INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

University Microfilms International
300 N. Zeaib Road
Ann Arbor, MI 48106
Knowledge based simulation system—an application in controlled environment simulation system

Zhang, Guoqing, M.S.
The University of Arizona, 1988
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark √.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy ______
6. Print shows through as there is text on both sides of page ______
7. Indistinct, broken or small print on several pages ______
8. Print exceeds margin requirements ______
9. Tightly bound copy with print lost in spine ______
10. Computer printout pages with indistinct print ______
11. Page(s) ________ lacking when material received, and not available from school or author.
12. Page(s) ________ seem to be missing in numbering only as text follows.
13. Two pages numbered _______. Text follows.
14. Curling and wrinkled pages ______
15. Dissertation contains pages with print at a slant, filmed as received ______
16. Other __________________________________________________________

______________________________________________________________

______________________________________________________________

U-M-I
KNOWLEDGE BASED SIMULATION SYSTEM
-- AN APPLICATION IN
CONTROLLED ENVIRONMENT SIMULATION SYSTEM

by
Guoqing Zhang

A thesis Submitted to the faculty of the
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
In Partial Fulfillment of Requirements
For the Degree of
MASTER OF SCIENCE
WITH A MAJOR IN ELECTRICAL ENGINEERING
In the Graduate College
THE UNIVERSITY OF ARIZONA

1988
STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under the rules of the Library.

Brief quotations from this thesis are allowed without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgement the proposed use of the material is in the interest of scholarship. In all other instances, however, permission must to be obtained from the author.

SIGNED:

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date below:

[Signature]

BERNARD P. ZEIGLER
Professor of Electrical and Computer Engineering
ACKNOWLEDGEMENT

I would first like to thank my advisor, Dr. Bernard P. Zeigler, for supporting the work and making valuable suggestions. He was always available for guidance and suggested the focus on system development and methods of model abstraction (creation). Professor Zeigler has also helped greatly to improve my English writing which requires heavy practice.

The committee composed of Dr. F.E. Cellier and Dr. J.W. Rozenblit made comments which served to both focus on the research and suggest possibilities with respect to knowledge-based simulation and modelling. To them, my special thanks.

Thanks are especially due to the Environmental Research Lab of University of Arizona for sponsoring the research.

Thanks to my parents and Mrs. Lilia Tongson whose encouragement and support has finally helped in yielding this thesis.

Thanks also to all those who ever discussed research topics and problems with me, especially to F. Luttmann.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>6</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>7</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>8</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>9</td>
</tr>
<tr>
<td>1.1 Concepts</td>
<td>10</td>
</tr>
<tr>
<td>1.2 Motivations and Objectives</td>
<td>11</td>
</tr>
<tr>
<td>1.3 Environment</td>
<td>13</td>
</tr>
<tr>
<td>1.4 Outline</td>
<td>15</td>
</tr>
<tr>
<td>2. Knowledge Representation for Simulation and Modelling</td>
<td>16</td>
</tr>
<tr>
<td>2.1 Knowledge Representation Schemes</td>
<td>16</td>
</tr>
<tr>
<td>2.2 System Entity Structure (SES)</td>
<td>17</td>
</tr>
<tr>
<td>2.3 Operations on SES</td>
<td>20</td>
</tr>
<tr>
<td>3. Framework of Model Creation and Utilization</td>
<td>22</td>
</tr>
<tr>
<td>3.1 Model Classification and Abstraction</td>
<td>23</td>
</tr>
<tr>
<td>3.2 Methods of Model Creation (Abstraction)</td>
<td>24</td>
</tr>
<tr>
<td>3.3 Hierarchical Utilization of Models</td>
<td>26</td>
</tr>
<tr>
<td>3.4 Knowledge-Based Simulation and Modelling Systems</td>
<td>27</td>
</tr>
<tr>
<td>3.5 Summary</td>
<td>28</td>
</tr>
<tr>
<td>4. Design of Controlled Environment Simulation System (CESS)</td>
<td>29</td>
</tr>
<tr>
<td>4.1 Introduction to TRNSYS</td>
<td>30</td>
</tr>
<tr>
<td>4.1.1 TRNSYS system and its Components</td>
<td>31</td>
</tr>
<tr>
<td>4.1.2 Component Type Number</td>
<td>32</td>
</tr>
<tr>
<td>4.1.3 System Unit Number</td>
<td>32</td>
</tr>
<tr>
<td>4.1.4 Ordering of Parameters, Inputs and Outputs</td>
<td>33</td>
</tr>
<tr>
<td>4.2 TRNSYS DECK Format</td>
<td>33</td>
</tr>
<tr>
<td>4.2.1 Simulation Control</td>
<td>34</td>
</tr>
<tr>
<td>4.2.1.1 SIMULATION</td>
<td>34</td>
</tr>
<tr>
<td>4.2.1.2 TOLERANCES</td>
<td>35</td>
</tr>
<tr>
<td>4.2.1.3 LIMITS</td>
<td>35</td>
</tr>
<tr>
<td>4.2.1.4 CONSTANTS</td>
<td>36</td>
</tr>
<tr>
<td>4.2.2 Component Control</td>
<td>37</td>
</tr>
<tr>
<td>4.2.3 Listing Control</td>
<td>38</td>
</tr>
<tr>
<td>4.2.4 Miscellaneous Control</td>
<td>38</td>
</tr>
<tr>
<td>4.2.5 Summary</td>
<td>39</td>
</tr>
<tr>
<td>4.3 Requirements of CESS</td>
<td>39</td>
</tr>
<tr>
<td>4.3.1 Approaches and Procedures</td>
<td>40</td>
</tr>
<tr>
<td>4.3.2 Objectives</td>
<td>41</td>
</tr>
<tr>
<td>4.4 Object-oriented Programming Environment</td>
<td>42</td>
</tr>
</tbody>
</table>
LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Working Environment</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>Entity Structure for TRNSYS_SYSTEM</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>System Elements</td>
<td>22</td>
</tr>
<tr>
<td>4</td>
<td>Model Abstraction : Bottom-up VS. Top-down</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>Solar Water-Heating System (SWHS)</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>CESS Composition</td>
<td>46</td>
</tr>
<tr>
<td>7</td>
<td>Activities in CESS Environment</td>
<td>53</td>
</tr>
<tr>
<td>8</td>
<td>Class Hierarchical Structure</td>
<td>56</td>
</tr>
<tr>
<td>9</td>
<td>Class Definitions</td>
<td>59</td>
</tr>
<tr>
<td>10</td>
<td>Class Definition of SYS_INFO</td>
<td>62</td>
</tr>
</tbody>
</table>
LIST OF TABLES

APPENDIX 1. CLASS MODULE 77
APPENDIX 2. TRNSYS_COMPONENTS METHODS 78
APPENDIX 3. TRNSYS_COMPONENTS HELPERS 78
APPENDIX 4. TRNSYS_SYSTEMS METHODS 79
APPENDIX 5. TRNSYS_SYSTEMS HELPERS 79
APPENDIX 6. INIT MODULE 80
APPENDIX 7. MODEL MODULE 80
APPENDIX 8. UTILITY MODULE 81
APPENDIX 9. DECK MODULE 81
APPENDIX 10. TYPE DEFINITIONS FOR SWHS 82
APPENDIX 11. DECK FILE FOR SWHS 86
ABSTRACT

This thesis systematically identifies the building blocks of a knowledge based system for simulation and modelling. We present the design and implementation of Controlled Environment Simulation System (CESS), which bridges a discrete event simulation system (DEVS-SCHEME) and a continuous simulation system (TRNSYS). The rationale behind the approach is that a discrete or a continuous model can be abstracted to a level at which the uniform treatment on these two kinds of models is possible. A top-down approach to model creation (abstraction) is proposed, in contrast to the traditional bottom-up approach.

CESS is implemented on an object-oriented programming environment (SCOOPS on TI-SCHEME). A knowledge representation scheme known as System Entity Structure is employed for MODEL management, recording system structural knowledge, and the utilization of techniques in Artificial Intelligence. Some prospective research topics are also brought up.
1. Introduction

The expanding demand for systems of ever-increasing complexity and accuracy has stimulated the need for higher level concepts, tools and techniques in every area of Computer Science and Engineering, as well as in other disciplines. Some of these fields, in particular, Simulation and Modelling, Artificial Intelligence and System Theory, are attempting to fulfill this need by introducing a new, more abstract level of system description.

Artificial Intelligence, System Theory, Simulation and Modelling have contributed independently to the development of concepts in their own enterprises. The integration of these different approaches has shown to be a definite trend. In this view, we made and are making our efforts to extract the similarities and differences from among them. The results we obtained are promising, especially with the application in reality. We also have learned that selecting a suitable knowledge-representation for the domain knowledge is one of the first problems to be encountered in building a knowledge-based system.

There are many available simulation systems for different purposes. They are written in diverse computer languages, FORTRAN, SIMSCRIPT II.5, etc. Modern modelling techniques are employed in their development. However, it is very difficult to integrate the
methods and techniques used in building these simulation systems because their development heavily depends on the specific application circumstances. This thesis describes the approach to develop a systematic simulation system, Controlled Environment Simulation System (CESS). CESS comes about through the merging of Artificial Intelligence, System design theory, Modelling and Simulation. Such an approach can be suited to developing and integrating similar systems in a knowledge-based simulation environment.

1.1 Concepts

MODEL is a central concept in this thesis. There are many ways to define this term, depending on who is talking about it. Minsky [7] presented the most general definition of Model as:

To an observer B, an object A is a model of an object C to the extent that B can use A to answer questions that interest him about C.

For the purpose aimed at in this thesis, mathematical models are considered. Simulation is thus the experimentation with models [6]. Modelling is the process in which the existing system or the system to be built is abstracted into mathematical models for later simulation and analysis.

A model base contains models in computational form written in computer languages.
In the area of Artificial Intelligence (AI), Knowledge Representation (KR) is one of the most central topics. KR intrinsically codes human knowledge in a certain domain into a form computers can recognize and utilize.

Wymore [17] gave a very general definition of system: A system is defined as a dynamic phenomenon that can be described with physical or conceptual boundaries that effectively divide the universe into two parts, inside and outside the system. Inputs get into the system across the boundaries. Outputs are ported out across the boundaries by the system. The system behavior is noted as the state of the system. Both the state and outputs depend upon the past state and inputs.

A knowledge-based simulation system or environment is the product of the achievements in both AI and Simulation Modelling. As in other AI systems, interactive access and display of attribute and behavioral description of systems are often provided. However, the model concept is the kernel of the intelligent management for system modelling and simulation. Some facilities for automatic result analysis are available for recommending good system design based on the simulation results.

1.2 Motivations and Objectives

Zeigler [18] identifies five elements constituting the basic approaches to modelling and simulation. They are (1) the real system, (2) the experimental frame, (3) the base model, (4) the lumped model,
and (5) the computer. Two levels of abstractions of models are mentioned: the base model and the lumped model. This doesn't cover the possible abstractions of models which we will look further into in the sequel. Zeigler's formalization of simulation models [18, 20] and discussion of applications [19] serve as a good starting point for formal discussion of model abstraction. However, much of the recent formal work was spurred by the earlier work in mathematical system theory [16, 17]. Innis [5] also presents an approach to tackle the problems of model abstraction. Huzar [4] gave a rough description of model generation at a less abstract level. Nevertheless, we need to look into this issue systematically and seek the ways to create model descriptions at more abstract levels.

