechs            :   ACT_ARROW_TYPE;
      Aliases          :   ALIAS_TYPE;
```

```
        Parent            :   db_Types.ACT_NAME_TYPE;
        Req_Num           :   db_Types.REQ_NUM_TYPE;
        Version           :   db_Types.VERSION_TYPE;
        Version_Comment   :   db_Types.Text_List.List;
        Date              :   db_Types.DATE_TYPE;
        Author            :   db_Types.AUTHOR_TYPE;
        Status            :   db_Types.ENTITY_STATUS_TYPE
                                := db_Types.Unchanged;
    end record;
```

The item that the data element list stores is a record of all the entries in the AFIT data element data dictionary (Figure 3.3). Smith defines this record as:

```
type DATA_ITEM_RECORD is
   record
      Name              :     db_Types.DI_NAME_TYPE;
      Record_Type       :     db_Types.ENTITY_RECORD_TYPE;
      Project           :     db_Types.PROJECT_NAME_TYPE;
      DI_Description    :     db_Types.Text_List.List;
      Data_Kind         :     db_Types.DATA_TYPE;
      Min_Value         :     db_Types.VALUE_TYPE;
      Max_Value         :     db_Types.VALUE_TYPE;
      DI_Range          :     db_Types.VALUE_TYPE;
      Values            :     db_Types.VALUE_TYPE;
      Part_Of           :     PIPE_CONTENTS_TYPE;
      Composition       :     PIPE_CONTENTS_TYPE;
      Aliases           :     ALIAS_TYPE;
      Sources           :     ACT_ASSOCIATION_TYPE;
      Destinations      :     ACT_ASSOCIATION_TYPE;
      Req_Num           :     db_Types.REQ_NUM_TYPE;
```

