Compiler for the Embedded Real-Time Systems Specification Language CLEOPATRA

by Robert L. Popp
Devora Reich

Department of Computer Science

Boston University

Boston, MA 02215

Abstract

Design and implementation of embedded real-time systems is a complex and currently a major area of research. This project focuses on the implementation of one such design, the **TRA model** (i.e., Time-constrained Reactive Automata), proposed and developed by Dr. Azer Bestavros. The TRA model is implemented via the embedded real-time systems specification language **CLEOPATRA -** *C-based Language* for *Event* driven *Object-oriented Prototyping* of *Asynchronous Time-constrained Reactive Automata.* The CLEOPATRA language has been developed to provide a way to specify TRA objects. The interconnection of multiple TRA objects via channels denotes an entire embedded system. A compiler for the specification language CLEOPATRA is the focus of this work.

This project involved the design of a compiler which takes a CLEOPATRA *source specification* file (e.g., **<file>.cleo)** and generates a C-based simulator that simulates an embedded real-time, event-driven reactive system. The CLEOPATRA language is C based, and therefor follows a similar syntax to that of the C language. In the construction of the CLEOPATRA compiler, the UNIX software tools *lex* and *yacc* were utilized. These utilities provided a clean and easily extendible format for the CLEOPATRA grammar.

email address: rpopp@cs.bu.edu

1 Introduction

Embedded systems have become a critical part of the computer science world. The design and implementation of such systems is very complex and currently a major area of research. This project focuses on the implementation of one such design, the **TRA model** (i.e., Time-constrained Reactive Automata) proposed by Dr. Azer Bestavros. The TRA model is implemented via the embedded real-time systems specification language **CLEOPATRA -** *C-based Language* for *Event* driven *Object-oriented Prototyping* of *Asynchronous Time Constrained Reactive Automata.* A compiler for the specification language CLEOPATRA is the focus of the following work.

The TRA model uses individual TRA objects to construct an embedded system. Each TRA object is its own reactive entity. Connected by input and output channels, these entities come together to form one unit, an embedded real-time system. The TRA objects interact by responding to inputs signaled on their input channels, and signaling output on their output channels. The model is nondeterministic with regard to time control and computation. Inputs and outputs signaled on channels are known as events. The events which trigger computations make up the time constrained causal relationships which determine the behavior of such a system. [Bestavros - PhD thesis, Chap 1]

This project involved the design of a compiler which takes a CLEOPATRA *source specification* file (e.g., **<file>.cleo)** and generates a C-based simulator. There are three source files to the simulator, a *header* file **<file>.h,** a *schema* file **<file>.s,** and a *C-based source* file **<file>.c.** The header file contains type definitions for all defined TRA classes in the CLEOPATRA source specification file. The schema file includes function definitions (e.g., *commit(), trigger(), & clean()* functions for each of the TRA classes defined) which are used in the TRA actions. And finally, the C-based source file contains simulator initializations, TRA object creation routines, and the basic structure of the actual simulator itself (e.g., *main()* routine definition). Upon successful compilation of the CLEOPATRA source specification file **<file>.cleo** by the CLEOPATRA compiler, the C-based source file **<file>.c** is fed to the GNU gcc compiler where an executable file comprising the simulator itself is generated. [Bestavros - PhD thesis, Chap 6]

In the construction of the CLEOPATRA compiler, the UNIX software tools *lex* and *yacc* were utilized. These utilities provided a clean and easily extendible format for the CLEOPATRA grammar. Tokens and keywords are defined in the lex specification file (i.e., **cleo.l),** where this file is compiled by the lex utility to produce the CLEOPATRA lexical analyzer. The CLEOPATRA grammar is defined in the yacc specification file (i.e., **cleo.y).** Yacc then generates a LALR (bottom up) parser written in C for the defined CLEOPATRA grammar. Error handling is done at two levels: syntax errors are caught both by the lexical analyzer (e.g., via unrecognized tokens) and the yacc generated parser (e.g., via the yacc pre-defined error token - **error -** from within the grammar); and semantic errors are caught during the code generation phase (e.g., via our own written error handling routines).

2 TRA Description

The CLEOPATRA language has been developed to provide a way to specify TRA objects. The interconnection of multiple TRA objects via channels denotes an entire embedded system. TRA objects have two main components, channels and state variables. **TET's** (i.e., Time Constrained Event Driven Transactions) are used to describe the causal relationship found between different channels, and the computations or state transitions that they trigger. In other words, TET's are used to specify the behavior of the given defined TRA object.