The incorporation of Knowledge Representation into simulation, modelling and system construction has been studied widely in recent years [11, 21]. From the application point of view, in general terms, people in fields other than AI have the desires to take advantage of AI techniques to perform various tasks intelligently. The gap between people in AI and people in other fields, Biology, for example, can be narrowed via a knowledge-based system dealing with the communication, organization and bridging problems. Such an approach can overcome some drawbacks in the "classical" modelling and simulation (lack of flexibility in expressing model structures, extensive programming effort, batch-oriented modelling, etc.).

Later we will see that a continuous simulation system, TRNSYS, can be linked to a discrete event simulation system, DEVS-SCHEME.
Theoretically, this work contributes to the investigation of methods to treat discrete and continuous models uniformly. Practically, it further explores the uses of TRNSYS and DEVS-SCHMEM.

With the above observations and requirements, we will make our efforts to investigate the following issues:

(1) Choice of a knowledge representation scheme.
(2) Methods for model creation and prospective utilizations.
(3) Development of tools for design and implementation of a knowledge-based simulation system.
(4) Framework of knowledge-based simulation system in an object-oriented programming environment.
(5) Design and Implementation of a practical knowledge-based simulation system: Controlled Environment Simulation System (CESS).

1.3 Environment

Currently, we have an object-oriented programming environment hosted in TI-Business-Pro personal computers (IBM AT compatible). The environment is set up by SCOOPs, a system built on TI-SCHMEM which is based on first-class environments and on multiple and dynamic inheritance [14]. TI-SCHMEM is a lexically scoped dialect of LISP developed at MIT. ESP4 [26] (Entity Structurer and Pruner, Version 4) is a PASCAL system available for VAX under VMS, and being ported to SCOOPs/TI-SCHMEM environment. ESP4 can perform the management of Knowledge Representation suitable for Simulation and Modelling, and
provides rich facilities to integrate models and organize large project activities. ESP-SCH
EME implements most of the facilities provided in ESP4, and has some extra useful tools for simulation and modelling. A DEVS-SCH
EME simulation and modelling environment is available for simulation tasks [23]. It supports constructing hierarchical discrete event models.

An IBM-PC version of TRNSYS 12.1 is available for transient system simulations with modular structures. TRNSYS is well suited to detailed analysis of dynamic systems whose behavior is dependent on the passage of time [15].

Figure 1. Working Environment
The CESS system to be built is to set up the interface between TRNSYS and DEVS-SCHEME in order to take advantage of AI achievements, and to simplify the use of TRNSYS for various practical simulation applications.

Figure-1 depicts the environment for the work to be performed. Each of the components will be explained later.

1.4 Outline

The thesis is application oriented, along with some insights into the theoretical research, especially in the model creation and utilization.

A brief introduction to knowledge representation is given at the beginning of chapter 2. System Entity Structure, a knowledge representation scheme, is also introduced along with its formal definition. The characteristics of knowledge-based simulation and modelling systems are addressed at the end of chapter 3. The other part of chapter 3 deals with the model creation and utilization. In chapter 4, CESS (Controlled Environment Simulation System) is described in detail at design level. Implementational issues concerning CESS are detailed in chapter 5. Finally in chapter 6, some conclusions are drawn; and the contributions of this thesis together with prospective research issues brought up by this thesis are also given.
2. Knowledge Representation for Simulation and Modelling

In the era of AI, knowledge representation (KR) received the most attention from researchers. Many representation schemes have been established. Some of the notable contributions have been made by Minsky [8], for example. Aside from the AI approach, the dynamic systems approach aims at representing the dynamic knowledge of systems from systems theory perspectives. If a KR is not properly chosen, a good knowledge-based system can not be well established. We will briefly describe knowledge representation schemes, followed by the clarification of the concepts of System Entity Structure (SES). The reasons for choosing SES for CESS are addressed. Finally, operations on SES are explained.

2.1 Knowledge Representation Schemes

Different KR schemes are found to be suited only to particular categories of knowledge. In the AI family, they exhibit similar restrictions to Production Rules, Predicate Calculus, Procedural Representation, frames, decision trees and Semantic Networks. In dynamic systems, however, they fall into two main classes, causal and empirical representations [24].

With the recognition that each of the above KRs in itself is incomplete, the coexistence approach is proposed by O’Hare [9].
Unfortunately, this approach also neglects the dynamic aspect. As a consequence, a more enriched framework capable of utilizing both structural and behavioral knowledge emerged that is derived both from AI and from dynamic systems approaches [25]. This framework is denoted as System Entity Structure (SES).

In the work described in this thesis, SES plays a key role in:

1. Characterizing the system structures, the structural knowledge about the system.

2. Specifying the knowledge associated with the models, the behavioral aspect of the knowledge about the system.

3. Representing the design purposes of multifaceted systems and coding the prospective system configurations.


5. Allowing the ultimate synthesis of a system design from design models in the model base.

Because of the above properties, SES has been selected for the design and implementation of CESS to be described in the sequel.

2.2 System Entity Structure (SES)

System Entity Structure (SES) can be visually depicted as a tree-like structure [18, 20, 24]. An entity is a node in the tree,
corresponding to an object in reality, conceptually or physically. Each entity may have several aspects, each denoting a decomposition and therefore having several entities. In Figure-2, Solar-Water-Heating-System (SWHS) is a system belonging to transient system modules. Each module has three functional parts, INPUT, OUTPUT and INTERNAL. The input part, in turn, has several other subparts. We will explain more on SWHS in chapter 4 and chapter 5.

Whereas the entity-aspect relation conveys decomposition knowledge, the entity-specialization relation expresses taxonomic knowledge. In Figure-2, the entity MODULE has several alternatives under the specialization, SYSTEM_TYPE.

A multiple entity expresses a set of entities that are all of the same type. Three vertical lines are depicted as the multiple decomposition into the individual entities of the same type. MODULEs in Figure-2 is a multiple entity. A multiple entity represents the extension of class as opposed to its intension (class definition). Class variables, carrying aggregation and distribution information, are associated with the multiple entity, whereas instance variables, belonging to each instance of the class are associated with the individual entity.

SES is the enriched framework capable of utilizing both kinds of knowledge representation schemes, deriving from AI approaches and from dynamic systems. Zeigler and Zhang formalized SES using the set mechanism [25, 27]. The formalization of SES makes it possible to
Figure 2. Entity Structure for TRNSYS_SYSTEM
prove the completeness, correctness and irredundancy of two kinds of transformations which support the use of the structure for automated construction of models from a model base.

For more detailed discussion on the concepts of System Entity Structures, the reader is referred to Zeigler [18, 24].

2.3 Operations on SES

As mentioned in section 2.1, the knowledge framework just introduced is intended to be generative in nature, i.e., it should be a compact representation scheme which can be unfolded to generate the family of all possible models synthesizable from components in the model base. Each final system structure generated above is represented by a pure entity structure, which has no specialization.

In order to get a pure entity structure, some pruning operations are required. These operations can then ultimately reduce the structure to a composition tree. This contains all the information needed to synthesize a model in a hierarchical fashion from components in the model base. Rozenblit [12] proposed some algorithms for this purpose. Zeigler and Rozenblit further elaborated this approach in [13].

Transformations, part of the pruning operations, are well organized by Zhang [27]. Two exclusive and complete forms of transformations are recognized. Zeigler and Zhang [25] proposed ways to prove the correctness of these transformations by building a formal definition for SES based on set theory.
In addition to two forms of transformations, some other pruning operations are used to select an aspect for each non-atomic entity, to specify an alternative for an entity, and to concretize information associated with the aspects, entities and specializations. Here atomic entity is defined as having neither specialization nor aspect. As in Figure-2, PRINTER is an atomic entity, but MODULE is not.

The operations on SES are very useful for CESS users who can thus work with the SES pruning interface to construct a system model structure rather than writing an extensive set of coupling specifications for the system in an obscure numerical coding.
3. Framework of Model Creation and Utilization

Any model in some sense is abstract to some extent because it stays at a certain level of detail and generality [3]. The whole universe cannot be simulated, so we need to specify to what limit we should characterize a system into an abstract model [18]. Model abstraction is thus a process of abstracting models for systems to be simulated. A model becomes significant when it is effectively used. We will investigate the ways to create models and to use them as well. The building blocks of our knowledge-based simulation system are described. We base our work on the DEVS formalism [20] and DEVS-SCHEME [23], a Lisp-based discrete event simulation environment combined with object-oriented programming facilities. System Entity Structure is used to organize models.

<table>
<thead>
<tr>
<th>System Elements</th>
<th>Variables</th>
<th>Values</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Outputs</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>States</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Figure 3. System Elements
3.1 Model Classification and Abstraction

Models can be classified in many ways. In the foregoing sections, we described a definition of system from a system-theoretical point of view. The same perspective can be employed to classify models. A model is identified with three systemic elements, i.e., inputs, outputs, and states. As depicted in Figure-3, each systemic element has to be represented by appropriate variables with well defined range sets, i.e., possible values and appropriate functions to specify how the values are organized in time [10].

The distinction between a discrete event model and a continuous model can be made with the observation as follows: in a discrete event simulation, only a finite number of state changes of the model can occur in any finite time interval, while the number of state changes of the model in a finite time interval of a continuous time simulation may be infinite.

We can notice common features of the above models of different categories. First of all, they in some way all characterize either an existing system or a system to be built. But none of them is sufficient to address all aspects of modelling at an appropriate degree of abstraction. Zeigler [18], as mentioned in chapter 1, defined two levels of model abstractions, i.e., the base model and the lumped model. A number of comparable efforts have been undertaken to construct lumped models from base models, a process which is well understood by the modeller. This bottom-up approach can succeed in
many cases of model construction and abstraction. An alternative approach would be the top-down approach. This route so far received much less attention. Huzar [4] made use of this approach, but his results are limited in generality to just one specific domain. In the top-down approach, a model can be refined into sets of submodels which can then be abstracted to form a more abstract model by way of bottom-up methods (Figure 4). The top-down approach is useful as we will find in building CESS to be described later in chapter 4.

![Figure 4. Model Abstraction: Bottom-up VS. Top-down](image)

3.2 Methods of Model Creation (Abstraction)

Two basic techniques of the bottom-up approach are: (1) to aggregate models and data structures, and (2) to simplify the model interactions [11]. The former one is also called static abstraction with which equivalent model aggregation combines several models into a single model. The new model must be in some sense the sum of the original models. If the new model's parameters are adjusted correctly, the new model will be functionally similar to the old group. The rest of the models are not affected. Similar results can
be obtained in data structure (class) aggregation.