```
        Version          :    db_Types.VERSION_TYPE;
        Version_Comment  :    db_Types.Text_List.List;
        Date             :    db_Types.DATE_TYPE;
        Author           :    db_Types.AUTHOR_TYPE;
        Status           :    db_Types.ENTITY_STATUS_TYPE;
    end record;
```

Smith defines various procedures and functions to add, delete, or edit each of the fields for both of these records. He also defines procedures to store all the information in *filename*.**dbs** and then retrieve it.

These are the same procedures that this interface 'reuses'. In developing the submodules for this effort, as much as possible of Smith's software was used first. Where a procedure or module was needed that was not defined, that software was implemented in this effort using the **separate** compilation ability of Ada (6).

### *4.3 Prototype Software Interface*

The purpose of this section is to describe the prototype software interface that was developed for this research effort. The approach to this discussion will be to describe the main driver and the major submodules that were developed to create each of the entries in the SAGEN syntax requirements shown in Appendix B.

The main driver, package *CREATE_SAGEN*, begins by inputting the *filename*.**dbs** file created by SAtool. Smith's procedure *dbs_loader* is used to accomplish this. This procedure creates the two linked lists discussed in Section 4.2. The driver creates the SAGEN source file *filename*.**sagen** where *filename* matches SAtool's.

A close review of the SAGEN source file syntax requirements and example platform descriptions listed in Appendix B shows that the SAGEN source file is a sequential file. Specifically, each of the platform's SADMT processes are decomposed down to their leaf node(s) before the next SADMT process is decomposed. Therefore,

to generate the sequential SAGEN source file, a procedure (*ORDER_THE_LIST*) was written to order all the activities in Smith's activity linked list such that the first activity was A–0, the second was A0, the third would be A1, then A11 if it exists, and so on.

Placing the list in order allows easy generation of the SAGEN source file by simply using a pointer to move through the list from top to bottom. The SAGEN specifications can then be generated for each SADMT process in the list. The sections below describe the submodules that were developed for each of the entries in the SAGEN specification.

Whenever a list of items are needed for one of SAGEN's specifications, the list is created by instantiating the same Booch software component mentioned above (3:79). By using a linked list there is no bound on the number of items in the SAGEN specification.

*4.3.1 Process Hierarchy, Ports, and Data Types.* The "[{*with_or_use*}]" entry in the SAGEN syntax is used for Ada context clauses (with) and/or visibility clauses (use) placed at the beginning of SADMT process descriptions. These clauses are represented by control arrows on the activities representing the SADMT processes. A function (*GET_CONTROLS*) was developed that creates a linked list of controls for the given process. This interface assumes that all context clauses will also be visibility clauses.

The next entry will be "*$platform* <*name*> := <*alias*> *is*" for the first entry in the activity linked list, and will be "*$process* <*name*> *is*" for all other entries. In each case, the <name> part of the statement is filled by the *ALIAS* field in the activity record.

The "*$subprocess*[*es*]..."entry in the SAGEN syntax is the listing of subprocesses for the current parent process. To create this list of subprocesses, a procedure (*FIND_SUBPROCESS*) was developed that searches the activity list for all processes

who have the given parent process. These subprocesses are stored in a linked list, so that there is no bound on the number of subprocesses a parent process can have.

The "*$[selectable_]...*" entry in the SAGEN syntax is a listing of the *inports* and *outports* for the given process. Two separate procedures were developed that create a linked list of inports and a linked list of outports respectively. The inport list is created by searching through the data element linked list for all the inputs in the *INPUTS* field of of the activity record and finding each input's data type. The port is then named "IN_*data element name*" where *data element name* is the name of the input, and has the same type as the data element. The outport list is created the same way using the *OUTPUTS* field. This interface does not support selectable ports.

The "*${cone|event|platform}_inport...*" entry in the SAGEN syntax is used to describe the various inports on a dynamic technology module. This interface creates platform descriptions, so this entry was not needed.

The "*$parameter...*" entry is presently only used to provide a platform designator for the platform. Since the designator is indicated by a mechanism arrow at the A-0 level, a function (*GET_MECHS*) was developed that creates a linked list of mechanisms for the given process. Any other uses of the "*$parameter*" entry is currently beyond the scope of this effort.

The "*$subdata...*" entry is the listing of all the data types that flow between the subprocesses of the given process. To create this list of subdata types, a procedure (*FIND_SUBDATA*) was developed that searches the activity list for all processes who have the given parent process. These subprocesses are stored in a linked list, so that there is no bound on the number of subprocesses a parent process can have.

The "*$cone[s]...*" entry is used to specify the data types for cones. Cones are only generated by technology modules. This interface creates platform descriptions, so this entry was not needed.

*4.3.2 Port Linkages.* The "[{*with_or_use*}]" entry in the SAGEN syntax for specifying port linkages is used to declare Ada context clauses (with) and/or visibility clauses (use). The example platform specifications provided by IDA show that any time a dynamic technology module is excluded, the port linkages specification "*withs*" the package for the dynamic technology module that is created by the SAGEN translator. The name of the package is obtained by appending "*_pkg*" to the name of the excluded dynamic technology module.

The "*$link*[*s*]..." statement is used to declare the links between subprocesses of a given process. The <name> variable is the name of the given process. There are two types of links, internal links, and inherited links. Two procedures (*FIND_PIG_LINKS* and *FIND_INTERNAL_LINKS*) were developed to create the internal links list. Procedure *FIND_PIG_LINKS* is only used at the platform level since it creates the internal links with the PIG process. Procedure *FIND_INTERNAL_LINKS* is used at all levels to create the internal links between subprocesses of a parent process. Finally, procedure *FIND_INHERITED_LINKS* was developed to create a linked list of inherited links.

*4.3.3 Port Semantics.* This final entry in the SAGEN syntax creates a task for a leaf node subprocess. This interface assumes that at the leaf nodes the user will have simplified the task to such a level that it can be easily written. Therefore, all of the Ada code for the task is placed in the activity description of the leaf SADMT process for which this task is created. The code follows the process description. See Appendix C for more information concerning the process semantics.

## *4.4 Summary*

This chapter has presented the detailed design decisions that were implemented in the prototype software interface. Specifically, this chapter has presented the overall approach used in developing the software, described the portions of Smith's code that

were reused, and finally discussed the prototype software that was implemented.

The overall approach to the software design was a combination of a top-down and a bottom-up approach. Since the software interface was independent of both SAtool and SAGEN, the main driver was developed top-down. Since many of the submodules needed to get information from the common database were already implemented, reusing them presented a bottom-up approach.

In developing his version of SAtool, Smith defines two linked lists to store all the data dictionary information for activities and data elements. He also defines many procedures to load, edit, and store each of these data dictionary entries. This interface reuses these procedures to retrieve information necessary to create SAGEN specifications.

Finally, the prototype software interface is discussed. The main driver reads in the *filename*.**dbs** created by SAtool and creates *filename*.**sagen** to be translated by the SAGEN translator. Each of the SAGEN specification requirements are created by individual submodules.

# V. Testing Approach

The previous chapter presented the detailed design decisions that were implemented in the prototype software interface. The purpose of this chapter is to discuss the testing approach that was used to validate and verify the prototype software interface.

## 5.1 General

The purpose of the designed software interface was to develop SAGEN source specifications from SAtool output files. Therefore, to test the validity of the software and to verify its correct operation, a test must generate a SAGEN specification and the specification should be compilable using the SAGEN translator provided by IDA. A final test of the specification would be to run the resulting platform description through an SADMT simulation. However, since that would require a configuration of platf - as well as a main driver, it was beyond the scope of this effort.

## 5.2 Methodology

The methodology used for this effort was to test the generated output against a known input. Specifically, several example platform descriptions were provided by IDA (1).

To ensure proper operation of the software interface, an SAtool analysis was performed on one of these platform descriptions. The resulting SAtool output file was run through the software interface, and then the generated SAGEN specification was compared to the original example provided by IDA. After the SAGEN specification was generated by the software interface, the specification was translated using the SAGEN translator. If the specification is correct, the translator will create several new files that contain the generated SADMT code. According to (9:10), the names of the new files will be:

| | |
|---|---|
| *filename*.a | for package specifications |
| *filename*_body.a | for package bodies |
| *filename*_link.a | for port linkages |
| *filename*_task.a | for process semantics |

If the generated specification is not correct, the SAGEN translator will create an error message or warning message.

This reverse engineering of a known platform description provides a measure of validity for the software interface while also providing a functional test of the entire system.

### *5.3 Test Description and Results*

The methodology described above was applied to the Command Post Platform description provided by IDA (1:20-25). An SAtool analysis as described in Appendix C was performed on the platform description and that analysis was stored. The interface was invoked creating a SAGEN specification of the platform description.

The SAGEN specification created by the software interface is shown in Appendix B after the original SAGEN specification provided by IDA. The only noticeable differences between the two are the naming conventions used for ports. The software interface automatically places an "IN_" before inport names and an "OUT_" before outport names.

The next step in the testing process was to run this SAGEN specification through the SAGEN translator. This step was successfully accomplished without generating any error or warning messages. The SAGEN translator created the four files shown in Section 5.2.

