The Verus Language:
Representing Time efficiently with BDDs[1]

Sérgio Campos and Edmund Clarke
Carnegie Mellon University
campos@cs.cmu.edu and emc@cs.cmu.edu

Abstract. There have been significant advances on formal methods and tools to verify real-time systems recently. Nevertheless, these methods have not yet been accepted as a realistic alternative to the verification of industrial systems. One reasons for this is that formal methods are still difficult to apply efficiently. Another reason is that even though being more efficient than previous ones, verification algorithms are still not efficient enough to handle many complex systems. This work addresses this problem by presenting a language designed especially to simplify writing real-time programs. It is an imperative language with a syntax similar to C. Special constructs are provided to allow the straightforward expression of timing properties. The familiar syntax makes it easier for non-experts to use the tool. The special constructs make it possible to model the timing characteristics of the system naturally and accurately. The language is compiled into state-transition graphs, which provide a simple but extremely efficient model of time. Symbolic model checking and quantitative algorithms are used to check the timing properties of the system. The efficiency of the representation allows complex realistic systems to be verified by the method as evidenced by the aircraft controller discussed in the paper. The model for this example has about 10^{15} states and counterexamples with thousands of states have been produced in seconds using the language and algorithms proposed.

1 Introduction

Formal verification tools are becoming more and more efficient every day. Until recently, it was not possible to verify large industrial real-time or general purpose systems using formal methods. The verification of complex systems was simply too expensive. Today this scenario has been changed by the development of more efficient verification methods such as symbolic model checking [1,10]. It is now possible to verify systems of realistic complexity that previously caused state explosion problems. For example, several real systems have been verified using symbolic model checking, such as the Futurebus cache coherence protocol [8] and the PCI Local Bus [6]. Many real-time systems have also been verified such as a medical monitoring system [5] and a robotics controller [7].

However, in spite of significant technical success, formal methods have yet to be recognized as a viable alternative to the verification of industrial systems. Two main reasons have prevented formal methods from being accepted by industry: the first relates to the efficiency of the verification algorithms. Even though

1. This research was sponsored in part by the National Science Foundation under grant no. CCR-8722633, by the Semiconductor Research Corporation under contract 92-DJ-294, and by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035.

current algorithms are significantly more efficient than their predecessors, there is still a limit on the size of problems that can be handled. Unfortunately several interesting examples are still out of reach. This problem is especially evident in real-time systems. Modeling time is difficult and frequently the time component is the bottleneck of the verification. The second reason is that most formal verification tools are difficult to use. Extensive knowledge about the verification method is frequently required. Also, the language used to verify the system is usually significantly different than the language used to implement it. As a consequence, the designer must maintain separate descriptions of the system, leading to problems in managing different versions of the code and potentially introducing translation errors. To complicate matters even further, the two goals of increasing the verification efficiency and the development of more expressive and simpler to use languages can be contradictory at times. A language with powerful constructs can be easy to program in, but at the same time the verification of those constructs can be expensive.

This work addresses these problems by describing a new language used to describe real-time systems called Verus. The Verus language has been introduced in [7], and this work details its semantics and how it affects the representation of time during verification. Verus provides a familiar environment for writing timed programs. Its syntax resembles the syntax of C, the language most frequently used to implement real-time systems. Designers can use Verus to verify models that are very close to the actual implementation. The development of the model and its verification can be performed faster since both languages are similar. Also, the translation process is less error-prone; to some extent it can even be automated.

Verus uses a discrete notion of time. The model of a Verus program is a finite state-transition graph and time passes by one time unit at each transition in the graph. The simplicity of this representation makes it amenable to a symbolic implementation using binary decision diagrams. This representation is very efficient, one real-time system verified has 15 concurrent processes and we have been able to produce counterexamples that have thousands of states in seconds. Perhaps more indicative of the usefulness of the method than the size of the counterexamples produced is the types of systems verified. We have applied this method to the verification of several real systems, such as an aircraft controller [4], a robotics controller [7] and a distributed heterogeneous real-time system [3]. In all cases the examples verified are either real existing systems or use components and protocols employed in current industrial products.

The Verus Language

The main goal of Verus is to allow engineers and designers to describe real-time systems easily and efficiently. It is an imperative language with a syntax resembling that of C. Special primitives are provided for the expression of timing aspects such as deadlines, priorities, and time delays. These primitives make timing assumptions explicit. A different approach is taken by many other languages, such as C, that allow programs where timing assumptions are not clearly stated. This results in ambiguous specifications that are difficult to prove correct. The approach taken in Verus makes the specification clearer and more complete.

The data types allowed are fixed-width integer and boolean. Nondeterminism is supported, which allows partial specifications to be described. Language constructs have been kept simple in order to make the compilation into a state-transition graph as efficient as possible. Simple constructs allow the precise expression of the desired features, since complex constructs sometimes force unnecessary details into the specification. Smaller representations can then be generated, which is critical to the efficiency of the verification and permits larger examples to be handled.