The simplification of model interactions is often referred to as dynamic abstraction which aims at simplifying simulation models by analyzing the stimulus-response event behavior of one or more models. It also tries to construct a single model with stimulus-response behavior that is statistically similar [11].

A very interesting investigation of the bottom-up approach is found in [3] where Fishwick proposed a new theory for hierarchical reasoning about processes involving complex actions and a variety of objects (models). The method for simulating processes over multiple levels of abstraction was also described.

But the implication of model/process abstraction involves more than just the above points. At a certain abstraction level, we can, by one way or another, tear a model into parts which can receive more detailed analysis. By applying bottom-up techniques to these submodels at this time, we can get a more understandable and more useful model in contrast to the original model. The idea of this approach originates from the common situation where a problem can not be solved directly but the round-about way can lead to a satisfactory answer to the problem. This is what we called top-down approach to model/process abstraction.

The combination of top-down and bottom-up approaches to model/process abstraction (creation) forms the base of the design and implementation of the CESS system.
3.3 Hierarchical Utilization of Models

With bottom-up and top-down methods of model abstraction (creation), several models at different abstraction levels can be used to represent the same thing (system, object, etc.). In this way, the model or process can be studied at abstraction levels that are appropriate for a specific interest or task. For the purpose of the thesis, three levels of model utilization are recognized: system level, model description level, and code level.

The world can be viewed as the collection of cooperative systems. The system level for model description is the most abstract level at which both discrete and continuous system/model descriptions can be correlated and studied uniformly. The way to reach this goal calls for a formal system specification common to both the discrete system formalism (DEVS by Zeigler [20]) and the continuous system formalism (ODE by Cellier [2]). This formalism must be able to characterize the structural and behavioral aspects of the systems.

At the model description level, models of either discrete or continuous nature can be described in terms of their relations to other models and the fundamental conditions for their existence, such as an initial value of an input variable in a continuous model. Utilization of models at this level is explored later in chapter 4 and chapter 5. The correlation between TRNSYS [13], a continuous simulation system, and DEVS-SCHME, a discrete event simulation system, is set up. This results in CESS, a knowledge-based simulation system (environment).
The code level is the lowest level of model utilization. Several computer languages can be used to codify the models. Cellier [1] described several languages aiming at describing models at this code level. It is true that the merging of continuous and discrete simulation at this low level can solve many problems encountered in modelling and simulating real systems. But it is also true that the techniques of model abstraction and knowledge-based simulation and modelling are not easy to employ at this level. With this perspective, we chose to implement CESS at the model description level.

3.4 Knowledge-Based Simulation and Modelling Systems

We have described the concepts of System Entity Structure (SES) which serves as an adequate knowledge representation scheme for knowledge-based simulation and modelling. In chapter 4, we will resume this issue in more detail. In addition to the knowledge representation scheme, several other components are required: Management of Knowledge Representation, Knowledge Access, MODEL management, MODEL reasoning, Simulation Control and User Interface.

Knowledge Representation Management enables users to codify their knowledge about the system to be studied. Storage and retrieval primitives are also supplied. The consistency of knowledge about the system is maintained as well.

Knowledge access is the critical issue that specifies which and
how users can get access to knowledge about the system.

MODEL management allows users to organize their models of the same system independent of other system models. A model base is thus the key component for MODEL management.

MODEL reasoning is responsible for model creation and abstraction. This is the most intelligent part of a knowledge-based simulation and modelling system, and supports multiple levels of model abstraction (creation).

Simulation Control deals with the simulation initiation and termination. The mechanism to organize model interactions and behaviors also rest on Simulation Control.

The requirement for USER interface varies according to the available equipment for implementation. The ideal operation environment will be equipped with a graphic display capability and fancy window facilities.

3.5 Summary

We described in this chapter several model abstraction methods, usages of models at multiple levels, and the primitive components in a knowledge-based simulation and modelling system. Based on these discussions, we have designed and implemented a knowledge-based simulation and modelling system, CESS (Controlled Environment Simulation System).
4. Design of Controlled Environment Simulation System (CESS)

A controlled environment is a self-sufficient system which is isolated from gas and other material exchange with the surrounding earth environment. The system, while materially closed, is open with respect to energy (the sun being the major source) and information (management and control). Material exchange will be also open for such expendables as equipment, spare parts, clothes, etc. Several modules are enclosed within the environment. Each module replicates a biome (self-sufficient eco-system): rain forest, savanna, desert, and ocean. In addition, a module for human habitation and one for intensive agriculture are also enclosed. Whether or not such an environment can be built is still under research. Simulation is the best way to investigate the possibilities and problems of developing such a complicated system.

TRNSYS, a simulation system, may simulate a wide variety of dynamic systems, and is especially well suited for the analysis of energy systems (e.g., heating, cooling and dehumidification systems for control of temperature and humidity). TRNSYS is appropriate when time-dependence is invoked for any or all of the following reasons:

(1) A component of the system has an OUTPUT variable which is a function of time.

(2) The mathematical algorithm of a component of the system
involves one or more time-dependent differential equations (with non-zero derivatives).

(3) A physical quantity calculated by the simulation model is to be interpreted over time by the quantity integrator component.

TRNSYS requires its users to describe the system to be simulated by means of an input data file coded in a special form. The file is referred to as TRNSYS DECK. The steps for setting up the TRNSYS DECK for different applications are almost the same. This motivates building a user-friendly front-end to TRNSYS. CESS is such a product (Figure-1).

Conceptually, the methodology of building CESS can readily be adapted to the development of similar systems. Before listing the requirements of CESS, the relevant concepts in TRNSYS are explained. The supporting tools of CESS are then outlined.

4.1 Introduction to TRNSYS

This section provides the concepts and definitions needed to understand the use of TRNSYS for hierarchical modular system simulation and modelling. The steps which should be followed to prepare a system for simulation using TRNSYS are then summarized. Much of the information given below stems from [15].
Figure 5. Solar Water-Heating System (SWHS)

4.1.1 TRNSYS system and its Components

A TRNSYS system contains a set of components, interconnected in such a manner as to accomplish a specified task. For example, a typical solar water-heating system (Figure 5) may consist of a solar collector, a pump, several temperature sensing controllers and some output equipment.

With a program such as TRNSYS which has the capability of interconnecting system components in any desired manner, solving
differential equations, and facilitating information output, the entire problem of system simulation is reduced to the problem of identifying all of the system components, and formulating a general mathematical description of each. The description is collected into a file, TRNSYS_DECK file. In chapter 5, the TRNSYS_DECK for SWHS will be described in detail.

4.1.2 Component Type Number

TRNSYS identifies the different kinds of components by associating a TYPE number with each kind of component. For example, the solar collector model provided with TRNSYS is designated a TYPE 1 component. The relation between a kind of component and its TYPE number is determined by TRNSYS internally.

4.1.3 System Unit Number

TRNSYS recognizes each component of a system by a unique but arbitrarily user-assigned UNIT number. The TYPE number does not always uniquely identify a component in the system, because there may be more than one component of the same TYPE in the system. As an example, we have two PRINTERS in SWHS. They all belong to the model type, PRINTER, but carry the unique unit numbers, 4 and 5, respectively.

UNIT numbers are unrelated to TYPE numbers, and need not be in sequential order. The only restrictions imposed on UNIT number
selection are that no two system components can have the same UNIT number and each UNIT number must be a positive integer between 1 and 50.

Thus, associated with each component in the system is a TYPE number, which defines the component's function, and a UNIT number, which distinguishes the component from all other components in the system.

4.1.4 Ordering of Parameters, Inputs and Outputs

A component may receive three types of information: INPUTS, PARAMETERS, and TIME. It sends information through its OUTPUTS. To aid in specifying the information flow to TRNSYS, each piece of information of a given TYPE (for each component) is numbered sequentially. If for example, a component has two inputs, they are numbered INPUT 1 and INPUT 2, respectively. But the actual names can vary. The same is true for the OUTPUTS, the PARAMETERS and DERIVATIVES of every system component.

4.2 TRNSYS_DECK Format

The purpose of the following sections is to describe the format of TRNSYS_DECK required to simulate a system. TRNSYS_DECK contains four kinds of information, component control, simulation control, listing control, and other miscellaneous controls.

The concepts and definitions given in section 4.1 serve to
indicate to the users the nature of the information required to describe a modular and hierarchical system consisting of interconnected components.

Component control defines the components in the system to be simulated together with their interconnections. Simulation control directs the operation of TRNSYS and defines such things as simulation length and error tolerances. Listing control affects the output from TRNSYS. Miscellaneous control contains the marking of the end of the TRNSYS_DECK, commenting statements, and the like. Appendix 11 shows the DECK file for SWHS.

4.2.1 Simulation Control

Simulation control is to specify the parameters for a TRNSYS simulation such as its duration and the error tolerances to be used. Four types of simulation control information are identified, SIMULATION, TOLERANCES, LIMITS and CONSTANTS. SIMULATION must be included in TRNSYS_DECK. The others are optional.

4.2.1.1 SIMULATION

SIMULATION determines the starting and stopping time of the simulation as well as the timestep to be used. The format is:

SIMULATION t0 t1 t2

where t0 is the hour of the year at which the simulation is to begin.
35

t1 is the hour of the year at which the simulation is to end.

t2 is the timestep to be used for numerical integration.

4.2.1.2 TOLERANCES

TOLERANCES is the optional control information used to specify the error tolerances to be used during a TRNSYS simulation. The control has the form:

TOLERANCES A B

or TOLERANCES -C -D

where A is a relative (and C is an absolute) error tolerance controlling the integration error.

B is a relative (and D is an absolute) error tolerance controlling the convergence of input and output variables.

If a TOLERANCES is not present in a TRNSYS_deck, the following TOLERANCES are assumed:

A = 0.01 and B = 0.01.

4.2.1.3 LIMITS

The LIMITS is an optional control and is used to set limits on the number of iterations that will be performed by TRNSYS during a timestep before it decides that the differential equations or the algebraic equations are not converging. The LIMITS control has the
form:

LIMITS m n p

where m is the maximum number of iterations which can be performed during a timestep before a WARNING message is printed out.

n is the maximum number of WARNING messages which may be printed before the simulation terminates in ERROR.

p is the optional limit. A component will be traced after it is activated p times. p has a default value of m if p is not given.

4.2.1.4 CONSTANTS

This control is useful when simulating a number of systems with identical component configurations but with different parameter values, initial input values, or initial values of time dependent variables. The format is:

CONSTANTS n

NAME1 = VALUE1 ... NAMEn = VALUEn

A constant name is a letter followed by up to two additional characters. Other characters are ignored. Values may be numbers, constants which have already been defined, or simple expressions which are much as FORTRAN arithmetic statements with some exceptions:

(1) Blanks must be placed on both sides of the operation codes +,
-, *, and /.

(2) Expressions are evaluated from left to right with no precedence of any operation over another.

(3) Parentheses are not allowed.