## *5.4 Summary*

This chapter has presented a methodology to test the prototype software interface. Testing was accomplished by performing an SAtool analysis on a known platform description provided by IDA. The SAtool analysis was run through the software interface. The resulting SAGEN specification was translated to SADMT using the SAGEN translator without generating any error or warning messages.

# *VI. Conclusions and Recommendations*

The purpose of this research effort was to investigate the usability of SAtool as a graphical front end to SADMT. To accomplish this task, a mapping between SAtool and SADMT was developed. Once the mapping development was complete, a prototype software interface was developed. The following sections present a summary, conclusions, and recommendations for further research.

## *6.1 Summary*

This research effort was successful in developing a one-way mapping from SAtool to SADMT. This mapping supports the development of SAGEN specifications for platforms using SAtool. The mapping does place constraints on its users as far as traditional $IDEF_0$ goes. A very specific template is placed on the user of this mapping when using SAtool to generate SADMT platform descriptions.

A secondary goal of this effort was to evaluate the full mapping which would enable the user to generate $IDEF_0$ from SADMT. Because of the specific template used in the one-way mapping, the full mapping was beyond the scope of this research and was not explored.

## *6.2 Conclusions*

- The mapping and subsequent interface developed in this effort indicate that SAtool can be used as a graphical front-end to SADMT, as long as a restricted subset of $IDEF_0$ defined in this interface is adhered to. The restricted subset redefines the control and mechanism data elements of $IDEF_0$.
- This redefinition was required because SADMT is a dataflow modeling technique. For this reason, a dataflow analysis computer-aided software engineering tool may better support the development of SADMT. However, a dataflow

analysis tool would not have the additional unused data elements that could be used to represent any other SAGEN feature.

- The E-R models that were developed for both $IDEF_0$ and SADMT proved to be invaluable in identifying relationships between both software packages. E-R diagrams permit graphic representation of the E-R model while maintaining the semantics of the underlying software.

### *6.3 Recommendations for Future Research*

- The interface should be offered as a user option on the finished version of SAtool II. The next thesis cycle is expected to complete the graphics portion of the tool. When this portion is completed, the SADMT interface should be a user option.
- The mapping development presently only enables the user to generate SAGEN specifications for platforms. Additional research efforts could investigate extending the mapping to support development of technology modules, dynamic technology modules, and ultimately to generate the main program which creates and initializes the platforms in a simulation.
- Smaller scale efforts could investigate extending the current mapping to support the $IDEF_0$ features that were not implemented in this prototype interface. Specifically, the mapping does not support all of the $IDEF_0$ 'pipe' constructs. SADMT is a strongly typed language. Therefore, only the pipe constructs that maintain data type were used in the prototype interface. It also does not support the SADMT selectable ports feature.
- A similar effort could investigate extending the current mapping to support the SADMT selectable port feature that was not implemented in this prototype interface. This feature was not implemented because there is no clear way in $IDEF_0$ to represent a selectable port. In $IDEF_0$, if an activity outputs a data element to three other activities, then each of the three activities will receive

that data element. There is no clear way to indicate that only one of the activities should receive the data element while the others do not.

# Appendix A. *E-R Model of IDEF$_0$*

This appendix presents a detailed description of the E-R Models for IDEF$_0$ presented in Section 3.1.1. Section A.1 describes the ACTIVITY Essential Data Model, and Section A.2 describes the DATA ELEMENT Essential Data Model.

## *A.1 ACTIVITY Essential Data Model*

This section presents a detailed description of each of the entities and relationships shown in Figure 3.4. Several of the entities and relationships also appear in Figure 3.5. For such cases, the entity or relationship is only defined once. The descriptions follow:

**activity** This entity represents the IDEF$_0$ (SADT) activities; as noted by the asterisk on the E-R diagram, *node number* is the key attribute, and *name* captures the name of the given activity. Attribute *description* allows the analyst to describe the activity.

**composed of** This relationship indicates that a given parent activity is composed of zero to many child activities. It also shows that each activity, except the A-0 activity, has exactly one parent activity.

**analyst** This entity is used to capture information about the analyst who performed the analysis. The entity **analyst** currently has the single attribute *author*, which identifies the person who performed the analysis.

**analyzes** This relationship indicates that an analyst analyzes zero to many activities (or data elements). The current model only allows an activity (or data element) to be analyzed by one analyst. Attribute *version* is used to record version information; *date* indicates when the analysis was performed; *changes* allows historical data about a given activity (or data element) to be captured.

**project** This entity identifies the project to which each activity (or data element) is assigned. Key attribute *project id* is the unique identifier for each project, and attribute *pname* indicates the name of the project.

**part of** This relationship indicates that an activity (or data element) is part of exactly one project, but a project contains one to many activities.

**ref** This entity captures any references associated with an activity (or data element). The key attribute *reference* identifies which reference is involved, and attribute *type* identifies the type of reference.

**based on** This relationship indicates that a given activity (or data element) is based on at least one but perhaps many references, and that a given reference is the basis for zero to many activities (or data elements).

**historical activity** This entity is primarily used as a convenience so that the database does not have to be loaded with analyses which were previously accomplished. The key attribute *project id* indicates which project contained the historical activity, and attribute *node number* identifies the specific activity within the project.

**calls** This relationship indicates that an activity can call from zero to many previously completed (historical) activities, and that a given historical activity is called by at least one but perhaps many activities.

**inputs** This relationship indicates that an activity can input zero to many data elements.

**outputs** This relationship shows that an activity must have at least one but can have many output data elements.

**is controlled by** This relationship shows that an activity must have at least one but can have many control data elements.

**is mechanized by** This relationship indicates that an activity can have zero to many mechanism data elements.

## *A.2 DATA ELEMENT Essential Data Model*

Figure 3.5 illustrates the essential data model for the $IDEF_0$ data elements. Each of the entities and relationships is explained in the descriptions that follow.

**data element** This entity represents the $IDEF_0$ data elements; as indicated by the asterisk on the E-R diagram, attribute *name* is the key. Attribute *data type* indicates the type of data; attribute *description* allows the analyst to describe the data element.

**ref** This entity is described in the previous section.

**based on** This relationship is described in the previous section.

**project** This entity is described in the previous section.

**part of** This relationship is described in the previous section.

**analyst** This entity is described in the previous section.

**analyzes** This relationship is described in the previous section.

**pipe** This entity is a specialized data element (as illustrated via the ISA construct on the E-R diagram). It has no additional attributes, but merely indicates that the data element is actually a pipe containing at least two other data elements.

**consists of** This relationship shows that a pipe consists of at least two data elements, and that a data element can be contained within at most one pipe.

**atomic data item** This entity is a specialized data element (as shown by the ISA construct on the E-R diagram) for capturing data elements that have atomic values, i.e., are not pipes. It has three attributes, *minimum* (minimum data value, if applicable), *maximum* (maximum data value, if applicable), and *range* (data value range, if applicable). In the case that none of the attributes are applicable, entity **values**, as described below, probably applies.

**values** This entity is used to accommodate atomic data items which have enumerated values, e.g., color can have values red, blue, and green. The entity has a single attribute *value*.

**can have** This relationship ties the atomic data item entity to its corresponding values entity (if it exists).

**alias** This entity captures any aliases that a given data element might have. The key attribute *name*, is the name of the alias, attribute *comment* is used by the analyst to clarify why the alias was needed, and attribute *where used* indicates where the alias is used.

**has an** This relationship shows that a data element can have zero to many aliases, and that a given alias corresponds to exactly one data element.

**inputs** This relationship is described in the previous section.

**outputs** This relationship is described in the previous section.

**is controlled by** This relationship is described in the previous section.

**is mechanized by** This relationship is described in the previous section.

# Appendix B. *SAGEN Syntax*

This appendix presents the correct SAGEN syntax for specifying the following:

1. Process hierarchy, ports, and data types,
2. Port linkages, and
3. Process Semantics.

The following sections are abstracted from (9). After the syntax is presented, several IDA example platform, technology module, and dynamic technology module descriptions are presented.

## *B.1 Notational Conventions*

The notational conventions used in the syntactic specification of SAGEN are defined as shown in Table B.1. Similarly, the variables used in the syntactic specification of SAGEN are defined as shown in Table B.2.

## *B.2 Process Hierarchy, Ports, and Data Types*

The syntax of SAGEN statements for specifying process hierarchy, ports, and data types is:

| *Notation* | *Convention* |
|---|---|
| <item> | a variable item |
| [item] | an optional item |
| {item1 \| item2} | item1 or item2 |
| {items}* | items repeated zero or more times |

Table B.1. Notational Conventions for SAGEN Syntactical Specification

| *Variable* | *Definition* |
|---|---|
| <alias> | a string of characters |
| <data_type> | a valid Ada data type |
| <declarations> | valid Ada declarations |
| <default> | a valid Ada expression |
| <discriminant> | a valid Ada discriminant |
| <name> | a valid Ada identifier |
| <param_list> | a list of parameters separated by commas |
| <range> | a valid Ada integer range |
| <with_or_use> | a set of valid Ada context and/or visibility clauses |

Table B.2. Variable Definitions for SAGEN Syntactical Specification