The CLEOPATRA language is C based, and therefor follows a similar syntax to that of the C language. A system description is characterized by the definition of TRA objects followed by the *main* TRA object which gives an instantiation to all of the other TRA objects. Each TRA object is specified by a **TRA_class** description in the CLEOPATRA language. A TRA_class definition consists of the TRA header followed by the TRA body. See Figure 1 for an example of the Factorial function defined within the CLEOPATRA syntax.

```
TRA_class fact()   request(int)  →  result(int)
{
      state:
        int x = -1, f = 0;

      act:
        request(x)  → :
        commit  { f = 1; }

        → :
        unless ( x <= 0 )
          commit { f = f * x; x--; }

        →  result(f):
        unless( x != 0 )
          commit { x--; }
}
```

Figure 1. The TRA definition for **TRA_class** fact. Note all keywords within the CLEOPATRA specification language are denoted in boldface print. For the TRA definition for fact, the header consists of the keyword **TRA_class,** followed by the identifier *fact,* and followed by its signature - input channel *request* (taking an int argument) and output channel *result* (also taking an int argument). The body of the *fact* TRA consists of its state variables located in the state section of the TRA definition denoted by the keyword **state:,** and its TETs specifying its behavior located in the actions section of the TRA definition denoted by the keyword **act:.**

3 TRA Header

The format of the TRA header is as follows:

TRA_class *Identifier* (TRA_optional_parameters) signature

The keyword **TRA_class** is followed by an *Identifier* naming the TRA class. If the TRA is parameterized, the parameter names and types are enclosed within the parentheses. In addition to the standard C types (e.g., int, float, double, etc...), the CLEOPATRA language also supports the Boolean data type (i.e., denoted by keyword **bool),** and the Unit data type (i.e., denoted by keyword **unit),** which takes on only the single value 0. When a TRA object is instantiated, the values of the actual parameters must obviously be used, and therefor, they must be previously defined (e.g., via the **#define** directive) or actual values. The final part of the TRA header is the external signature for the TRA class. The external signature consists of a list of input channels followed by the keyword symbol denoting a signature: →**,** which is then followed by a list of output channels. Input and output channels are denoted by the channel identifier followed by the channel type. Channel types are used to indicate the signalling range of values triggered on those channels. For example, if a channel has type double, then any value signalled on that channel will be of type double. If no channel type is explicitly declared, the default is the **unit** type. All TRA's have at least one input channel known as the *start* channel. This channel may be declared in the external signature, however, if it is not, it is implicitly assumed. If the *start* channel is explicitly declared, it must be triggered by an event from another TRA. However, if it is not declared, the initialization of the system automatically triggers it. More about this will be discussed later. See Figures 2 and 3 for some further examples of TRA headers.

TRA_class integrate(**double** TICK**,** TICK_ERROR)
in(**double**) → out(**double**)

Figure 2. This header for **TRA_class** *integrate* has two parameters - TICK and TICK_ERROR - of type double, two input channels *start* and *in,* and one output channel *out.* Note that the parameters to the **TRA_class** *integrate* could have been specified as: (double TICK; double TICK_ERROR).

TRA_class *ramp*() start() → out(**double**)

Figure 3. This header for **TRA_class** *ramp* has zero parameters, one input channel *start,* which is explicitly stated, and only one output channel *out* with channel type double.

4 TRA Body

The TRA body, which, like C functions, is enclosed in curly braces, has three main components. A *declarations* section, an *initializations* section, and a *transactions* section. These three parts come together to govern the behavior of the particular TRA class being defined. TRA classes can be classified on the basis of their complexity into two main groups: those which are termed *basic,* and those which are termed *composite.* Basic TRA's are those consisting of only their own class definitions, whereas, composite TRA's are those that are constructed by the inclusion of other TRA classes (i.e., the composition of two or more TRA classes).