4.2.2 Component Control

This piece of control information specifies the number of inputs, parameters, outputs and derivatives employed in each component model as well as the type of the component being used, the values of PARAMETERS, and the initial values of the INPUTS and DERIVATIVES. The format for this control is as follows:

UNIT n  TYPE m  Comment
PARAMETERS p
PAR1, PAR2, ... PARp
INPUTS i
F1,T1 F2,T2  ...  Fi,Ti
INP1, INP2, ... INPi
DERIVATIVES d
DER1, DER2, ... DERd

where n is the system UNIT number for the component, m is the TYPE number of the component, p (i, d) is the number of parameters (inputs, derivatives) of the component. PARk (INPk, DERk) is its (initial) value. Fk,Tk is a pair telling that the input NO.k will receive message from the output NO. Tk of the component with unit number Fk. If p is zero, PARAMETERS can
be removed. Same with INPUTS and DERIVATIVES if $i$ or $d$ is zero, respectively.

4.2.3 Listing Control

Listing control specifies the format of TRNSYS output listing. They are:

1. WIDTH $w$
2. LIST
3. NOLIST

WIDTH tells the number of characters to be allowed on a line of TRNSYS output. $n$ is the specified value and must be between 72 and 132. LIST (NOLIST) is used to turn on (off) TRNSYS output listing.

4.2.4 Miscellaneous Control

Some controls are used to direct the correct performance of TRNSYS. They are:

1. MAP
2. END
3. TRACE
4. COMMENT

MAP is an optional control used to obtain a component output map listing which is useful in debugging component interconnections.

END indicates to TRNSYS that no more control information follows and that the simulation can commence at this point.
TRACE causes the values of the PARAMETERS, INPUTS, OUTPUTS and DERIVATIVES of the component to be printed out whenever that component is referenced by TRNSYS during a simulation.

COMMENT is used to document the TRNSYS_DECK to make it more readable and understandable. A comment statement must have a star, *, in column one of a line.

4.2.5 Summary

A TRNSYS simulation is defined and controlled by a set of control information as described through section 4.2.1 to 4.2.4. SIMULATION must appear in each TRNSYS_DECK file, usually near the beginning of the TRNSYS_DECK file.

Each component is identified with a UNIT number and a TYPE number. Interconnections are specified with Component control. The simulation starts when END is encountered by the TRNSYS_DECK processor.

4.3 Requirements of CESS

System Entity Structure (SES), as described in section 2, is the appropriate knowledge representation scheme for our purpose. SES forms a solid base for the CESS development. CESS is a knowledge-based simulation system. The importance of model creation and effective utilization of models have been discussed in section 3. The requirements can now be defined with respect to TRNSYS, and the TRNSYS_DECK described above.
4.3.1 Approaches and Procedures

A component of a system is described in a fixed format which is part of the TRNSYS_Deck, though the information about the component may vary greatly in content. The descriptions of a component in TRNSYS match satisfactorily what the DEVS formalism [20] can do for simulation model descriptions. DEVS-SCHEME is thus the ideal environment for building CESS. In addition, DEVS-SCHEME supports the management of SES which is employed to organize the model (component) specifications as well as the interconnections among the models (components).

Some common procedures exist in setting up the TRNSYS_Deck for different system simulations in TRNSYS:

1. Choose component TYPES.
2. Make instance components out of the TYPES selected above.
3. Extract information from the instance components.
4. Establish the interconnections.
5. Specify the simulation control information.

The above observations suggest that we need the following: (1) a model base, MBASE which stores all model (component) knowledge, (2) an entity structure base, ENBASE which stores the TRNSYS system composition knowledge (structure, interconnections, etc.), (3) a TYPE base, TBASE which has all knowledge about the TYPE definitions, and (4) the TRNSYS_Deck file base, DECK_BASE which stores the TRNSYS_Deck files for TRNSYS to perform the simulations. These memory units are
depicted in Figure 1.

The feasibility of this approach will be further justified in section 4.4, where the mapping of a TRNSYS system to a DEVS model can be readily seen.

4.3.2 Objectives

We have identified the approach to build CESS and the procedures to construct the TRNSYS simulation system specifications. To meet the needs of knowledge-based simulation and of bridging continuous (TRNSYS) and discrete (DEVS) event simulation, CESS has to meet some additional requirements described in later sections. CESS is built upon DEVS-SCHEME, and its functional requirements are thus designed to be discrete event simulation oriented.

The following are the requirements of CESS:

(1) Declaratively represent models (components in TRNSYS) that reduce or eliminate programming efforts.

(2) Represent behaviors of system components through an object-oriented (frame-based) Knowledge Representation so that components can be altered without altering the simulation model interpreter.

(3) Automatic model abstraction where model is represented at multiple levels of abstraction so the users can specify the simulation level, and the system in turn automatically configures the models.

(4) Interactive access to model building and simulation through a
command interface using windows and graphics.

(5) Descriptions of TRNSYS Simulation components in terms of the System Entity Structure.

(6) Retrieval of TRNSYS simulation components from the Entity Structure without user interventions.

(7) Pseudo-simulation of TRNSYS components in the DEVS-SCHEME environment. Pseudo-simulation is explained in section 4.5.3.

(8) Definition of a complete TRNSYS simulation (composition and coupling) by pruning and transforming existing entity structures.

The above definitions are functional in nature. The detailed issues will be addressed in chapter 5 where the implementation of CESS is presented.

4.4 Object-oriented Programming Environment

As mentioned in section 4.3.2, CESS is built upon an Object-Oriented Programming Environment [14] in order to explore and use modern AI techniques. This section describes the fundamental concepts and definitions of an object-oriented programming language, followed by the illustration of the correspondence between objects in object-oriented programming style and the models in simulation. Much of the information below is referred to [25a].
4.4.1 Object-Oriented Programming Concepts

A program written in conventional languages such as C, FORTRAN or PASCAL is centered around a main routine which invokes other cooperative routines at appropriate times. The main routine acts as the central officer, the others play a supporting role and come alive only when the control flow is passed through them. An object-oriented program, however, has a group of objects. Such objects may behave as units of their own and possess sufficient knowledge to tackle problems and to make decisions falling into their scopes. As a result, the main routine can be removed. A desired task to be performed by an object can be initiated by sending messages to the object.

OBJECT is the key concept in object-oriented programming style. Each object has its own descriptive variables and methods to manipulate its variables. The methods can assign and update the values of variables. The variables characterize the state of an object. Methods are the only means by which states of objects can be changed.

Objects must communicate with other objects. The process to enable objects to communicate with each other is called message passing. A message from an object (calling object) contains the information which another object (called object) will process. The calling object must inform the called object of the method which is to operate on the message. Upon receiving the message, the called object may change its state and produce an output in response to the message.
One of the many features in object-oriented programming is that different objects can have variables or methods with identical names. The same message can be passed to equally named methods in different objects. However, the results can vary greatly. This approach encapsulates the implementation details of methods from outside the object. Besides, it supports the idea of model abstraction as described in chapter 3. Indeed, the abstraction offers the ability to extend a software system by introducing new definitions without modifying earlier ones, as long as the external description of the definition remains the same.

In an object-oriented programming environment, an object is defined as the instance of an object class, a template for generating any number of instances. Each of the instances copies the basic information associated with its object class. In the case of the Solar Water Heating System (SWHS as in Figure 5), SWHS can be defined as a class which can then be used as the basis for creating instances of SWHS.

Some so called class variables are associated with the class definition. Any instance out of this class can obtain access to the class variables. Instance variables, however, are declared in class definition and are local to the specific instances. In any event, the initial values of both class and instance variables can be defined at the time when the class is defined.

Large systems can be organized through an inheritance structure of classes, thus permitting a modular design and avoiding the
specification of redundant information. Inheritance thus supports building a high level of abstraction as mentioned above. As an example in Figure 2, the Solar Water Heating System and the Air Flow System all have the same system composition (INPUT, OUTPUT, INTERNAL). Such information can be associated with the class, MODULES, which is the higher level definition for both systems, SWHS and AFS. In the definition, a class, say A, can be explicitly specified as the superclass of another class, say B. B is thus called the subclass of A. The class and instance variables of B are the union of class and instance variables of its superclasses. However, the class variables that B inherits from its superclass, A, are shared by all instances of B, but not shared by instances of its superclass, A. Methods of A can be inherited by B with one exception that B’s methods supersede A’s methods which have the same names as those of A’s.

4.4.2 Objects in Simulation and Modelling

Comparing the concepts of object-oriented programming and its inheritance principles with the previously exposed programming elements of TRNSYS, we notice the close correspondence between the CLASS concept and the TYPE in TRNSYS, and between the INSTANCE concept and the system COMPONENT in TRNSYS. This directs us to choose DEVS-SCHEME as the working environment for CESS. Beyond that, DEVS-SCHEME provides abundant facilities for managing and organizing instances (components) of different classes. In the following sections, the different CESS management elements are specified, while the design stage and implementational details are deferred until chapter 5.
MEMORY

TRNASYS DECK files

TRNSYS DECK files

create-deck-header
create-deck-unit
create-deck-end

CESS COMPOSITION

choose-type
show-type
update-type

def-instances
create-deck

TYPE Definitions

DECK BASE

create-deck-header
create-deck-unit
create-deck-end

DEVS-SCHEME ESP-SCHEME

update-instance
serial-update-instance

save-type-info
restore-type-info

Figure 6. CESS Composition
4.5 Organization of CESS

Figure 1 shows the whole environment upon which CESS is built. Figure 6, however, depicts the physical composition of CESS. The fundamental description of each of the physical parts is given in the sequel. The four primary components of CESS management are as follows: TYPE management, INSTANCE management, DECK management, and Pseudo-simulation.

4.5.1 TYPE Management

The information common to TRNSYS (continuous) and DEVS (discrete) is captured in this management module, thus facilitating further management within the DEVS-SCHEME environment. TYPE information described in section 4.1 is summarized below:

- TYPE name
- TYPE number
- Number of PARAMETERS; their names and initial values
- Number of INPUTs; their names and initial values
- Number of OUTPUTs; their names and initial values
- Number of DERIVATIVEs; their names and initial values
- DELETING FLAGS: (DINP DOUT DPAR DDER), DINP specifies whether the input variables can be deleted after the instance of the TYPE has been created. DOUT, DPAR and DDER are corresponding flags for OUTPUTs, PARAMETERs and DERIVATIVEs, respectively.
ADDING FLAGS : (AINP AOUT APAR ADER), AINP specifies whether the input variables can be added when the instance of the TYPE is made. AOUT, APAR and ADER are for OUTPUTs, PARAMETERs and DERIVATIVEs, respectively.

This information will be put together as the definition of the TYPE, then put into the TYPE_BASE (TBASE) which contains all TYPE information.

The UPDATE facility provides three operations:

Adding -- Introduce additional variables (Parameters, Inputs, Outputs, Derivatives) into TYPE.
Deleting -- Delete variables from TYPE.
Modifying -- Change values and names of variables.

The mapping from TYPE to CLASS of the object-oriented environment is part of the function of this module. Other facilities related to TYPE management are:

SAVE and RESTORE TYPE information from TYPE_BASE.
DISPLAY TYPE information on screen.
Hard copy of TYPE information.