Other Real-Time Languages

There are several other languages for specifying finite-state real-time systems. Esterel [2] is one example of an imperative language used to describe real-time systems. Its syntax, however, may be very unfamiliar to many designers of real-time systems, used to program in C or similar languages. Moreover, Esterel constructs, unlike those of Verus, have not been designed to simplify modeling real-time systems. For example, specifying the execution of a periodic process with a deadline is not as straightforward as in Verus. Finally, Esterel is a deterministic language; it does not allow the expression of nondeterminism.

Modechart [10] is another example of real-time specification language. It is a graphical language in which nodes represent states, and transitions are explicitly drawn between states. This language, however, is more restrictive than Verus due to its graphical nature. Complex constructs such as periodic may be difficult to draw. Moreover, it is an explicit state enumeration language, since individual states are drawn in the program. Many systems are too large to be naturally described using graphical languages.

2 Overview of Verus

This section provides an overview of the language by presenting a simple real-time program. This program implements a solution for the producer-consumer problem by bounding the time delays of its processes. No synchronization is needed if the time delays of producer and consumer are defined properly.

The code for the `producer` process is shown below. Variable `p` is a pointer to the buffer in which data is stored. The `producer` initializes its pointer `p` to 0 and the `produce` variable to *false*. It then enters a nonterminating loop in which items are produced at a certain rate. Line 9 introduces a time delay of 3 units, after which an item will be produced. Line 10 marks the production of an item by asserting `produce`. In line 11 the pointer is updated appropriately. Line 12 makes sure that the event `produce` is observed. It is needed because the state of a Verus program can only be observed at `wait` statements. If a `wait` is not introduced in line 12, line 13 would cancel the effect of the assertion of `produce` before it can be observed. This behavior is discussed in detail later.

```
1   producer(p)
2   int p;
3   {
4     boolean produce;
```

```
5
6     p = 0;
7     produce = false;
8     while(!stop) {
9       wait(3);
10      produce = true;
11      p = p+1;
12      wait(1);
13      produce = false;
14    };
15  }
```

Figure 1. Producer code

Wait Statements

An important feature of Verus is illustrated in line 9. In Verus time passes only on `wait` statements. Lines 6, 7 and 8 execute in time zero and time elapses only after the loop condition has been tested. This feature allows a more accurate control of time, and eliminates the possibility of implicit delays influencing the results of the verification. It also generates models with fewer states, since contiguous statements are collapsed into one transition. Notice that this feature affects the behavior of the program significantly. For example, a block of code not containing the `wait` statement executes atomically.

Nondeterminism

To illustrate another characteristic of Verus, let's assume that the `producer` is not required to actually produce an item after 3 time units, but may instead leave the value of `p` unchanged. This can be modelled in Verus by changing line 11 to:

```
11      p = select{p, p+1};
```

The `select` statement introduces a nondeterministic choice in the program. The value of `p` after executing `select` can be either `p` or `p+1` (addition in Verus is defined modulo the maximum value for the variable). These choices can characterize the fact that the producer *may* produce an item, but it may also *not* produce it. This way we can model both possibilities without having to specify all the details that are actually needed to decide between these two options. Besides hiding unnecessary details, nondeterminism can be used to verify partial specifications. Whenever the value of a variable hasn't been determined by the design, nondeterministic constructs can specify all possible values the variable could take. This approximates the behavior of the actual system by exploring all possibilities. As the design process evolves, the values can be restricted until the correct behavior is determined. Nondeterminism encourages the use of automated verification in earlier phases of the design. Components of the system can be validated before all modules have been specified. In this way errors can be uncovered before propagating to components added later in the design.

```
16  consumer(p, c)
```

```
int p, c;
{
  boolean consume;

  c = 0;
  consume = false;
  while (!stop) {
    wait(1);
    if (p != c) {
      consume = true;
      c = c + 1;
      wait(1);
      consume = false;
    };
  };
}
```

Figure 2. Consumer code

The `consumer` process is very similar to the `producer`. The basic differences are that it waits for less time before consuming, and that it only consumes if `p` and `c` have different values (`p == c` signals an empty buffer). Notice that the `producer` does not check if the buffer is full before inserting another item. The time delays of both processes guarantee that an overflow will never occur.

The `main` function

As in the C language, `main` has a special function in Verus. In this function all processes are instantiated, and global variables can be declared. The variables `p` and `c` (used as pointers in the buffer) are declared and the `producer` and `consumer` processes are instantiated in the `main` function of the example code. An implicit instantiation of the `main` module is assumed, where `main` executes as another module.