4.1 Declarations Section

The declarations section of a TRA class consists of three sections all of which may or may not be needed for a particular TRA class. The first component is the state declarations section, denoted by the keyword **state:.** Next is the internal channel declarations section, denoted by the keyword **internal:.** And finally, following the internal section is the included declarations section, denoted by the keyword **include:.**

4.1.1 State: Section

The state declaration section begins with the keyword **state** followed by a colon. Then, legal C-style declarations for variables are specified. Note that initialization of variables as they are being declared in this section are allowed (similar to C-style variable declaration semantics). These variables, or states as we prefer to refer to them, will be used internally in the given TRA class. Note that legally within the TRA semantics, all states can only be referred to within the context of the given TRA class definition, and any reference to a TRA's states from outside the class definition violates the semantics. And finally, the section concludes with a semicolon. See Figure 4 for an example of a state declarations section.

```
TRA_class fifo(int N)
      in(float) → out(float), overflow(), ack()
{
     state:
        int i, j;
        float y;
        bool f;
}
```

Figure 4: The **TRA_class** *fifo* definition containing only a **state:** section in its declarations section. In the **state:** section are declarations for four states: *y,* of type float; *i,* and *j,* of type int; and *f,* of type bool.

4.1.2 Internal: Section

The internal section is used to declare channels which are used only within the particular TRA object being defined itself, as opposed to external channels which are used for interactions between different TRA objects. This section consists of the keyword **internal** followed by a colon. Then follows a signature specification (i.e., input and output channel definitions) which has similar structure syntactically as the external signature specification of the TRA header described above. The section concludes once again with a semicolon. See Figure 5 for an example of the **internal:** section in a TRA class definition.

```
TRA_class main() →
{
    internal:
      input(int) → output(int), done();
}
```

Figure 5: The **TRA_class** *main* definition containing only an **internal:** section in its declarations section. In the **internal:** section are declarations for three internal channels: *input,* an input channel of type int; *output,* an output channel of type int; and *done,* an output channel of type unit.

4.1.3 Include: Section

The final declarations section is the include section. It is here where composite TRA classed are formed. The keyword **include** immediately followed by a colon opens the section. Then comes the included TRA objects each of which has the following form: the name of the included TRA class followed by any necessary parameter instantiations (i.e., if the TRA has parameters, they must be instantiated here), followed by the external signature of the included TRA. The signature specification here differs from that in the TRA header itself in that only the channel names are specified and not their channel types. The **include:** section also concludes with a semicolon. Note that according to TRA semantics, any included TRA object in a given TRA class definition must be previously defined and specified, otherwise an error occurs. See Figure 6 for an example of the **include:** section in a TRA class definition.

```
TRA_class main() →
{
    include:
      square()  in() → w1();
}
```

Figure 6: The **TRA_class** *main* definition containing only an **include:** section in its declarations section. In the **include:** section is a declaration for one included TRA object: *square,* with external signature input channel *in* and external signature output channel *w1*.

4.2 Initialization Section

After the declarations portion of a TRA class definition, an optional initialization section may follow. The section begins with the keyword **init** followed by a colon. This section contains C style initializations of state variables. The end of the section is signaled by a semicolon. Note that most simple initializations of state variables can be accomplished initially when they are declared in the **state:** section and don't have to be initialized in the **init:** section of a TRA class definition. Any legal C-style variable initialization occurring during its declaration is allowed within CLEOPATRA semantics. See Figure 7 below for an example of an **init:** section for a TRA class definition.

```
TRA_class sync2(double TMIN, TMAX)
    in0(), in1() → sync()
{
    state:
      bool flag0, flag1;

    init:
      flag0 = FALSE;
      flag1 = FALSE;
}
```

Figure 7: The **TRA_class** *sync2* definition containing both a **state:** section and an **init:** section in its TRA body. In the **state:** section is a declaration for two boolean states of type bool: *flag0* and *flag1*. In the **init:** section is the initialization of two states: *flag0* and *flag1* both initialized to the boolean value of false.

4.3 Transactions Section

The last and most complex section of the TRA class definition is the transactions section. Here is where the **TET's,** which specify the Time-constrained Event-driven Transactions, are defined. The section begins with the keyword **act** immediately followed by a colon. Then, the TET definitions follow. There are many different possible types of TETs. Their common bond is that they all describe a response of a TRA to some type of event or set of events - basically, the triggering or firing of actions along their specified channels. A particular TRA may have many TET's or none at all, depending on its purpose. Although there are many different types of TET's, however, they all follow the same basic format. The TET can be decomposed into two parts, a *header* or *trigger* section, and a *body* or *reaction* section. Next we describe each of these sections in more detail.