Since TYPE management is the very first step leading to the other parts of CESS, we require this task to be as simple and flexible as possible.
4.5.2 INSTANCE Management

Following TYPE management, INSTANCE management faces the problem of mapping a TRNSYS system component to an object in the DEVS-SCHEME environment, i.e. the bridging between a continuous model and a discrete model. The inheritance mechanism embedded in the DEVS-SCHEME environment makes it feasible to replicate several TRNSYS system components of the same or different TYPES. Each component has its own unique unit number.

INSTANCE management should generate multiple copies of a certain TYPE for system components with unique identifiers. The order in which components are created is not significant. An option is provided for defining and updating system components. Knowledge about system components can be added to and extracted from the Knowledge Base (ENBASE, MBASE, TBASE).

Models, as mentioned in chapter 3, have several forms of abstractions. Models in forms of executable procedures are stored in the TRNSYS library. TBASE contains the model types at the descriptive level. MBASE however has high level abstractions of models in terms of DEVS-SCHEME statements.

Model creation is the key requirement of INSTANCE management, enabling pseudo-simulation of any TRNSYS system. Section 4.5.3 will give more details about this topic. Facilities for model utilization are also part of INSTANCE management. Some miscellaneous controls
include the mapping function between System Entity Structure and models, and the interface maintenance between DEVS-SCHEME and TRNSYS models (TBASE, TRNSYS library, MBASE).

4.5.3 Pseudo-simulation Management

A pseudo-simulation is an important issue in CESS design. As described in the foregoing section, TRNSYS routines or models are coded in FORTRAN, and are not usable by DEVS-SCHEME. As to the high level control over the controlled environment, we need a way to demonstrate the real simulation performed in the TRNSYS environment. By constructing conceptual models in DEVS-SCHEME, a TRNSYS system composition and the interactions of its components can be shown on the screen. Pseudo-simulation calls upon supports from both DEVS-SCHEME and ESP-SCHEME. SES is again the key element supporting the pseudo-simulation.

In foregoing sections, we justified that SES is suited for our management of TRNSYS models, system structures, and system coupling specifications. As a knowledge-based simulation system, CESS should explore SES as follows:

(1) SES generation, or INVERSE-TRANSFORMATION
(2) TRNSYS system specification in terms of SES
(3) Transformation of SES to models in DEVS-SCHEME
(4) Connection of SES to TYPEs
(5) Connection of SES to TRNSYS system components.
SES generation, or inverse-transformation, takes a model structure in DEVS-SCHEME and constructs the corresponding entity structure under isomorphism principles (a correspondence between the two structures preserved under all relevant operations on and queries about them) [24]. The resulting entity structure has information about the system structure and the detailed component descriptions such as input variables, parameters, etc. Thus the entity structure can be pruned and transformed into models.

An entity structure can also be constructed via a sequence of commands (similar to commands in ESP4 [26]). SES is descriptive in nature and must be made to connect executable models in MBASE or the TRNSYS library in order to perform pseudo-simulation or real simulation. Procedure TRANSFORM provided by DEVS-SCHEME is used for this purpose.

The correspondence between an entity in SES and a system or system component in TRNSYS, and between a multiple entity in SES and the TYPE in TRNSYS, is readily observed. So, a TRNSYS system can be easily represented by an entity structure.

4.5.4 DECK Management

DECK, as described in section 4.2, contains the text data for TRNSYS to run an actual simulation as opposed to the pseudo-simulation in DEVS-SCHEME.

DECK_BASE is the physical storage of TRNSYS deck files. Each TRNSYS system has a corresponding file in DECK_BASE available to the
TRNSYS interpreter (running on IBM PC compatible). The following requirements are identified:

1. Interactive interface to user to define the simulation control, component control, listing control, and others.

2. SAVE to and RESTORE from DECK_BASE the information about the TRNSYS system, including the configuration and the internal state information.

3. Maximum flexibility for use with minimum user intervention in creating TRNSYS deck files for real simulation.

4. Automatic error checking while creating DECK files such as the unassigned or invalid values of some variables (INPUTS, PARAMETERS, OUTPUTS, DERIVATIVES).

4.6 Summary

TRNSYS, like other continuous simulation systems, has a way to define simulation models and the component configurations. Because of the correspondence between such a descriptive scheme and the existing theory of system definitions, the combination of discrete and continuous modelling and simulation can be achieved at a high abstraction level. CESS presents a first step in this direction.
Figure 7. Activities in CESS Environment
CESS can specify the hierarchical models. If these models are of discrete event type, they can be simulated in the DEVS-SCHEME environment. However, CESS transforms the hierarchical models into single level models (continuous or discrete), so that TRNSYS can accept these model descriptions.

Upon introducing TRNSYS and DECK format, requirements of CESS are given at design level. An object-oriented programming environment is chosen for its support of using model abstraction and creation methods as described in chapter 3. Figure 7 shows the possible activities in the CESS environment with the support from DEVS-SCHEME and ESP-SCHEME.
5. Implementation of CESS

Having presented the design specifications for CESS in chapter 4, we describe its implementation in this chapter. A case study is presented at the end to illustrate the usefulness of CESS.

5.1 Implementational Issues

The essential points to the implementation of CESS are specified in the following sections. Zeigler [20] described in detail the class inheritance structure in DEVS-SCHEME as shown in Figure 8. TRNSYS_COMPONENTS and TRNSYS_SYSTEMS are two new classes for CESS. All nodes in Figure 8 have corresponding classes available in CESS. The TI-SCHEME command (DESCRIBE name_of_class) shows the descriptions of the class, name_of_class. As an example, (DESCRIBE TRNSYS_SYSTEMS) displays on the screen all class and instance variables associated with TRNSYS_SYSTEMS, as well as the methods operating on these variables. The implementational issues are presented in terms of the activities being performed in CESS as shown in Figure 7.

5.1.1 Classes Models and Processors

All classes in DEVS-SCHEME are subclasses of the universal class ENTITIES which provides tools for manipulating objects in these classes (these objects are thereafter called objects). Models and processors are two main subclasses, providing the basic constructs for modelling and simulation.
MODELS is further specialized into ATOMIC_MODELS and COUPLED_MODELS. TRNSYS_COMPONENTS is the atomic level of a CESS model. It inherits the properties embedded in ATOMIC_MODELS which implements the DEVS formalism developed by Zeigler [16, 18].

TRNSYS_SYSTEMS, on the other hand, represents the model composition constructs. TRNSYS_SYSTEMS is a subclass of DIGRAPH_MODELS. An instance of DIGRAPH_MODELS specifies a finite set of explicitly given components with explicitly specified couplings.

PROCESSORS containing simulators, co-ordinators, and root-co-
ordinators carry out the pseudo-simulation of TRNSYS systems. These simulation classes can be used to perform real simulations of discrete event systems. A simulator is paired with an instance of class TRNSYS_COMPONENTS, and a co-ordinator is associated with an instance of the class TRNSYS_SYSTEMS.

A root-co-ordinator, however, governs the overall simulation and is linked to the co-ordinator of the most abstract (the highest) coupled model (being of class DIGRAPH_MODELS or TRNSYS_SYSTEMS).

Associated with TRNSYS_COMPONENTS are some class and instance variables specific to CESS. They are:

CLASS VARIABLES:

  LST -- list of instances made in this class, inherited from ENTITIES.
  COUNTER -- Number of instances made in this class, this also acts as the unique unit number assigned to TRNSYS system components.

INSTANCE VARIABLES:

  NAME -- The name of the TRNSYS system component, inherited from ENTITIES.
  PORTS -- a compound variable which has four parts, INP-VARS, OUT-VARS, PAR-VARS and DER-VARS. INP-VARS is a list of input variables of a TRNSYS system component; OUT-VARS is a list of
output variables of the component; PAR-VARS is a list of PARAMETERS of the component; DER-VARS is a list of derivatives of the component. Each variable has a name and an initial value for it. A symbol '?' denotes no initial value is defined.

TYPE -- The TYPE number of the component.
UNIT -- The unique UNIT number of the component.

For the class TRNSYS_SYSTEMS, the following variables are designed for CESS:

INSTANCE VARIABLES:

NAME -- the name of the TRNSYS system, inherited from ENTITIES.
INI -- Initialization flag specifying whether the model has been initialized.
CHILDREN -- a list of system components belonging to this instance, inherited from COUPLED-MODELS.
CONSTANTS -- A list of constants, each of which is a 3-letter name and a value as described in section 4.2.
COUPLINGS -- All inputs of all instances need to be connected. The coupling to an input is
defined as a pair or a constant (the initial value of the input). A pair is a UNIT number and an OUTPUT number. For each instance, the coupling of its INPUTs are listed sequentially. A list of couplings for all instances is available. The format of couplings is:

Coupling List: (( unit 1 ) ... ( unit i ) ... )
UNIT i : (( in 1 ) ... ( in j ) ... )
IN j : (UNIT# OUTPUT#) or (CONSTANT)

Figure 9 shows the definitions of TRNSYS_COMPONENTS and TRNSYS_SYSTEMS written in DEVS-SCHEME. MIXINS specifies the inheritance relationships. For instances, (MIXINS ATOMIC_MODELS) specifies that TRNSYS_COMPONENTS inherits everything that ATOMIC_MODELS has. Three options determine whether the variables can be assigned with initial values, be readable or be modifiable.

(define-class TRNSYS_COMPONENTS
classvars 1st counter)
(instvars name port type unit )
mixins atomic-models)
(options gettable-variables
settable-variables
inittable-variables
))

(define-class TRNSYS_SYSTEMS
classvars 1st)
(instvars name constants ini children Couplings)
mixins digraph-models)
(options gettable-variables
settable-variables
inittable-variables
))

Figure 9. Class Definitions
5.1.2 TYPE Operations

A TYPE in TRNSYS can be defined with the command:

(DEFINE_TYPE) or (DEFINE-TYPE)

This command will prompt the user for the name of the TYPE to be defined. The existence of this TYPE definition is then checked prior to prompting the user for detailed contents. The options to abort this process, override the previous definition or update the old one, are provided if the specified TYPE exists. The contents of a TYPE definition is described in section 4.5.1.

To update an existing TYPE, the command (UPDATE_TYPE) or (UPDATE-TYPE) can be called.

With this command, the user can modify a previously entered TYPE definition. All modification commands are listed on the screen at the time the command is invoked. The basic commands are:

ADD --- adding some more variables or flags
DEL --- deleting some variables
MOD --- updating the variable names or values
SEE --- show definition on the screen
COM --- show this list
END --- end of modifications

The user will have a final chance to decide whether this modification should be saved to replace the previous definition of this TYPE.
If the specified TYPE hasn’t been defined yet, the DEFINE process is called instead.

All TYPE definitions are stored in TBASE. The available types can be listed by the command (DIR-TBASE).

5.1.3 Define TRNSYS System

To define a TRNSYS system in CESS, several steps are needed:

STEP 1. Define the name of the system.

The command for this action is (MK-SYSTEM name_of_system). For example, we can enter (MK-SYSTEM SWHS) to define SOLAR-WATER-HEATING-SYSTEM (SWHS).

STEP 2. CHOOSE all TYPES needed for the system just made.