```
[{with_or_use}]
{$process <name> | $platform <name> := <alias> |
$tech[nology]_module <name> |
$dynamic_tech[nology]_module <name> := <alias> } is
    $subprocess[es] <name> [(range)][:=(<param_list>)]
        {,<name>[(<range>)][:=(<param_list>)]}*;
    $[selectable_][{data_|control_|mech_}]{inport[s]|outport[s]}
        <name> [(range)]:<data_type>[:=(<param_list>)]
        {,<name>[(<range>)]:<data_type>[:=(<param_list>)]}*;
    ${cone|event|platform}_inport [<name>];
    $parameter[s]<name>:<data_type>[(<discriminant>)]:=(<default>)
        {,<name>:<data_type>[(<discriminant>)]:=(<default>)}*;
    $subdata <data_type>{,<data_type>}*;
    $cone[s] <data_type>{,<data_type>}*;
$begin

  [Ada source lines (body)]

$end;
```

## *B.3 Port Linkages*

The syntax of SAGEN statements for specifying port linkages is:

```
[{with_or_use}]
```

```
$link[s] <name> is
  [<declarations>]
$begin

  [Ada source lines (link)]

$end;
```

### *B.4 Process Semantics*

The syntax of SAGEN statements for specifying process semantics is:

```
[{with_or_use}]
$task <name> is
  [<declarations>]
$begin

  [Ada source lines (task)]

$end;
```

### *B.5 Sample SAGEN Source Specification*

This section presents a sample SAGEN source specification for a platform. This sample was provided by IDA (1).

```
-- ------------------------------------------------------------------------
-- --              Command_Post Platform Description
-- ------------------------------------------------------------------------

--  DESCRIPTION
--
--     The Command Post (CP) enables the system (by sending an "at war"
--     message) once it hears about any targets. It disables the system
--     (by sending an "at peace" message) after hearing about zero
```

```
--      targets for more than an hour.
--      The CP sends one of these messages to all satellites every 10 seconds.
--     On a peace-to-war transition, a message is sent immediately.


-- PLATFORM SPECIFICATION

$platform Command_Post := Command_Post is

   $parameter  CP_id : string(1..7) := ("--CP---");

   $subprocess Command_Post_Processing:= (CP_id),
               Ground_Station_Communication_TM:= (CP_id);
               -- The PIG is a predefined subprocess of every platform

   $subdata    Order, Track_data, Cone_msg;

$end;

-- PORT LINKAGES

with Sensor_Cone_Response_TM_pkg;
$links Command_Post is

$begin
    -- Exclude the Radar Return TM
    exclude_dyn_module(MYSELF,
        Sensor_Cone_Response_TM_pkg.Sensor_Cone_Response_TM_designator);

    -- Connect Command_Post to Ground_Station_Communication_TM
    internal_link (Command_Post_Processing.Send_status_msg,
                   Ground_Station_Communication_TM.Order_xmit);
    internal_link (Ground_Station_Communication_TM.Sensor_data_rcv,
                   Command_Post_Processing.Recv_sensor_msg);
    -- Connect PIG to Ground_Station_Communication_TM
    internal_link (PIG.Beamings,
                   Ground_Station_Communication_TM.Cone_in);
$end;


-- PROCESS SEMANTICS
-- The semantics of this platform are defined within its subprocesses.
```

```
-- Older versions of Sagen required the $task, this one doesn't

------------------------------------------------------------------------
--                    Command_Post_Processing Process
------------------------------------------------------------------------

-- PROCESS SPECIFICATION

$process Command_Post_Processing is

   $parameter  CP_id : string(1..7) := ("--CP---");

   $inport     Recv_sensor_msg : Track_data;

   $outport    Send_status_msg : Order;

   $subprocess Processor:= (CP_id),
               Timer:= (CP_id);

   $subdata    Time, Boolean;

$end;

-- PORT LINKAGES

$links Command_Post_Processing is

$begin
   -- Connect Processor to parent (Command_Post_Processing)
   inherited_link (Recv_sensor_msg,
                   Processor.Recv_sensor_msg);
   inherited_link (Processor.Send_status_msg,
                   Send_status_msg);
   -- Connect Processor to Timer
   internal_link (Processor.Start_timer,
                  Timer.Start_timer);
   internal_link (Timer.Sound_alarm,
                  Processor.Timer_alarm);
$end;

-- TASK SEMANTICS
-- The semantics are defined in the subprocesses Processor and Timer.
```

```
-- Older versions of Sagen required the $task, this one doesn't

------------------------------------------------------------------------
--                        Processor Process
------------------------------------------------------------------------

-- PROCESS SPECIFICATION

$process Processor is

   $parameter  CP_id : string(1..7) := ("--CP---");

   $inport     Recv_sensor_msg : Track_data,
               Timer_alarm : Boolean;

   $outport    Send_status_msg : Order,
               Start_timer : Time;

$end;

-- PORT LINKAGES
-- There are no port linkages within this process
-- Older versions of Sagen required the $link, this one doesn't

-- TASK SEMANTICS

$task Processor is

   Status            : Order;
   Last_target_time  : PDL_time_type;
   Targets_detected  : Boolean;
   Sensor_info       : Track_data;

   sec               : constant := PDL_ticks_per_second;
   min               : constant := 60 * PDL_ticks_per_second;

$begin

   -- Start at peace
   Status.Initiator := CP_id;
   Status.Order := DEFCON7;
```

```
-- Send status to all satellites
Emit (Send_status_msg, Status);

-- *** DEMO MSGS ***
if Current_debug_level > 20 then
   Put (CP_id);
   Put (" broadcast status ");
   if Status.Order = DEFCON1 then
      Put ("war.");
   elsif Status.Order = DEFCON7 then
      Put ("peace.");
   end if;
   Put (" T=");
   Put (Current_PDL_time);
   New_line;
end if;
-- *** DEMO MSGS ***