4.3.1 TET Header

The TET header specifies a set of *triggering* channels, followed by the reserved symbol →**,** followed by at most one *control* channel. There may be zero or more triggering channels in the trigger section, each of which may be associated with at most one state variable. That is, each trigger channel consists of a channel identifier followed by parenthesis which may or may not enclose a state variable. It is in this state variable that the value of the trigger is recorded when signalled. A TET which contains only a trigger section would have the following format:

*Channel-Identifier***(** *State-Identifier* **)** → **:**

In this case, every time an action on *Channel-Identifier* is signaled, its value will be stored in the state *State-Identifier.* It is also possible to have a TET with no specified trigger section. In this case, the triggering channels are considered to be the same as the set of all TRA channels defined in the TRA class itself, and if at any time one of these channels is signaled, it triggers the TET without the trigger section. Such a TET may be useful for specifying an iterative fixed point computation. [Bestavros - PhD thesis]

The second part to the TET header is the fire section. The fire section specification consists of at most one (i.e., zero or one) channel. Here is where the action value to be signalled on the controlled channel is specified as a result of firing the TET. Any expression on the state of the TRA is a possible value. If only a channel name, and not an expressioned-parameterized channel is specified, a random value is signaled. Of course this must be from the signaling range of the controlled channel as determined by its channel type.

4.3.2 TET Body

The body of the TET contains possible reactions to the TET triggers. There are three possible components of a TET reaction: a *disabling condition,* a *time constraint,* and a *state transformation schema.* Most combinations of these three components form a legal TET action.

The purpose of the disabling condition is to disable or abort an intended reaction under certain specified conditions. It follows the form:

unless (Boolean-Expression **),** or

while (Boolean-Expression).

In the case of the **unless** clause, the forthcoming reaction will not occur if the *Boolean-Expression* becomes true at any point after the triggering has occurred. In the case of the **while** clause, its associated action is disabled if the expression ever becomes false.

The time constraint places an *upper* and *lower* bound on the time permitted to elapse between the scheduling of an action and its commitment. An *open* time frame will have the form:

before Constant-Expression, or

after Constant-Expression,

meaning the time bound is one sided. On the other hand, a *closed* time frame has a much stricter timing constraint than that of the open time frame. The closed time frame has the following form:

within [Lower-Bound ˜ Upper-Bound **].**

This provides a time constraint at both the lower and upper ends. If there is not a stated time constraint, a lower bound of zero and an upper bound of infinity are assumed. That is, the cause and effect relationship between the trigger and the reaction is not bound by time.

The state transformation schema is of the form:

commit { C-code **}.**

It gives a method for computing the next state of the TRA once the reaction has been committed. All **commit** clauses are assumed to be atomic, that is, all or none of the clause executes without interruption, and it is assumed to occur instantly. The token *C-code* is used to denote the method of the **commit** clause, and it should be noted that this token simply stands for any legal C code. If there is no **commit** clause, no state change results from committing the reaction. [Bestavros - PhD thesis, Chap 4] See Figure 8 for an example of a TET, assuming a given TRA class definition.

```
act:
    in(x), out() → out(y):
      within [ DELAY-EPOCH ˜ DELAY+EPOCH ]
        unless ( x < y )
            commit { int v; v = x; x = y; y = v; }
        unless ( x >= y )
            commit { x = y; }
```

Figure 8. A TET definition for a given TRA class definition. In the TET header, there are two input triggering channels in the trigger section - *in,* and *out,* where state variable *x* will record the value of the trigger when channel *in* is triggered. The header also specifies one output channel in the fire section - *out,* where the action value of this channel is given by *y.* The body of the TET consists of a closed time frame timing constraint specified by the **within** clause, with lower bound DELAY-EPOCH and upper bound DELAY+EPOCH. Two disabling conditions correspond to the timing constraint, specified by the two **unless** clauses, and with each **unless** clause is an associated state transformation schema **commit** clause with its C-based specified method within the curly braces.

5 CLEOPATRA Example

Lets now look at a complete example to understand more clearly some of the previously described components of a TRA class definition. In the figure below, I give once again the TRA class definition for the factorial function, along with a main TRA class definition. Refer to this figure in the following discussion.

```
TRA_class fact()  request(int)  →  result(int)
{
      state:
        int x = -1, f = 0;

      act:
        request(x)  → :
        commit  { f = 1; }
        → :
        unless ( x <= 0 )
          commit { f = f * x; x--; }
        →  result(f):
        unless( x != 0 )
          commit { x--; }
}

TRA_class main()  →
{
      state:
        int n, f;

      internal:
        output(int)  →  input(int),  done();

      include:
        fact()  input()  →  output();

      act:
        start(),  done()  →  input(n):
          commit { printf("n = "); scanf("%d",&n); }
        output(f)  →  done():
          commit { printf("fact = %d", f); }
}
```