As a matter of fact, this command can be invoked before STEP 1. The command used for choosing TYPES is:

(CHOOSE-TYPE type1 type2 ... typen) or
(CHOOSE_TYPE type1 type2 ... typen).

This command will look through TBASE and restore TYPE information into CESS. A default instance in class TRNSYS_COMPONENTS with name name_of_type0 will be made to hold the TYPE information, making an efficient access to a given TYPE information. For example, (CHOOSE-TYPE HEATER) will establish the relevant information of HEATER in CESS. HEATER0 contains state information about HEATER.
(define-class SYS_INFO
  (classvars )
  (instvars unit-id Ainp Aout Apar
    Ader Dinp Dout Dpar Dder )
  (options gettable-variables
    settable-variables
    inittable-variables )
))

Figure 10. Class Definition of SYS_INFO

The management information about the TYPE, such as whether an instance of this TYPE permits deleting or adding INPUT variables, is recorded in an instance of a class, SYS_INFO, with the same name as TYPE. Figure 10 is the class definition for SYS_INFO.

Taking the above example, (CHOOSE-TYPE HEATER) will create an instance of SYS_INFO. The instance is named HEATER. Managing information about HEATER, the FLAGS for example, is available by entering the command (DESCRIBE HEATER).

STEP 3. Define TRNSYS system components (instances).

The command (DEF-INSTANCES class_name #_of_instances) is designed for making instances of the specified TYPE, class_name. For example, (DEF-INSTANCES 'HEATER 2) creates HEATER1 and HEATER2 as two components for a specific TRNSYS system. Both HEATER1 and HEATER2 have the same information as in TYPE HEATER. Modification on them is possible by using some commands given later. DEF-INSTANCES updates managing information, e.g., it increments by two the COUNTER in class TRNSYS_COMPONENTS.
5.1.4 Coupling Specification

After defining all system components, the system configuration can be specified. The command (CONSTRUCT-COUPLING system_name) or (CONSTRUCT_COUPLING system_name) looks through all instances made for a TRNSYS system and asks the user to enter the connections of all input variables for each instance. (0 0) can be entered to signify that this input variable has no connection to other instances.

All constants, which are values for other variables (except output variables, because initial values for output variables don't make sense here), will be asked for a number, or a simple arithmetic expression in terms of some numbers and other constants described in section 4.2.

Generator is an instance model in class TRNSYS_COMPONENTS. It generates constants to all the instance models made above, if variable values (except OUTPUTS) of these models are defined in terms of constants. Thus, the pseudo-simulation structure has all instance models and the Generator as the sub-models. Generator is created at the end of the execution of this command.

Much consistency checking is done in this command. The most obvious one is that UNIT numbers specified in coupling pairs must match an existing instance. Some other considerations are: A variable (except OUTPUT) must have an initial value; a constant must be defined before being used; etc.
In addition, some special treatment of TYPE of PRINTER is considered. All input variables need not be connected to some outputs and are treated as labels whose values have the prefix $.

5.1.5 DEVS Simulation Structure and Pseudo-Simulation

The command (CONSTRUCT-DEVS name_of_system) looks into the coupling definition for the TRNSYS system, name_of_system, and constructs a DEVS simulation structure which is the base for creation of an Entity Structure and a Model. As the result of the command, a hierarchical simulation structure can be displayed on the screen, and the pseudo-simulation is performed at the same time. Current pseudo-simulation is designed in such a way that the generator keeps sending constant values to instance models where the constant definitions are given.

As an example, (CONSTRUCT-DEVS SWHS) will set up the desired simulation structure for the Solar Water Heating System and then perform the pseudo-simulation.

5.1.6 DECK File Creation

The command (CREATE_DECK system_name) or (CREATE-DECK system_name) will prompt the user for information necessary for the actual simulation using TRNSYS. The simulation information needed is described in section 4.2.

The DECK file is stored in DECK_BASE and can then be transferred
to the TRNSYS interpreter for the simulation. The list of all DECK files is available using command (DIR-DBASE).

5.1.7 SES Creation

This function is one of the DEVS-SCHEME facilities. It extracts information from the DEVS system simulation structure and translates it into an entity structure which is stored in the Entity Structure Base (ENBASE). The command for this function is (CREATE-ENT). It assumes that CONSTRUCT-DEVS has been invoked so that the DEVS system structure is accessible from the default root-co-ordinator, SIMULATION-SYSTEM.

The name of the entity structure will be E:system_name which contains all component state information, as well as the system configurations (or couplings).

To show the entity structure, the command (PE E:system_name) can be used. The command (DIR-ENBASE) provides a list of entity structures in ENBASE.

An entity structure can be loaded from ENBASE using the DEVS facility (LOAD-ENTSTR name_of SES).

5.1.8 Model Creation

Each system component has been linked into a TRNSYS system during the above descriptions. The corresponding DEVS model exists in core memory. CESS has an intelligent facility to generate executable code
for these models. This is one of the main contributions of this thesis: MODEL CREATION. All these models in terms of DEVS-SCHME statements are the images of the continuous models in TRNSYS written in FORTRAN.

These models can be reused to generate DECK files without going back to choose TYPEs, etc. The potential use of DEVS-SCHME is explored in depth in this implementation.

A specific model can be created for a system component, say PRINTER1, with the command (CR_save_model PRINTER1). The whole system, say SWHS, can be saved with the command (CR_save_system SWHS). All models created are stored in the model base, MBASE. All model names in MBASE are available using the DEVS-SCHME command (DIR-MBASE).

5.1.9 System Reusability and Construction

The advantage of a knowledge-based approach to design and implementation of simulation systems is that efficient management on system composition can be achieved with the aid of a suitable knowledge representation, the System Entity Structure. DEVS-SCHME provides abundant facilities for managing entity structures, modelling, and simulation activities. With the created models and the entity structures, we can rebuild the TRNSYS system at any time. We can then perform modifications on models, if needed. Finally, a new DECK file can be generated, and another simulation can be performed.

Besides, we can construct a new TRNSYS system by pruning general
entity structures such as the one in Figure 2, and then apply a TRANSFORMATION operation to map a pruned entity structure onto a DEVS simulation structure. This of course assumes that models have been created by CR_save_model or CR_save_system.

The command (CR_restore_system system_name) accomplishes many tasks in restoring a system that was previously saved using CR_save_system command. It accesses the entity structure created by use of the CREATE-ENT command; it invokes the DEVS command (TRANSFORM) to transform an entity structure to a DEVS simulation structure; it accesses models in MBASE; and it initiates a pseudo-simulation.

In chapter 6, we will mention some ideas for further elaboration on these concepts, and for future research on this approach of generating TRNSYS systems.

5.1.10 Batch Execution

CESS works in two execution modes: INTERACTIVE and BATCH. The default mode is INTERACTIVE which means that questions such as "What is the name of the TRNSYS system?" must be answered explicitly by the user before the next activity can begin. In BATCH mode, however, the answers to all questions are stored in a file.

The global variable ACTION_MODE has the value CONSOLE to represent the INTERACTIVE mode. The BATCH mode is specified if the value of ACTION_MODE is a file pointer (using an OPEN statement).

For example, if the input is stored in file "C:EXAMPLE" and if we
want to execute CESS in BATCH mode, the following command is needed:

```
(DEFINE ACTION_MODE (OPEN-INPUT-FILE "C:EXAMPLE"))
```

5.2 Configurations of CESS components

Figure 6 shows the CESS composition which identifies TYPE management, INSTANCE management, DECK management, INSTANCE management, Pseudo-simulation control. The following program files that are called modules accomplish the above management requirements: CLASS module, TRNSYS module, TRNSYS_SYSTEMS module, DECK module, INIT module, MODEL module, and UTILITY module.

The CLASS module contains class definitions needed in CESS. Two classes are defined, TRANS_MODEL_BASE and TRNSYS_SYSTEMS which have been described in section 5.1.2. Appendix 1 contains the source program of the CLASS module.

The TRNSYS module contains two sub-modules, the module containing all methods for TRNSYS_COMPONENTS, and the module having helpers for the methods. Appendix 2 contains the source program of methods, and Appendix 3 presents supporting functions.

The TRNSYS_SYSTEMS module contains two sub-modules, the module containing all methods for class TRNSYS_SYSTEMS, and the module giving support for the methods. Appendix 4 lists the methods, and Appendix 5 provides the supporting procedures.
The INIT module contains the definition for class SYS_INFO which has been described in section 5.1.3. The procedures and functions to define and update TYPE information are included in this module. Appendix 6 is dedicated for the programs of the INIT module.

The MODEL module holds all procedures to generate models and retrieve models. These procedures are collected in Appendix 7.

The UTILITY module maintains all utility procedures such as the switching function for INTERACTIVE and BATCH modes, functions to interface with the user, CESS debugging tools, etc. Detailed information is given in Appendix 8.

The DECK module contains all functions needed to generate TRNSYS deck files for actual simulations. Appendix 9 lists the relevant procedure details.

5.3 Case Study - Solar Water Heating System

In Figure 5, the configuration of the solar water heating system was presented. In this section of the thesis, a detailed study is shown. Figure 5 depicts the detailed configuration of the Solar Water Heating System.

5.3.1 SWHS Component Descriptions

This system consists of a flat-plate collector, a controller which activates a pump, and a constant temperature water supply as shown in
Figure 5. To analyze the system performance, the following TRNSYS TYPEs are needed:

1) a TYPE 9 data card reader for data input
2) a TYPE 1 flat-plat solar collector
3) a TYPE 2 controller
4) a TYPE 3 pump
5) a TYPE 24 integrator
6) a TYPE 16 solar radiation processor
7) two TYPE 25 printers
8) a TYPE 26 plotter

The connections among these system components are shown in Figure 5. Some TYPEs like the Solar Collector have several working modes if used in simulation. In this example, the solar collector works in mode 1, but the solar radiator processor works in mode 3. In Appendix 10, all TYPE definitions for the above selected TYPEs are included. These pieces of information are stored in TBASE and will be used later to generate the DECK file.

5.3.2 CESS Operations on SWHS

Following the description in section 5.1, the CESS operations are listed below, resulting in the DECK file (Appendix 11). This DECK file can then be fed to the TRNSYS interpreter to conduct the actual simulation.
Operation 1. (*MK-SYSTEM* SWHS). A TRNSYS system named SWHS is created in the CESS environment. The notification message is shown on the screen.

Operation 2. (*CHOOSE-TYPE* pump printer collector plotter card integrator controller processor). This retrieves all TYPE information into the CESS environment. Section 5.1.3 has described the effects of this command in detail.

Operation 3. (*DEF-INSTANCES* 'collector 1)
   (DEF-INSTANCES 'controller 1)
   (DEF-INSTANCES 'pump 1)
   (DEF-INSTANCES 'printer 2)
   (DEF-INSTANCES 'plotter 1)
   (DEF-INSTANCES 'integrator 1)
   (DEF-INSTANCES 'processor 1)
   (DEF-INSTANCES 'card 1)

As a result, the following instances are made available to CESS (the number in parentheses is the unit number assigned to the instance): COLLECTOR1(1), CONTROLLER1(2), PUMP1(3), CARD1(9) PRINTER1(4), PRINTER2(5), PLOTTER1(6), INTERGRATOR1(7), and PROCESSOR1(8). Any time an instance model is made, e.g., PRINTER1, a message is displayed on the screen: A newly made model PRINTER1.