-- Start the 10 second timer
Emit (Start_timer, 10*sec);

loop        ---- continuously
   -- *** DEBUG ***
   if current_debug_level > 110 then ---debug
      write_process_full(MYSELF,"*> ",
           " Top of wait-for-activity loop."); ---debug
   end if; ---debug
   -- *** DEBUG ***

   -- If no msgs are waiting, then
   if (Port_length (Recv_sensor_msg) = 0) AND
      (Port_length (Timer_alarm) = 0) then

      -- Wait 10 sec to send next status msg or
      -- until msg arrives from sensors
      Wait_for_activity;

   else

      -- If any msgs arrived from the sensors, then
      if Port_length (Recv_sensor_msg) > 0 then
```

```
-- Check to see if any sensors report targets
Targets_detected := FALSE;
for i in 1..Port_length (Recv_sensor_msg) loop
   Sensor_info := Port_data (Recv_sensor_msg);

   -- *** DEMO MSGS ***
   if Current_debug_level > 20 then
      Put (CP_id);
      Put (" recvd msg from sensor");
      Put (Sensor_info.Initiator);
      Put ("; ");
      Put (Sensor_info.Number_of_targets);
      Put (" targets detected.");
      Put (" T=");
      Put (Current_PDL_time);
      New_line;
   end if;
   -- *** DEMO MSGS ***

   Consume (Recv_sensor_msg);
   if Sensor_info.Number_of_targets > 0 then
      Targets_detected := TRUE;
   end if;
end loop;

-- If any sensors report targets detected, then
if Targets_detected then

   -- If status had been peace, then change to war
   if Status.Order = DEFCON7 then
      Status.Order := DEFCON1;

      -- Send status msg to all satellites
      Emit (Send_status_msg, Status);

      -- *** DEMO MSGS ***
      if Current_debug_level > 20 then
         Put (CP_id);
         Put (" broadcast status ");
         Put ("war.");
         Put (" T=");
         Put (Current_PDL_time);
```

```
            New_line;
         end if;
         -- *** DEMO MSGS ***

         -- Record the last time targets were seen
         Last_target_time := Current_PDL_time;
      end if;
   end if;
else
   -- Timer must have gone off
   -- *** DEBUG ***
   Break_ZZZ; ---debug
   -- *** DEBUG ***

   -- Remove the Timer msg from the queue
   Consume (Timer_alarm);

   -- Send status msg to all satellites
   Emit (Send_status_msg, Status);

   -- *** DEMO MSGS ***
   if Current_debug_level > 20 then
      Put (CP_id);
      Put (" broadcast status ");
      if Status.Order = DEFCON1 then
         Put ("war.");
      elsif Status.Order = DEFCON7 then
         Put ("peace.");
      end if;
      Put (" T=");
      Put (Current_PDL_time);
      New_line;
   end if;
   -- *** DEMO MSGS ***
end if;

-- If at war and no targets are detected for 1 hr, then
-- return to peaceful status.
if Status.Order = DEFCON1 then
   if Current_PDL_time > Last_target_time + 60*min then
      Status.Order := DEFCON7;
   end if;
```

```
         end if;
      end if;
   end loop;
   exception
when others => Put_line("**Some error in Processor_init**");
$end;

-------------------------------------------------------------------------
--                          Timer Process
-------------------------------------------------------------------------

-- PROCESS SPECIFICATION

$process Timer is

   $parameter  CP_id : string(1..7) := ("--CP---");

   $inport  Start_timer : Time;

   $outport Sound_alarm : Boolean;

$end;

-- PORT LINKAGES
-- There are no port linkages within this process
-- Older versions of Sagen required the $link, this one doesn't

-- TASK SEMANTICS

$task Timer is

   Interval : Time;
   Wake_up  : constant Boolean := TRUE;

$begin
   -- *** DEBUG ***
   if init_debug_level > 100 then ---debug
      write_process_full(MYSELF,"*> "," started up."); ---debug
   end if; ---debug
   -- *** DEBUG ***

   wait_for_activity;
```

```
   Interval := Port_data (Start_timer);
   consume (Start_timer);

   loop
      -- *** DEBUG ***
      if current_debug_level > 110 then ---debug
         write_process_full(MYSELF,"*> "," Top of wait loop."); ---debug
      end if; ---debug
      -- *** DEBUG ***

      wait (Interval);
      emit (Sound_alarm, Wake_up);

   end loop;
   exception ---debug
when others =>write_process_full(MYSELF,"**Some error in "); ---debug
$end;
```

### *B.6 Sample SAGEN Specification Generated by Interface*

This section presents a SAGEN source specification for the platform in the preceding section; however, this specification was generated by the software interface.

```
-- ********************************************************
--
--                  Command Post Platform
--
-- ********************************************************
--
-- This is the platform description block.  For the purpose
--
-- of this test, only short descriptions are used.  Also,
--
-- when describing process semantics in the task code, only
--
-- null descriptions are used.
--
```

```
-- PLATFORM SPECIFICATION

$platform Command_Post := Command_Post is

  $parameter CP_id:STRING(1..7):=("--CP---");

  $subprocess Ground_Communication_TM:=(CP_id),
              Command_Post_Processing:=(CP_id),
              Ground_Communication_TM:=(CP_id);
              -- The PIG is a predefined subprocess of every platform.

  $subdata beaming, Track_Data, Order, cone;

$end;

-- PORT LINKAGES

with Sensor_Cone_Response_TM_pkg;
$links Command_Post is

$begin

  -- Exclude unwanted dynamic technology modules.
  exclude_dyn_module(MYSELF,
    Sensor_Cone_Response_TM_pkg.Sensor_Cone_Response_TM_designator);

  -- Connect subprocesses.
  internal_link(Ground_Communication_TM.OUT_sensor_msg,
                Command_Post_Processing.IN_sensor_msg);
  internal_link(Command_Post_Processing.OUT_status_msg,
                Ground_Communication_TM.IN_status_msg);
  internal_link(PIG.Beamings,
```

```
                Ground_Communication_TM.Cone_in);

$end;

-- ***********************************************************
--
--                  Command Post Process Processing
--
-- ***********************************************************
--
-- This is the process description block.
--

-- PROCESS SPECIFICATION


$process Command_Post_Processing is

  $parameter CP_id:STRING(1..7):=("--CP---");

  $inport IN_sensor_msg:Track_Data;

  $outport OUT_status_msg:Order;

  $subprocess Timer:=(CP_id),
              Processor:=(CP_id);

  $subdata Order, Track_Data, Boolean, Time;

$end;
```

```
-- PORT LINKAGES
$links Command_Post_Processing is


$begin


  -- Connect parent process to its subprocesses.
  inherited_link(IN_sensor_msg,
                 Processor.IN_sensor_msg);
  inherited_link(Processor.OUT_status_msg,
                 OUT_status_msg);


  -- Connect subprocesses.
  internal_link(Timer.OUT_alarm,
                Processor.IN_alarm);
  internal_link(Processor.OUT_time,
                Timer.IN_time);


$end;


-- ********************************************************
--
--                  Processor Process
--
-- ********************************************************
--
-- This is the process description block.  This is a leaf
--
-- process, so the semantics will follow in a task.
--


-- PROCESS SPECIFICATION


$process Processor is
```

```
  $parameter CP_id:STRING(1..7):=("--CP---");

  $inport IN_sensor_msg:Track_Data,
          IN_alarm:Boolean;

  $outport OUT_time:Time,
           OUT_status_msg:Order;

$end;

$task Processor is

$begin

  null;

$end;

-- ********************************************************
--
--                    Timer Process
--
-- ********************************************************
--
-- This is the process description block.  This is a leaf
--
-- process, so the semantics follow in a task.
--

-- PROCESS SPECIFICATION
```

```
$process Timer is

  $parameter CP_id:STRING(1..7):=("--CP---");

  $inport IN_time:Time;

  $outport OUT_alarm:Boolean;

$end;


$task Timer is


$begin

  null;