In the above figure, the TRA specifications for both *fact* and *main* are given, where we can use this system to compute the factorial function. For the TRA specification for *fact,* the header consists of the keyword **TRA_class** followed by the identifier *fact.* In this case the class is not parameterized, so the identifier *fact* is simply followed by an opening and closing parenthesis. The signature of *fact* has one explicitly defined input channel - *request* - and one output channel - *result.* The signalling range of these channels is on the set of integers. Although it is not explicitly declared, *fact* also has the *start* input channel as part of its external signature, and whose signaling range type is the unit type.

The open curly brace denotes the start of the TRA body which in this example has two parts. In the **state:** section, two states - *x* and *f* - both of type int, are declared. They are both initialized and declared in the same location. The **init:** and **include:** sections are not needed in this TRA class definition.

Under the **act:** section, three TETs are used to define the behavior of the TRA class. In the first TET, when the *request* channel is triggered, the value signaled on the channel is stored in the state *x,* and in the state transformation schema, *f* is assigned the value one. The second TET is triggered the first time by the first *request(x)* action. Thereafter, this TET will continue to trigger, fire, and retrigger itself until it reaches a point where the value of state *x* is less than one. The state *f* acts as an accumulator in this case, and when this TET ceases to fire, it contains the correct value of the factorial function. The third TET is then triggered, and the result is recorded in state *f.* Here the disabling condition is **unless** (x != 0), (i.e., state *x* has not decremented enough and the result in *f* is incorrect). In this case, the TET should not fire. The TRA class definition concludes with the right curly brace.

For the TRA specification for *main,* it has no explicitly declared external channels, however, it does have the implicit external channel *start* with unit channel type. It also has a **state:** section with states *n* and *f* both of type int. Unlike its counterpart *fact,* TRA class *main* has both and **internal:** and an **include:** section. In its internal section, *main* declares three channels: input channel *output* with channel type int; and output channels *input* and *done,* with channel *input* having channel type int and channel *done* having channel type unit. In its include section it includes one TRA object of TRA class fact. And similar to the *fact* TRA class, the *main* TRA class defines TETs in its **act:** section.

To compile and run the simulator for this CLEOPATRA source specification file, assume the definitions for the TRAs *main* and *fact* given above are in a file called **factorial.cleo** (note: in order to compile a CLEOPATRA source specification file, the file must be suffixed with **.cleo).** Also assume the CLEOPATRA compiler is in an executable file called **ccleo.**

To compile, type at the UNIX prompt: **ccleo factorial.cleo**
The CLEOPATRA compiler will generate four files in the working directory:

1. cleo.h /* simulator definitions, globals, and types */
2. factorial.h /* type definitions for TRA_class definitions */
3. factorial.s /* schema definitions for TRA_class definitions */

4. factorial.c /* TRA object creation for TRA_class definitions */

To run the simulator, compile, via GNU's **gcc** compiler, the file factorial.c. Now simply run the executable version for the compiled file factorial.c and you will be running the simulator.

6 CLEOPATRA Grammar

Below is a **BNF** grammar for the CLEOPATRA language. Note that the terminals (tokens) are indicated in boldface text, while the nonterminals (productions) are indicated in italicized text. Also note that the empty production is indicated by the symbol $\in$.

program $\rightarrow$
    definitions

definitions $\rightarrow$
    tra_definition
    | *definitions tra_definition*

tra_definition $\rightarrow$
    tra_header **{** *tra_body* **}**

tra_header $\rightarrow$
    TRA_class id (*tra_opt_params* **)** *signature*

tra_opt_params $\rightarrow$
    tra_param_list
    | $\in$

tra_param_list $\rightarrow$
    param_list
    | *param_list* **;** *tra_param_list*

param_list $\rightarrow$
    type param_names

param_names $\rightarrow$
    id
    | **id ,** *param_names*
    | *array_type*
    | *array_type param_names*

array_type $\rightarrow$

id [num_int]
| id [id]

signature →
opt_ch_specs → *opt_ch_specs*
| *opt_ch_specs* →
| → *opt_ch_specs*
| →

opt_ch_specs →
opt_ch_spec
| *opt_ch_spec* **,** *opt_ch_specs*

opt_ch_spec →
id (*type*)
| **id** ( )