Operation 4. (*CONSTRUCT-COUPLING* SWHS). The coupling specification is given at this time. The constants and the undefined variable values are also specified in this operation. Appendix 11 (the DECK file) shows clearly the configurations for the
SWHS. The detailed descriptions of this command is in section 5.1.4.

Operation 5. (CREATE_DECK SWHS). The DECK file in Appendix 11 is generated by use of this command.

Operation 6. (CONSTRUCT-DEVS SWHS). This operation builds the DEVS simulation structure and performs the pseudo-simulation in the CESS environment. This command takes the information specified in operation 4, and links all instance models (created by the DEF-INSTANCES command) to form a hierarchical model structure. The message "Set up Influence Diagram" tells that the relationships among instance models have been set up, e.g., COLLECTOR1 will receive data from PUMP1. The message "Set up Internal Coupling" tells that the detailed configurations among instance models have been established, e.g., the input variable COLLECTOR_INLET_FLUID_TEMPERATURE of COLLECTOR1 will receive data from the output variable OUTLET_TEMPERATURE of PUMP1. Finally, the hierarchical model structure is displayed on the screen. More detailed information about simulation (pseudo-simulation in the CESS) in the DEVS-SCHEME environment can be found in Zeigler [22].

Operation 7. (CREATE-ENT). An entity structure is generated and stored in ENBASE. The configuration information is associated with the entity structure as well.

Operation 8. (PE E:SWHS). The entity structure created in Operation 7 is displayed on the screen.
Operation 9. (DIR-ENBASE). A list of file names is displayed on the screen. Each of these files corresponds to an entity structure, and is stored in ENBASE. SWHS.E can be found in the list.

Operation 10. (CR_SAVE_SYSTEM SWHS). The TRNSYS system, Solar Water heating System, is then saved into the model base. The task of model creation is performed in this operation.

Operation 11. (DIR-MBASE). This command shows the names of created models such as PRINTER1.M, CARD1.M, etc.

Operation 12. (CR_RESTORE_SYSTEM SWHS). This operation first prunes the entity structure with name E:SWHS and then transforms the pruned entity structure into a simulation structure in the DEVS-SCHME environment. The previously saved models, e.g., PRINTER1.M, are restored into the CESS environment. We can thus redo the operations starting at Operation 4, 5, or 6.

The above illustrations briefly go through the activity diagram in Figure 7.

The successful simulation based on the DECK file created in CESS shows the feasibility of MODEL creation at a level of abstraction at which discrete and continuous simulations can be described with the same modelling mechanism. Cellier [1] mentioned some approaches to the merging of discrete and continuous simulation at a relatively low level of system descriptions. The approach this thesis adopted is to form a common background for two simulation directions.
6. Summary

We can view the world as composed of systems which can be described or abstracted by continuous or discrete approaches. This thesis presents a combined method by which a continuous simulation system can be connected to a discrete event simulation system. The bridge between these two disciplines is a knowledge-based simulation system which supports model creation (Top-Down), model construction (Bottom-Up), multiple model utilization, system knowledge representation, and construction of a user-friendly interface. AI techniques, modern modelling and simulation methodology, and system theory, offer a solid background for knowledge-based simulation systems.

The Controlled Environment Simulation System (CESS) is constructed systematically along the above lines. A continuous simulation system called TRNSYS can thus cooperate with DEVS SCHEME, a discrete event simulation system.

For CESS, the System Entity Structure (SES) is chosen as the knowledge representation scheme, embedded in an object-oriented programming environment (SCOOPS in TI SCHEME), and a systematical approach is adopted.
6.1 Contributions

This thesis makes the following contributions:

1. A suitable level of abstraction common to both discrete and continuous models is established. Thus two approaches can share each other's advantages.

2. A systematic approach is proposed to build a knowledge-based simulation system which interfaces discrete and continuous simulation systems.

3. A method for model creation is introduced, allowing pseudo-simulation and actual simulation to be performed at the same time (OFF-LINE). This is very useful in real time simulation and control.

6.2 Future Work

The following topics may be worthwhile a further elaboration based on this thesis:

1. Find some mapping methods for models at a lower abstraction level (code level) such that simulation and modelling can be conducted interchangeably using discrete and continuous approaches.

2. Extend CESS in order to enable the TRNSYS interpreter to directly access DECK files and perform the actual simulation (ON-LINE).

3. Similar to DEVS, derive a formalism for describing continuous systems at an appropriate abstraction level. Beyond this, a formalism
that combines discrete and continuous formalisms and is suitable for performing modelling and simulation activities can then be invented.

(4) Seek ways to construct new models based on previously created models for model optimization purposes. This can be classified as MODEL LEARNING.

(5) Experimental frame concepts [10,18] and ideas of MODEL LEARNING can merge to generate optimal models in terms of optimization criteria.

(6) Map DEVS-SCHEME with a general purpose continuous simulation language such as DESIRE rather than TRNSYS.

(7) Methods and strategies of using DEVS hierarchically, also for continuous models, can be developed and are very useful.

(8) The amalgamation of entity structures to yield one global entity structure can be further studied and implemented. Some information is given in Zeigler [18].

(9) The idea of managing TYPEs of TRNSYS can be further explored. In CESS, SYS_INFO holds the management information about TYPEs, such as whether new input variables can be added to or deleted from an instance model of a TYPE. This kind of information is called META level knowledge about model CLASSES (TYPEs in TRNSYS). We need efficient and complete methods for management of META level knowledge about model classes.
APPENDIX 1. CLASS MODULE

Class definitions for TRNSYS_COMPONENTS

1. (define-structure ports inp-vars out-vars par-vars der-vars)
2. (define-class TRNSYS_COMPONENTS)
3. (compile-class TRNSYS_COMPONENTS)
4. (define-class TRNSYS_SYSTEMS)
5. (compile-class TRNSYS_SYSTEMS)

(define-structure ports inp-vars out-vars par-vars der-vars)

(define-class TRNSYS_COMPONENTS
  (classvars 1st (counter 1))
  (instvars name (port (make-ports))
    (type -2) type of model (a number)
    '-2' represents uninitialized
    (unit -2) global instant number
  )
  (mixins atomic-models)
  (options
    gettable-variables
    settable-variables
    inittable-variables
  )
)

(compile-class TRNSYS_COMPONENTS)

Definitions for coupled class of TRNSYS_SYSTEMS

(define-class TRNSYS_SYSTEMS
  (classvars lst)
  (instvars name
    (children '()) List of components
    (ini #false) Initialisation flag
    (constants '()) Constant definitions with the
      format : ((name value) ...).
    (Couplings '()) All couplings , the format is
      as follows : ( (--1--) (--2--) ...
      (--i--) ...).
  )
  (mixins digraph-models)
  (options gettable-variables
    settable-variables
    inittable-variables
  )
)

(compile-class TRNSYS_SYSTEMS)
APPENDIX 2. TRNSYS COMPONENTS METHODS

This module contains all methods related to TRNSYS COMPONENTS. It is to handle parameters, input variables, output variables and others.

1. (add-inp-vars Vname Value)
2. (add-out-vars Vname Value)
3. (add-par-vars Vname Value)
4. (add-der-vars Vname Value)
5. (update-vars Thisone Thename Ofvalue)
6. (delete-vars Thisone Thename)
7. (setup-TRNSYS_COMPONENTS)
8. (show)
9. (allowed? ThisName ThisFun)
10. (set-inp-vars val)
11. (set-out-vars val)
12. (set-par-vars val)
13. (set-der-vars val)
14. (get-inp-vars)
15. (get-out-vars)
16. (get-par-vars)
17. (get-der-vars)

APPENDIX 3. TRNSYS_COMPONENTS HELPERS

1. (def-instances class cnt) or (def_instances class cnt)
2. (build-instances class cnt uid id)
3. (put-to-end ele 1st)
4. (new-put-to-end ele 1st)
5. (delete-lst ele 1st)
6. (update-variables ele 1st)
7. (setup-Default class)
8. (show-instances) or (show_instances)
9. (is_model? item) or (is-model? item)
10. (model_external s e x)
11. (model_internal s)
12. (model_output s)
APPENDIX 4. TRNSYS_SYSTEMS METHODS

routines used here but defined elsewhere

UPDATE-VARIABLES
PUT-TO-END
NEW-PUT-TO-END

Methods for TRNSYS_SYSTEMS

1. (init-TRNSYS_SYSTEMS)
2. (update-vars Thisone Vname Munit)
3. (add-vars Thisone Vname Munit)
4. (delete-vars Thisone Vname)
5. (show)

APPENDIX 5. TRNSYS_SYSTEMS HELPERS

Helper Routines for TRNSYS_SYSTEMS models

1. (construct-coupling simu)
2. (coupling_right? mod) or (coupling-right? mod)
3. (fetch-all-models uid)
4. (get-Ith-unit I)
5. (setup-coupling-for INST)
6. (sure_read inst var)

The following routines are for the use of setting up pseudo-simulation structure.

1. (construct_DEVS simu) OR (construct-DEVS simu)
2. (convert_to_names lst)
3. (mk-system name) ---MACRO
4. (setup_inf_dig simu)
5. (collect_constants simu)
6. (get_constants vars)
7. (ask_for_values names)
8. (create_generator simu)
9. (generator_intfn s)
10. (generator_outfn s)
11. (generator_influencee simu)
12. (setup_int_coup_for_all simu)
13. (setup_int_coup_for_model models cnt simu)
14. (setup_int_coup_for_generator models simu)
15. (initialize_generator simu)
APPENDIX 6. INIT MODULE

1. (define-class SYS_INFO)
2. (compile-class SYS_INFO)
3. (macro CHOOSE-TYPE)
4. (define-type) or (define_type)
5. (define_type_info name)
6. (show_type_info lst)
7. (save_type_info lst)
8. (restore_type_info name)
9. (get_variables kind N_vars)
10. (update_type) or (update-type)
11. (update_type_info name)
12. (show_update_command)
13. (add_Ith_lst pos ele lst)
14. (delete_Ith_var cnt lst)
15. (modify_Ith_var cnt lst)
16. (Lmodify ele)
17. (choose-all lst)

APPENDIX 7. MODEL MODULE

0. CR_model_storage
1. CR_Model_For
2. CR_simulator_for
3. CR_TRNSYS_COMPONENTS_for
4. CR_output_S:NAME
5. CR_output_NAME
6. CR_attach_for
7. CR_class_for
8. CR_save_object
9. CR_restore_object
10. CR_restore_all_objects
11. CR_restore_model
12. CR_save_system
13. CR_restore_system
APPENDIX 8. UTILITY MODULE

All library routines are included here

1. (yes_answer? question)
2. (prompt_user question)
3. (writef con fp)
4. (writel lst fp)
5. (get_pair question)
6. (Defined? name lst)
7. (replace_flag cnt lst)
8. (Ith_ele I lst)
9. (reverse_lst lst)
10. (place_to_end ele lst)
11. (read_mode)
12. (wait_a_while n)
13. (ed sou)
14. (mk-fsl sou)
15. (remove-$ name)
16. (IN_LST? name lst)
17. (locate_lst name lst)
18. (tolower name)
19. (toupper name)
20. (get_class_name name)
21. (destroy_odd_strings lst)
22. (clear_system)
23. (dir-tbase)
24. (serial-update-inst inst)
25. (extended_number? num)
26. (update-instance inst)
27. (table? el)
28. (newdisp content wind)
29. (show-lst cnt lst wind)

APPENDIX 9. DECK MODULE

1. (Create_Deck simu)
2. (create_deck_header simu fp)
3. (create_deck_unit simu model fp)
4. (create_deck_end simu fp)
5. (N-S fp n) or (N_S fp n)
   -- output N spaces to FP
APPENDIX 10. TYPE DEFINITIONS FOR SWHS

The TYPE number is at the beginning of the TYPE definition.