$end;
```

# Appendix C. *User's Manual*

This appendix presents the User's Manual for the software developed in this research effort.

## *C.1 Introduction*

This manual explains how to use the Structured Analysis Tool (SAtool) Interface to the Strategic Defense Initiative Architecture Dataflow Modeling Technique (SADMT) (SAIS). This tool is used to create SADMT Generator (SAGEN) source files using SAtool as a graphical front-end. Specifically, the SAGEN source file is generated from the underlying data dictionary entries stored by SAtool. Since the data dictionary entries will be used to generate SAGEN source files, all entries should not violate Ada syntax. This manual is not intended to be a tutorial in the general use of SAtool or SADMT. It is intended to be a tutorial on the specific use of SAtool to decompose platform descriptions. SAIS uses the output of SAtool (*filename*.dbs) to generate a source file (*filename*.**sagen**) for SAGEN. The resulting SAGEN file can be translated to SADMT using the SAGEN translator. When all the platforms in an arbitrary system are developed, the SADMT Simulation Framework (SADMT/SF) can simulate the system. **This interface is limited to creating individual platform descriptions.** It is assumed that the user is familiar with $IDEF_0$, the general use of SAtool, SADMT, and the syntax of the Ada programming language.

When used correctly, this interface will enable the knowledgeable SAtool user to analyze an arbitrary system, identify the individual platforms in the description, and create SAGEN specifications for SADMT platform descriptions that can be simulated on the SADMT/SF. All this will be done without having to generate the complex template required by SADMT. In essence, the interface (along with SAGEN), will create the template for you.

## *C.2 General*

There are several SAtool fields that are used for the same entry no matter what the level of decomposition is. The first is the *DESCRIPTION* field in the activity data dictionary. This field must be manually entered since it can not be derived from SAtool diagrams. This entry is used to provide a textual description of what the activity does. At the platform level, this is a platform description. At the process level, this is a process description. Finally, for leaf processes, the *DESCRIPTION* field is used to enter the process semantics in the form of an Ada task. These process semantics are entered after the textual description of the process. The textual description of a leaf process and the process semantics are separated by a single line entry. This entry, "<task code>," signifies the beginning of the process semantics. The description field has a maximum line length of 60 characters, but there is no limit on the number of lines for the description.

The next general entry is the *ALIAS* field for the activity data dictionary. The *ALIAS* field must also be manually entered since it can not be derived from SAtool diagrams. The *ALIAS* field for each activity should have one entry and that entry should be the Ada name for the platform/process that the activity is representing. Whatever name is in this field, it will be used as the variable name for the platform/process in the generated SAGEN specification. The maximum length for this entry is 25 characters.

The final general entry is the entry for all data elements. The name used for each data element, which can be derived from SAtool diagrams, must not violate Ada syntax. Each data element must have a unique name. The maximum length for each data element name is 25 characters. Additionally, the data type for each data element must be manually entered since this can not be derived from SAtool diagrams. The data type of data elements is needed because the data type is used to create the data types of ports and links in the SADMT description.

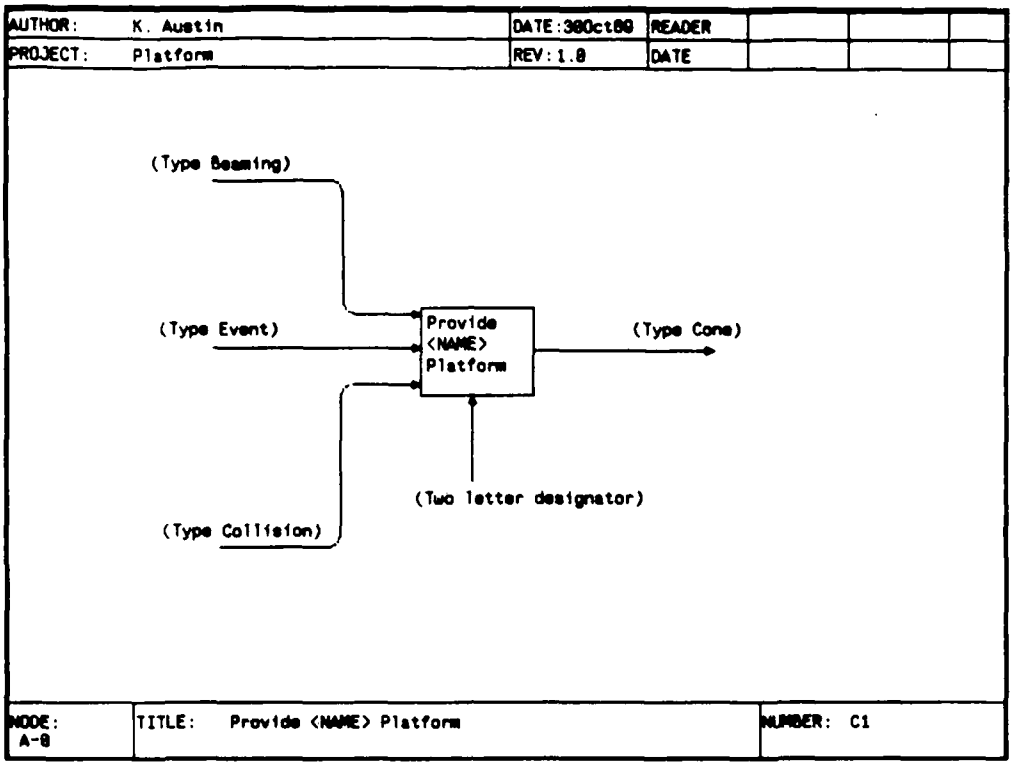


Figure C.1. Sample of A–0 Diagram

## *C.3 A–0 Level*

At the A–0 level, the analysis begins with an activity called "Provide <*platform_name*>" as shown in Figure C.1. The optional inputs at the A–0 level are limited to inputs of data type *beaming*, *event*, and/or *collision*. These are the three types of inputs that a platform may receive from the simulated environment via the PIG process. SAIS presently requires a separate data element for each of these inputs. The only branch constructs supported are those that maintain data element data type.

Each platform description requires a platform designator. This is represented by a mechanism arrow as shown in Figure C.1. The platform designator should be a unique two character string.

The optional outputs of the activity are of data type *cone*. Similar to the input data elements, the interface requires a separate data element for each output.

As mentioned previously, the textual description of the platform is manually entered into the *DESCRIPTION* field of the activity data dictionary. Similarly, the platform name is manually entered in the *ALIAS* field.

### *C.4 A0 Level*

The A0 level basically represents the internal view of a platform, whereby the platform is further described by processes. The SADMT processes are represented as activities at the A0 level.

The first type of SADMT process is a technology module. As shown in Figure C.2, a technology module is represented by an activity with a mechanism arrow. The mechanism arrow indicates that the technology module is represented elsewhere. This is a key point since this interface assumes a library of technology modules exists, and in that library the technology module is further decomposed. Therefore, when using this interface the technology module decomposition stops at the A0 level. One important note, all inputs and outputs to/from a technology module interact with the simulated environment. Therefore, all A–0 boundary arrows must be inputs or outputs from a technology module at the A0 level.

The second type of SADMT process is an SADMT logical process. This is represented at the A0 and all subsequent levels as an activity without mechanism arrows as shown in Figure C.2. The data element inputs and outputs for logical processes can come from other SADMT logical processes or from technology modules at the A0 level. A control arrow on an SADMT logical process indicates that the process needs visibility to an Ada package. The data element name of the control arrow will be the name used for the Ada package in the SAGEN specification. Like all other data element entries, the name field for the control arrow is 25 characters. Once again, this name must not violate Ada syntax.

The final type of SADMT process is a dynamic technology module. A dynamic technology module is a special type of technology module automatically placed on ev-

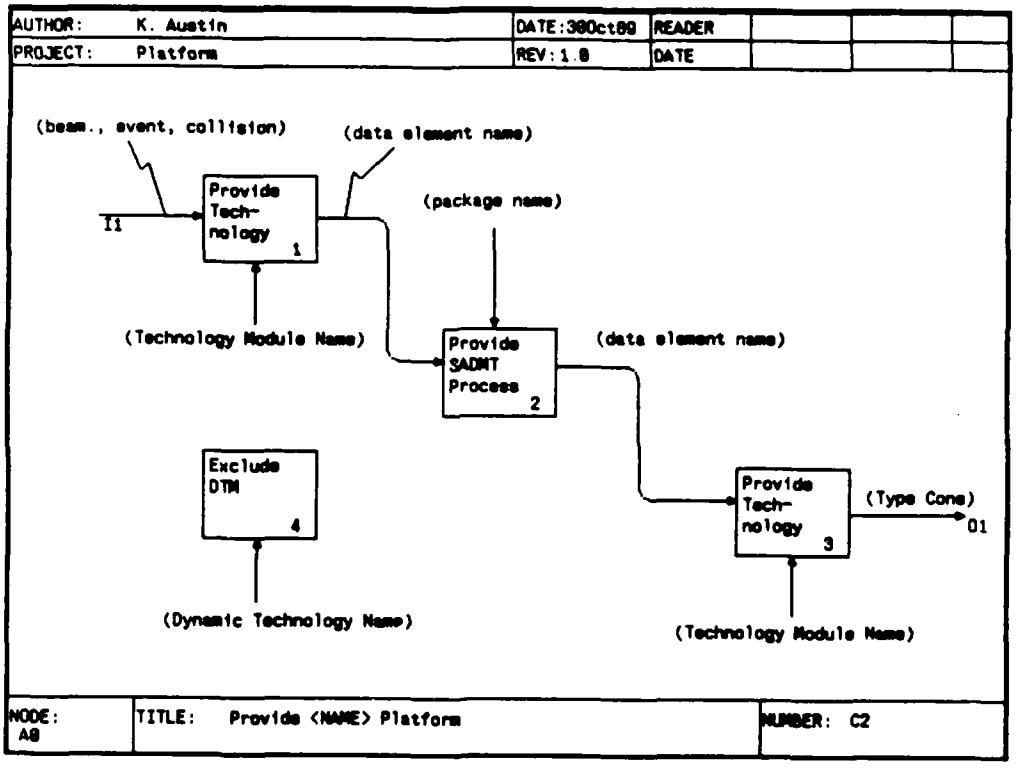


Figure C.2. Sample of A0 Diagram

ery platform in an architectural description unless explicitly excluded. SAIS enables the user to explicitly exclude dynamic technology modules. An excluded dynamic technology module is represented by an activity at the A0 level that has no inputs, no outputs, and a mechanism arrow (as shown in Figure C.2). The name of the data element representing the mechanism arrow will be used as the dynamic technology module name in the SAGEN specification.

As mentioned previously, the textual description of each process is manually entered into the *DESCRIPTION* field of the activity data dictionary. Similarly, the process name is manually entered in the *ALIAS* field.

### *C.5 Levels of Decomposition below A0*

At levels of decomposition below A0, all activities represent SADMT logical processes. The decomposition of SADMT processes now exactly matches a traditional SA analysis. When a satisfactory level of decomposition is achieved, the decomposition stops and the activities at that level are leaf processes. Figure C.3 illustrates two leaf processes. As mentioned previously, a leaf process contains the process semantics in the form of an Ada task. The code for the Ada task is manually entered into the *DESCRIPTION* field of the leaf process's activity data dictionary. The task code is entered after the textual process description is entered. The task code is delimited by a single line entry, "<task code>." Just like the A0 level, a control arrow on an SADMT logical process indicates that the process needs visibility to an Ada package. The data element name of the control arrow will be the name used for the Ada package in the SAGEN specification. An important note concerning Ada visibility, if a task representing a leaf nodes process semantics needs visibility to an Ada package, that visibility must be in the task code entered in the *DESCRIPTION* field.

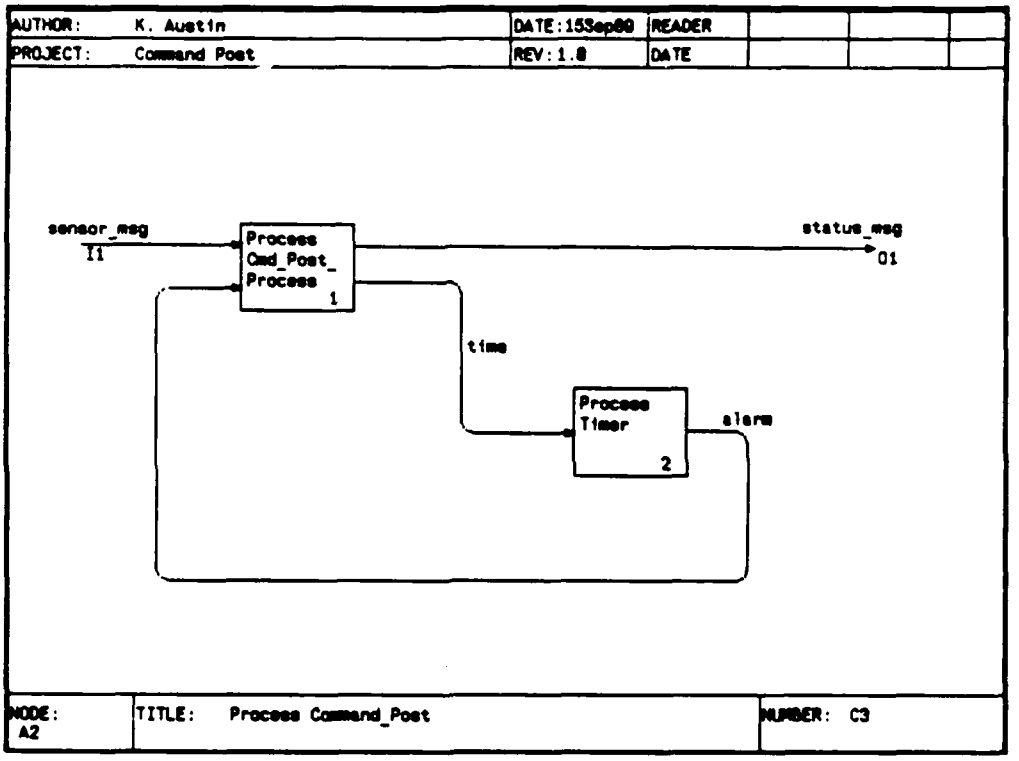


Figure C.3. Sample of Leaf Processes

### *C.6 Invoking SAIS and SAGEN*

Once the SAtool analysis is completed, the analysis should be stored in a file (the name of the file can be up to 8 characters long). This will result in the generation of *filename*.**dbs**. This is also the input file for SAIS. To invoke the interface, at the system prompt type:

SAIS

The software will prompt the user to enter an 8 character filename. The name of the .**dbs** file is entered. The interface will then create *filename*.**sagen**, which is the resulting SAGEN specification. This specification can then be translated to SADMT by the SAGEN translator.

The SAGEN translator is invoked much the same way. To invoke SAGEN, at the system prompt type:

SAGEN

The software will prompt the user to enter the SAGEN source file. The generated SAGEN source specification is entered, *filename*.**sagen**. If no errors or warnings are generated, the SAGEN translator will create the following files:

| | |
|---|---|
| *filename*.a | for package specifications |
| *filename*_body.a | for package bodies |
| *filename*_link.a | for port linkages |
| *filename*_task.a | for process semantics |

# Appendix D. *Technology Modules*

A description of the technology modules provided by IDA is listed below:

```
================================================================
Technology Module Descriptions:
================================================================
Ground_Station_Communication_TM
    This is the transmitter/receiver for use by ground station
    particles.  It can transmit commands of type Order (defined
    in Order_pkg), and receive sensor data in the form of
    Track_Data messages.
    Its ports are:
        wave_in     : wave messages in from the PIG (this should
                      be internal_link'ed to the PIG output port
                      wave_out by the particle's initialize).
        order_xmit  : input port which the BM/C3 of the ground
                      station should emit the order to be
                      transmitted into.
        sensor_data_rcv : output port which received sensor data
                          is emitted on.

Weapons_Platform_Communication_TM
    This is the receiver for use by weapons platform particles.
    It can receive commands of type Order (defined in Order_pkg),
    or receive sensor data in the form of Track_Data messages.
    Its ports are:
        wave_in     : wave messages in from the PIG (this should
```

be internal_link'ed to the PIG output port wave_out by the particle's initialize).

order_rcv : output port which received orders are emitted on.

sensor_data_rcv : output port which received sensor data is emitted on.

Sensor_Satellite_Communication_TM

This is the transmitter/receiver for use by sensor satellite particles. It can transmit its data of type Track_Data (defined in Track_Data_pkg), receive orders of type Order transmitted from a ground station, and receive sensor data in the form of Track_Data messages.

Its ports are:

wave_in : wave messages in from the PIG (this should be internal_link'ed to the PIG output port wave_out by the particle's initialize).

order_rcv : output port which received orders are emitted on.

sensor_data_xmit : input port which the BM/C3 of the satellite should emit its data on in order for it to be transmitted.

sensor_data_rcv : output port which received sensor data is emitted on.

KKV_Weapon_TM

This TM is a simple KKV weapon for use on the weapon platforms. It has a single input port of type Vector, and will compute a

trajectory and fire a KKV at a constant velocity of 5km/sec along this vector. The KKV will self-destruct when it reaches the end of this vector. The TM has only 12 KKV's, and will not fire any more. It is up to the BM/C3 to keep track of how many are left, however. It's input port is:

target_in : input port for vector to fire KKV along.

Particle_Collision_TM

This TM has a single input port which should be tied to the PIG. It waits for a collision with another particle to occur, and then destroys the particle containing it.

part_in : particle collision message in from the PIG (to be internal_link'ed to the PIG's output port part_out).

Russian_Missile_Launcher_TM

This is a TM used by the simulated Russian Missile Base to fire missile particles. It has a single input port of type Target (defined in Target_pkg), specifying the latitude and longitude of a target. It then creates a Russian_Missile_Particle with the appropriate equation of motion to strike the target.

target_in : target position information of type Target.

Russian_Missile_TM

This is a slightly more complicated version of the Particle_Collision_TM. It causes the particle's destruction if it strikes another particle, if it reaches the end of its equation of motion, or if it reaches the end of its expected

lifetime. It has two input ports:

part_in : particle collision input from the PIG (connected to the PIG's part_out output port).

event_in : event information from the PIG (connected to the PIG's event_out output port).

Sensor_Wave_Response_TM

This is a simple sensor wave reflector. It waits for an incoming sensor wave, and then sends out a reflection to the wave's initiator containing position information. It never communicates with the BM/C3. Its single input port is:

wave_in : input port for wave information, which should be connected to the PIG output port wave_out.

Sensor_Device_TM

This is a basic idealized sensor for used by the sensor satellites. When given a sensing command (of type Sense_Req, defined in the Sense_Req_pkg), it sends out a sensing wave, collects all of the responses, and builds a linked list of track records. It then returns a pointer (of type Track_Data_rec_ptr) to this list to the BM/C3:

wave_in : input port for wave information, which should be tied to the PIG output port wave_out.

sense_cmd : input port to receive sense commands from the BM/C3. Commands contain a vector for the axis of the sensing cone, and a half angle to specify its scope.

sensor_data : output port to return pointer to linked list of

Track_Data_rec's to the BM/C3.

# Appendix E. *Configuration Guide*

This appendix presents the configuration guide for the software developed in this research effort. The following configuration and order of compilation will ensure correct operation. The files marked with an * contain the packages from Smith's version of SAtool II that were reused in this interface.

ada -e generic_entity_list_spec.a

ada -e generic_entity_list_body.a

ada -e generic_list_utilites.a

ada -e ISDM_db_Types_Package.a*

ada -e sagen_types.a

ada -e ISDM_Activity_Package.a*

ada -e find_dtm.a

ada -e order_list.a

ada -e find_activity.a

ada -e find_subprocess.a

ada -e get_inputs.a

ada -e get_outputs.a

ada -e get_mechs.a

ada -e get_controls.a

ada -e get_description.a

ada -e get_alias.a