type →
int | id | bool | unit | double | float
| **enum** { *param_names* }

tra_body →
opt_declar opt_init opt_transactions

opt_declar →
opt_state opt_internal opt_included

opt_state →
state : *state_var_defs*
| ∈

state_var_defs →
state_var_def
| *state_var_def state_var_defs*

state_var_def →
type var_list_defs **;**

var_list_defs →
var_list_def
| *var_list_def* **,** *var_list_defs*

var_list_def →
id *opt_const_expr*

| *array_type*

opt_const_expr →
= *const_exp*
| ∈

opt_internal →
internal : *signature* **;**
| ∈

opt_included →
include : *included_objs*
| ∈

included_objs →
included_obj
| *included_obj included_objs*

included_obj →
tra_instant **;**

tra_instant →
id *optional_actual_params ext_bindings*

optional_actual_params →
( )
| **(** *actual_params* **)**

actual_params →
const_exp
| *const_exp* **,** *actual_params*

ext_bindings →
opt_ch_list → *opt_ch_list*
| *opt_ch_list* →
| → *opt_ch_list*
| →

opt_ch_list →
id ( )
| **id ( ) ,** *opt_ch_list*

opt_init →
init : *assign*

| ∈

assign →
id **=** *expression* **;**
| **id** **=** *expression* **;** *assign*
| *array_type* **=** *expression* **;**

opt_transactions →
act **:** *xacts*
| ∈

xacts →
xact xacts
| ∈

xact →
xact_header **:** *xact_body*

xact_header →
opt_trigger_list → *out_sig_spec*

opt_trigger_list →
trigger
| *trigger* **,** *opt_trigger_list*
| ∈

trigger →
id **(** *array_type* **)**
| **id** **(** **id** **)**
| **id** **(** **)**

out_sig_spec →
id **(** *opt_exp* **)**
| ∈

xact_body →
acts

acts →
act
| *acts* *act*

act →
fire_acts

| *opt_cond fire_acts*
| *opt_cond opt_time fire_acts*
| *opt_time fire_acts*
| *opt_time opt_cond fire_acts*

fire_acts →
{ *C-code* **}**
| *computation*

computation →
commit { ***C-code* **}
| **do { ***C-code* **}**

opt_cond →
unless (*expression* **)**
| **while (** *expression* **)**

opt_time →
closed_time_frame
| *open_time_frame*

closed_time_frame →
within [*expression* ~ *expression* **]**

open_time_frame →
before *expression*
| **after** *expression*

opt_exp →
expression
| ∈

expression →
simple_expression
| *simple_expression* **relop** *simple_expression*

simple_expression →
term
| *sign term*
| *simple_expression* **addop** *term*
| *simple_expression* || *term*
| *simple_expression* | *term*

term →

factor
| *term* **mulop** *factor*
| *term* **&&** *factor*
| *term* **&** *factor*

factor →
id | **num_int** | **num_real**
| (*expression*)
| **!** *factor*
| ^ *factor*
| *array_type*

sign →
addop

const_exp →
num_int | **num_real** | **id** | *array_type*
| *sign* **num_int** | *sign* **num_real**

7 Conclusion

Design and implementation of embedded real-time systems is a complex and currently a major area of research. This project focused on the implementation of one such design, the **TRA model,** proposed and developed by Dr. Azer Bestavros. The TRA model is implemented via the embedded real-time systems specification language **CLEOPATRA -** *C-based Language* for *Event* driven *Object-oriented Prototyping* of *Asynchronous Time Constrained Reactive Automata.* A compiler for the specification language CLEOPATRA was the focus of this work.

I should note that the user of the CLEOPATRA compiler should be cautious when using the array data type within the state section of a TRA class definition. Currently, this issue has not been resolved, so the outcome of such array data type declarations cannot be predicted. Also it should be noted that there are many possible extensions to the TRA model specifications, such as creating arrays of included TRA classes within a TRA class definition - a sort of vector of TRAs - or a TRA class definition that has an array of channels as one of its types, and so on. The work presented here provides for the fundamental components of the TRA model semantics, and future extensions are left for those interested. Any bugs, comments, and/or critiques can be sent to email: *rpopp@brc.uconn.edu.*

8 References

Azer Bestavros. *Time-constrained Reactive Automata: A novel development methodology for embedded real-time systems.* PhD thesis, Harvard University, Division of Applied Sciences (Department of Computer Science), Cambridge, Massachusetts, September 1991.