(1) Card Reader

9
PARAMETERS 5
((NUMBER OF VALUES TO BE READ 5) (TIME INTERVAL OF DATA ONE)
(T3 -1) (MULTIPLE_FACTOR3 ONE) (ADDITION_FACTOR3 ZERO))
INPUTS 0
() DERIVATIVES 0
() OUTPUTS 20
((Y1 ?) (Y2 ?) (Y3 ?) (Y4 ?) (Y5 ?) (Y6 ?) (Y7 ?) (Y8 ?)
(Y9 ?) (Y10 ?) (Y11 ?) (Y12 ?) (Y13 ?) (Y14 ?) (Y15 ?)
(Y16 ?) (Y17 ?) (Y18 ?) (TIME OF LAST DATA ?)
(TIME OF NEXT DATA ?))
DELETE-FLAGS
(!TRUE () () !TRUE)
ADDING-FLAGS
(!TRUE () () !TRUE)

(2) Pump Controller

2
PARAMETERS 3
((NSTK 10) (DELTA_TEMPERATURE_HIGH 10)
(DELTA_TEMPERATURE_LOW ONE))
INPUTS 3
((UPPER_INPUT_TEMPERATURE 20) (LOWER_INPUT_TEMPERATURE 20)
(INPUT_CONTROL_FUNCTION ZERO))
DERIVATIVES 0
()
OUTPUTS 1
((OUTPUT CONTROL_FUNCTION ?))
DELETE-FLAGS
() () () ()
ADDING-FLAGS
() () () ()
(3) Pump

3
PARAMETERS 1
((MAXIMUM FLOW RATE 200))

INPUTS 3
((INLET FLUID TEMPERATURE 20) (INLET MASS FLOW RATE 200)
(CONTROL FUNCTION ZERO))

DERIVATIVES 0
()

OUTPUTS 3
((OUTLET TEMPERATURE ?) (OUTLET MASS FLOW RATE ?)
(PUMP POWER CONSUMPTION ?))

DELETE-FLAGS
((0) (0) (0) (0))

ADDING-FLAGS
((0) (0) (0) (0))

(4) Flat-Plate Collector

1
PARAMETERS 12
((COLLECTOR MODE ONE) (NUMBER OF COLLECTORS IN SERIES ONE)
(TOTAL COLLECTOR AREA 6.5)
(SPECIFIC HEAT OF COLLECTOR FLUID 4.19)
(EFFICIENCY MODE ONE) (FLOW RATE PER UNIT AREA 50)
(INTERCEPT OF EFFICIENCY CURVE 0.7)
(NEGATIVE OF EFFICIENCY CURVE SLOPE 15)
(HEAT EXCHANGER FACTOR -1)
(SPECIFIC HEAT OF LOADSIDE FLUID 4.19) (OPTICAL MODE ONE)
(INCIDENT ANGLE MODIFIER 0.1))

INPUTS 10
((COLLECTOR INLET FLUID TEMPERATURE 20)
(COLLECTOR FLUID MASS FLOW RATE ZERO)
(HEAT EXCHANGER LOADSIDE FLUID MASS FLOW RATE ZERO)
(AMBIENT TEMPERATURE 15.6) (INCIDENT RADIATION ZERO)
(TOTAL HORIZONTAL RADIATION ZERO)
(HORIZONTAL DIFFUSE RADIATION ZERO)
(GROUND REFLECTANCE 0.2)
(INCIDENT ANGLE ZERO) (COLLECTOR SLOPE 40))

DERIVATIVES 0
()

OUTPUTS 20
((OUTLET FLUID TEMPERATURE ?) (OUTLET FLUID FLOWRATE ?)
(RATE OF ENERGY GAIN ?))

DELETE-FLAGS
((0) (0) (0) (0))

ADDING-FLAGS
((0) (0) (0) (0))
(5) Integrator

24
PARAMETERS 0
()
INPUTS 2
((FIRST_QUANTITY_TO_BE_INTEGRATED ZERO) (SECOND ZERO))
DERIVATIVES 0
()
OUTPUTS 5
((INTEGRAL_OF_FIRST_QUANTITY ?) (SECOND ?) (THIRD ?)
(FOURTH ?) (FIFTH ?))
DELETE-FLAGS
(() #!TRUE () #!TRUE)
ADDING-FLAGS
(() #!TRUE () #!TRUE)

(6) Solar Radiation Processor

16
PARAMETERS 7
((RADIATION_MODE 3) (TRACKING_MODE ONE)
(DAY_OF_YEAR_OF_START_OF_SIMULATION 8)
(LATITUDE 40) (SOLAR_CONSTANT 4.871E+03)
(SHIFT_IN_SOLAR_TIME_HOUR_ANGLE ZERO)
(SOLAR_TIME_EQUALS_SIMULATION_TIME_IF_THIS_IS_ZERO -1))
INPUTS 6
((HORIZONTAL_BEAM_RADIATION ZERO)
(TIME_OF_LAST_RADIATION_DATA_READING ZERO)
(TIME_OF_NEXT_RADIATION_DATA_READING ZERO)
(GROUND_REFLECTANCE 0.2) (SLOPE_OF_SURFACE 40)
(AZIMUTH_OF_SURFACE ZERO))
DERIVATIVES 0
()
OUTPUTS 10
((EXTRATERRESTIAL_HORIZONTAL_RADIATION ?)
(SOLAR_ZENITH_ANGLE ?) (SOLAR_AZIMUTH_ANGLE ?)
(TOTAL_HORIZONTAL_RADIATION ?)
(DIFFUSE_HORIZONTAL_RADIATION ?)
(TOTAL_INCIDENT_RADIATION ?)
(BEAM_INCIDENT_RADIATION ?) (DIFFUSE_INCIDENT_RADIATION ?)
(INCIDENCE_ANGLE ?) (SLOPE ?))
DELETE-FLAGS
(() () () ()
ADDING-FLAGS
(() () () ()
}
(7) Printer

25
PARAMETERS 1
  ((PRINTING_TIME_INTERVAL 24))
INPUTS 5
  ((SOLS $SOLS) (TOUT $TOUT) (QU $QU) (TSOLS $TSOLS)
   (TQU $TQU))
DERIVATIVES 0
  ()
OUTPUTS 0
  ()
DELETE-FLAGS
  (() #TTRUE () ()
ADDITION-FLAGS
  (() #TTRUE () ()

(8) Plotter

26
PARAMETERS 1
  ((MODE ONE))
INPUTS 1
  ((SOLS $SOLS))
DERIVATIVES 0
  ()
OUTPUTS 0
  ()
DELETE-FLAGS
  (() #TTRUE () ()
ADDITION-FLAGS
  (() #TTRUE () ()

APPENDIX 11. DECK FILE FOR SWHS

* SIMULATION DECK FOR : SWHS

*** INPUT INFORMATION AND DATA READER
*

SIMULATION 0 24 1

WIDTH 72

CONSTANTS 2

ZERO = 0  ONE = 1

UNIT 1 TYPE 1 COLLECT1

PARAMETERS 12
ONE ONE 6.5 4.19 ONE 50 0.7
15 -1 4.19 ONE 0.1

INPUTS 10
3,1 3,2 3,2 9,2 8,6 8,4
8,5 0,0 8,9 8,10
20 ZERO ZERO 15.6 ZERO ZERO
ZERO 0.2 ZERO 40

*** OUTPUTS 3
* OUTLET_FLUID_TEMPERATURE OUTLET_FLUID_FLOWRATE
* RATE_OF_ENERGY_GAIN

UNIT 2 TYPE 2 CONTROL1

PARAMETERS 3
10 10 ONE

INPUTS 3
1,1 0,0 2,1
20 20 ZERO

*** OUTPUTS 1
* OUTPUT_CONTROL_FUNCTION
UNIT 3 TYPE 3 PUMP1

PARAMETERS 1
200

INPUTS 3
0,0 0,0 2,1
20 200 ZERO

*** OUTPUTS 3
* OUTLET TEMPERATURE OUTLET MASS FLOW RATE
* PUMP POWER CONSUMPTION

UNIT 4 TYPE 25 PRINTER1

PARAMETERS 1
24

INPUTS 3
0,0 0,0 0,0
SOLS TOUT QU

UNIT 5 TYPE 25 PRINTER2

PARAMETERS 1
24

INPUTS 2
0,0 0,0
TSOLS TQU

UNIT 6 TYPE 26 PLOTTER1

PARAMETERS 1
ONE

INPUTS 1
0,0
SOLS
UNIT 7 TYPE 24 INTEGRAT1

INPUTS 2
8,6 1,3
ZERO ZERO

*** OUTPUTS 5
* INTEGRAL OF FIRST QUANTITY SECOND
* THIRD FOURTH
* FIFTH

UNIT 8 TYPE 16 SOLAR1

PARAMETERS 7
3 ONE 8 40 4.871E+03 ZERO -1

INPUTS 6
9,1 9,19 9,20 0,0 0,0 0,0
ZERO ZERO ZERO 0.2 40 ZERO

*** OUTPUTS 10
* EXTRATERRESTIAL HORIZONTAL RADIATION SOLAR ZENITH_ANGLE
* SOLAR AZIMUTH_ANGLE TOTAL HORIZONTAL RADIATION
* DIFFUSE HORIZONTAL RADIATION TOTAL INCIDENT RADIATION
* BEAM INCIDENT RADIATION DIFFUSE INCIDENT RADIATION
* INCIDENCE_ANGLE SLOPE
*

UNIT 9 TYPE 9 CARD1

PARAMETERS 5
5 ONE -1 ONE ZERO

*** OUTPUTS 20
* Y1 Y2
* Y3 Y4
* Y5 Y6
* Y7 Y8
* Y9 Y10
* Y11 Y12
* Y13 Y14
* Y15 Y16
* Y17 Y18
* TIME_OF_LAST_DATA TIME_OF_NEXT_DATA
*

*** END OF DECK DATA FOR : SWHS ***
END
REFERENCES