```
ada -e ISDM_Data_Item_Package.a*
ada -e find_type.a
ada -e find_subdata.a
ada -e find_inport.a
ada -e find_outport.a
ada -e ISDM_Time_Package.a*
ada -e ISDM_Project_Package.a*
ada -e ISDM_Data_Filer_Package.a*
ada -e sagen_utilities.a
ada -e find_internal_links.a
ada -e find_pig_links.a
ada -e find_inherited_links.a
ada -e length_of.a
ada -M create_sagen.a -o SAIS
```

This compilation will result in an executable file named **SAIS**.

# Appendix F. *Summary Paper*

This appendix presents a summary of this research effort.

## *F.1 Introduction*

According to Pressman, the ultimate goal for automation in software engineering would be for the computer–aided software engineering tool to cover the entire spectrum of software development (14:192). The user would enter the requirements for a certain problem, and the computer tool would analyze them and design the needed software.

The present status of computer–aided software engineering tools in the software engineering world is far from that ultimate goal. Tools are available to assist in individual parts of software development, but none cover the entire spectrum.

This paper describes the interfacing of two of these tools, Structured Analysis Tool (SAtool) and the Strategic Defense Initiative Architecture Dataflow Modeling Technique (SADMT).

The development of an interface between SAtool and SADMT brings the discipline of software engineering one step closer to the ultimate goal by allowing a model created with SAtool to be simulated by SADMT.

## *F.2 Background*

Developing an interface between two software packages requires a thorough understanding of each software package. The following sections provide an overview of the main features of each software package.

*F.2.1 SAtool.* In 1987, an Air Force Institute of Technology (AFIT) thesis by Steven E. Johnson (8) resulted in the development of SAtool, a computer–aided

software engineering tool based upon Structured Analysis (SA) (8). This tool automated two approaches for documenting software requirements analysis: SA diagrams and data dictionaries.

SAtool is a graphics editor which allows the analyst to draw SA diagrams and enter portions of the data dictionary for the requirements analysis phase of software development. The remaining elements of the data dictionary are automatically derived from the diagram.

The SAtool user can generate a printout of the SA diagram, a facing-page text printout, and a printout of the data dictionary. The analysis results can be saved in a standard data file for uploading into a common database, and the tool saves the graphical drawing information so the user can recall the diagram for editing.

*F.2.2 SADMT.* SADMT is a system developed for the Strategic Defense Initiative Organization (SDIO) by the Institute for Defense Analyses (IDA) as a method for describing system architectures. SADMT specifically supports Battle Management/$C^3$ and SDI architectures using Ada syntax. The main purpose of SADMT is to capture architectures in early development stages for simulation and evaluation purposes (11). The architectural descriptions use the standard syntax and semantics of Ada to capture the information needed to simulate the system through the SADMT Simulation Framework (SADMT/SF).

SADMT descriptions use a complex Ada template to simulate SDI architectures. Because of this complexity, the SADMT GENerator (SAGEN) translator was implemented to more easily facilitate SADMT descriptions. The translator accepts a simpler specification of the architecture being modeled and automatically generates the required SADMT template (9).

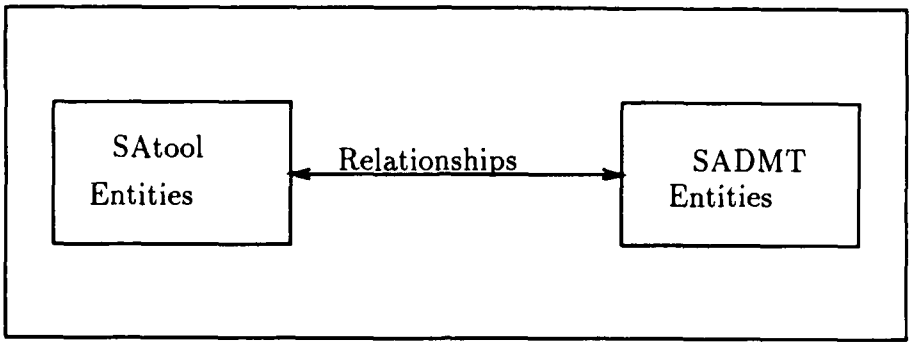


Figure F.1. Relationships between SAtool and SADMT

### *F.3 Preliminary Design*

In order to develop an interface between SAtool and SADMT, the underlying entities of each package were modeled to determine the relationships, if any, that exist between the two software packages. Figure F.1 illustrates this idea.

An entity-relationship (E-R) model for $IDEF_0$ (the underlying language of SAtool) and SADMT was developed. As shown in Figure 3.1, once the entities from both software packages were identified, the relationships between the two software packages were also identified and developed.

Based on the relationships identified in the E-R models described above and the SAGEN syntax specifications, the starting point for the mapping was determined to be at the platform level. Having made that decision, the remainder of the mapping process concentrated on finding an $IDEF_0$ graphical construct to represent a required SAGEN feature.

### *F.4 Detailed Design*

Once the mapping was developed, a prototype software interface was designed and implemented in the Ada programming language. The overall approach to the software design was arrived at after reviewing the purpose of this effort, which was to

investigate the usability of SAtool as a graphical front-end to SADMT. The interface was to operate on SAtool output files and generate SAGEN input files.

A combination of a top-down and bottom-up development methodology was used to develop the prototype software. A combination of methodologies was used because:

1. The prototype software was to be independent of both SAtool and SADMT (SAGEN). Therefore, the main part of the program was developed top-down.
2. In designing the interface, many of the submodules needed to get information from the common database were already implemented by SAtool. To avoid 'reinventing the wheel', every effort was made to use as much of the existing software as possible. Since the software reused was in submodules of SAtool, reusing them presented a bottom-up approach.

### *F.5 Testing Approach*

The methodology used for this effort was to test the generated output against a known input. Specifically, several example platform descriptions were provided by IDA (1).

To ensure proper operation of the software interface, an SAtool analysis was performed on one of these platform descriptions. The resulting SAtool output file was run through the software interface, and then the generated SAGEN specification was compared to the original example provided by IDA. After the SAGEN specification was generated by the software interface, the specification was translated using the SAGEN translator. If the specification is correct, the translator will create several new files that contain the generated SADMT code. If the generated specification is not correct, the SAGEN translator will create an error message or warning message.

This reverse engineering of a known platform description provided a measure of validity for the software interface while also providing a functional test of the entire system.

### *F.6 Test Description and Results*

The methodology described above was applied to the Command Post Platform description provided by IDA (1:20-25). An SAtool analysis was performed on the platform description and that analysis was stored. The interface was invoked creating a SAGEN specification of the platform description.

The only noticeable differences between the original description and the one created by the interface were the naming conventions used for ports. The software interface automatically places an "IN_" before inport names and an "OUT_" before outport names.

The final step in the testing process was to run this SAGEN specification through the SAGEN translator. This step was successfully accomplished without generating any error or warning messages.

### *F.7 Conclusions*

The purpose of this research effort was to investigate the usability of SAtool as a graphical front end to SADMT. This research effort was successful in developing a one-way mapping from SAtool to SADMT. This mapping supports the development of individual SAGEN specifications for platforms using SAtool. The mapping does place constraints on its users as far as traditional $IDEF_0$ goes. A very specific template is placed on the user of this mapping when using SAtool to generate SADMT platform descriptions.

Therefore, the mapping and subsequent interface developed in this effort indicate that SAtool can be used as a graphical front-end to SADMT, as long as a restricted subset of $IDEF_0$ defined in the interface is adhered to.

## *Bibliography*


1. Ardoin, Cy D., et al. *A Simple Example of an SADMT Architecture Specification*. Version 1.5, IDA Paper P-2036, Institute for Defense Analyses, April 1988.
2. Bachert, Robert F., et al. "SADT/SAINT Simulation Technique." In *National Aerospace Electronics Conference*, pages 4–9, IEEE Computer Society Press, 1981.
3. Booch, Grady. *Software Components with Ada*. Menlo Park, California: The Benjamin/Cummings Publishing Company, Inc., 1987.
4. Chen, Peter Pin-Shan. "The Entity-Relationship Model–Toward a Unified View of Data," *ACM Transactions on Database Systems*, *1*(1):9–36 (March 1976).
5. Connally, Ted D. *Common Database Interface for Heterogeneous Software Engineering Tools*. MS thesis, AFIT/GCS/ENG/87D-8, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1987 (AD-A189628).
6. Department of Defense, Washington, D.C. *Ada Programming Language*, February 1983. ANSI/MIL-STD-1815A-1983.
7. Hartrum, Thomas C. *System Development Documentation Guidelines and Standards (Draft #4)*. Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH, January 1989.
8. Johnson, Steven E. *A Graphics Editor for Structured Analysis with a Data Dictionary*. MS thesis, AFIT/GE/ENG/87D-28, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1987 (AD-A190618).
9. Kappel, Michael L., et al. *SAGEN User's Guide*. Version 1.5, IDA Paper P-2028, Institute for Defense Analyses, April 1988.
10. Korth, Henry F. and Silberschatz, Abraham. *Database System Concepts*. New York: McGraw-Hill Book Company, 1986.
11. Linn, Joseph L., et al. *Strategic Defense Initiative Architecture Dataflow Modeling Technique*. Version 1.5, IDA Paper P-2035, Institute for Defense Analyses, March 1988.
12. Materials Laboratory, Air Force Wright Aeronautical Laboratories, Air Force Systems Command, Wright-Patterson AFB, OH. *Integrated Computer-Aided Manufacturing (ICAM) Function Modeling Manual ($IDEF_0$)*, June 1981.
13. Morris, Gerald R. tentative title *A Comparison of a Relational and Non-Relational $IDEF_0$ Data Model*. MS thesis, School of Engineering, Air Force

Institute of Technology (AU), Wright-Patterson AFB, OH, anticipated March 1990.

14. Pressman, Roger S. *Software Engineering: A Practitioner's Approach* (Second Edition). New York: McGraw-Hill Book Company, 1987.
15. Ross, Douglas T. "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, *SE-3*(1):16–34 (January 1977).
16. Ross, Douglas T. "Applications and Extensions of SADT," *IEEE Computer*, *18*(4):25–34 (April 1985).
17. Ross, Douglas T. and Schoman, Kenneth E. Jr. "Structured Analysis for Requirements Definition," *IEEE Transactions on Software Engineering*, *SE-3*(1):6–15 (January 1977).
18. Smith, Nealon F. *SAtool II: An IDEF$_0$ Syntax Data Manipulator and Graphics Editor*. MS thesis, AFIT/GCE/ENG/89D-8, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1989.

# *Vita*

Captain Kenneth A. Austin

[redacted] He attended Valparaiso University from which he received his Bachelor of Science degree in Electrical Engineering in May, 1984. Upon graduation, he attended Officers Training School from which he received a commission in the USAF in August 1984. He served as Chief, Test and Evaluation Team at the $1815^{th}$ Operational Test and Evaluation Squadron, Wright–Patterson AFB, Ohio, from November 1984 to May 1988. In May 1988 he entered the School of Engineering, Air Force Institute of Technology to pursue a Master of Science degree in Computer Engineering.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

# REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION<br>UNCLASSIFIED | 1b. RESTRICTIVE MARKINGS |
| 2a. SECURITY CLASSIFICATION AUTHORITY | 3. DISTRIBUTION/AVAILABILITY OF REPORT<br>Approved for public release; distribution unlimited |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>AFIT/GCS/ENG/89D-1 | 5. MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a. NAME OF PERFORMING ORGANIZATION<br>School of Engineering<br>6b. OFFICE SYMBOL (If applicable)<br>AFIT/ENG | 7a. NAME OF MONITORING ORGANIZATION |
| 6c. ADDRESS (City, State, and ZIP Code)<br>Air Force Institute of Technology<br>Wright-Patterson AFB, Ohio 45433-6583 | 7b. ADDRESS (City, State, and ZIP Code) |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION<br>SDIO Phase I Program Office<br>8b. OFFICE SYMBOL (If applicable)<br>SDIO/S/PI | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER |
| 8c. ADDRESS (City, State, and ZIP Code)<br>Room 1E149, The Pentagon<br>Washington D.C. 20301-7100 | 10. SOURCE OF FUNDING NUMBERS<br>PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO. |

11. TITLE (Include Security Classification)
STRUCTURED ANALYSIS TOOL INTERFACE TO THE STRATEGIC DEFENSE INITIATIVE ARCHITECTURE DATAFLOW MODELING TECHNIQUE

12. PERSONAL AUTHOR(S)
Kenneth A. Austin, Capt, USAF

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Year, Month, Day) | 15. PAGE COUNT |
|---|---|---|---|
| MS Thesis | FROM _____ TO _____ | 1989 December | 112 |

16. SUPPLEMENTARY NOTATION

| 17. COSATI CODES | | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | |
| 12 | 05 | | Software Engineering, SADT, SADMT, CASE |
| | | | |

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Advisor: Dr. Thomas C. Hartrum
Associate Professor
Department of Electrical and Computer Engineering

| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT | 21. ABSTRACT SECURITY CLASSIFICATION |
|---|---|
| ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS | UNCLASSIFIED |

| 22a. NAME OF RESPONSIBLE INDIVIDUAL | 22b. TELEPHONE (Include Area Code) | 22c. OFFICE SYMBOL |
|---|---|---|
| Dr. Thomas C. Hartrum | (513) 255-3576 | AFIT/ENG |

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

A software interface was designed and implemented that extends the use of Structured Analysis (SA) Tool (SAtool) as a graphical front-end to the Strategic Defense Initiative Architecture Dataflow Modeling Technique (SADMT). SAtool is a computer-aided software engineering tool developed at the Air Force Institute of Technology that automates the requirements analysis phase of software development using a graphics editor. The tool automates two approaches for documenting software requirements analysis: SA diagrams and data dictionaries. SADMT is an Ada based simulation framework that enables users to model real-world architectures for simulation purposes.

This research was accomplished in three phases. During the first phase, entity-relationship (E-R) models of each software package were developed. From these E-R models, relationships between the two software packages were identified and used to develop a mapping from SAtool to SADMT.

The next phase of the research was the development of a software interface in Ada based on the mapping developed in the first phase. A combination of a top-down and a bottom-up approach was used in developing the software. A summary of the design decisions made in developing the software is presented.

The final stage of the research was an evaluation of the interface using known SADMT architecture descriptions. This technique compared the outputs of the software interface a known SADMT description. A summary of the results is presented.