Process instantiation in Verus follows a synchronous model. All processes execute in lock step, with one step in any process corresponding to one step in the other processes. Asynchronous behavior can be modeled by using *stuttering*, which introduces nondeterministic transitions and effectively models the desired behavior. This technique is described in detail in [].

Specifications can also follow the code as can be seen below. These specifications compute the minimum and maximum time between producing an item and consuming it, as well as checking that a `produce` is always followed by a `consume`. More information about the verification method can be found in the appendix and in [].

```
main()
{
  int p, c;

  process prod producer(p, c),
```

```
          cons consumer(p, c);

  spec MIN[prod.produce, cons.consume]
       MAX[prod.produce, cons.consume]
       AG(prod.produce -> AF cons.consume)
}
```

Figure 3. Producer/consumer main function

Periodic Execution.

To illustrate different features of Verus some extensions to the program above are considered. The first comes from realizing that both processes will always execute, even when no data exists. For example, the `consumer` executes even if the `producer` does not generate items. This can be avoided using *periodic execution*, where execution is scheduled at specific points in time. It can be easily specified in Verus. The `producer` can be made into a periodic process executing once every 10 time units as seen in figure 4.

The `periodic` statement has four parameters, the last being the code that will be executed periodically. The first parameter is the *start_time*, which specifies how many time units the periodic code will idle *before* starting its execution for the first time. In this example it will start immediately. The second parameter is the *period*. In this case the statements following `periodic` will execute once every 10 time units. The third parameter defines a *deadline*. It states that the execution must finish in less than 10 time units or an exception will be raised (exception handling is discussed later). Execution may take longer than the sum of the waits because of synchronization with other processes.

```
producer(p, c)
int p, c;
{
  boolean produce;

  p = 0;
  produce = false;
  periodic(0, 10, 10) {
    wait(3);
    produce = true;
    p = p+1;
    wait(1);
    produce = false;
  };
}
```

Figure 4. Periodic producer

Deadlines

Verus also allows the definition of deadlines independent of periodicity. For example, we can specify that `producer` will be an aperiodic process, but that it must finish each iteration in less than 10 time units by substituting the `periodic` statement above by `deadline(10)`.

Exceptions

Exception handling is used to control abnormal situations. The only exception currently defined in Verus is a *missed deadline*. It occurs when the code inside a `deadline` or a `periodic` statement does not finish within the specified time. An exception handler must be specified for the exception to take effect. If no exception handlers are defined, the exception is ignored. Whenever a deadline is missed the code designated as handler is executed. After the execution of the exception handler the rest of the code inside the deadline scope is ignored. Control is then passed to the statement following the deadline statement. This could be the next instantiation of a periodic process when the exception occurs inside a `periodic` statement or the code after a `deadline` statement.

```
producer(p, c)
int p, c;
{
  boolean produce;

  handler {
    error = 1;
  } for {
    p = 0;
    produce = false;
    periodic(0, 10, 10) {
      wait(3);
      produce = true;
      p = p+1;
      wait(1);
      produce = false;
    };
  };
}
```

Figure 5. Exception handling

Figure 5 shows the typical exception handling mechanism. Whenever a deadline is missed an error flag is asserted. The verification procedure can then check to see if the error condition is reachable. In some other applications, however, a different behavior may be the desired one. For example, a multimedia program might choose to ignore some image frame that hasn't arrived, provided the last *n* arrived. An exception handler to model this behavior can be easily written: whenever an exception occurs, the handler would record the number of the frame missed. If the current missed frame is at least *n* frames apart from the previous one, no error would be issued. Otherwise an error condition would be asserted.

3 Semantics

The meaning of a Verus program is a state-transition graph. Section 3.1 explains how state-transition graphs are represented in Verus. In section the concept of *wait graphs* is introduced. Wait graphs are an abstraction used to keep track of the control flow of the program. Verus consists of two parts, the basic constructs form the core language, and the more complex ones form the extension language. The formal semantics of the core language is described in section 3.2, while section 4.4 presents the semantics of the extensions to the core language. Finally, section 4.5 discusses the semantics of concurrent processes in Verus.

3.1 State-Transition Graphs in Verus

The state-transition graph constructed from a program P is $G_P = (S_P, I_P, T_P)$, where S_P is the set of states, I_P is the set of initial states and T_P is the transition relation. The set of states is defined by the variables in the program. Each possible assignment to the variables is a state. I_P and T_P are defined by the program as will be seen shortly.

Symbolic Representation

States are defined by the assignment of values to program variables. Each possible assignment to the program variables is a state. For example, if the program has three boolean variables a, b and c, examples of states are (a,b,c), $(a,\overline{b},\overline{c})$ and $(\overline{a},\overline{b},c)$, where, for variable v, v means the variable is true in the state, and $\overline{v}$ means the variable is false. Boolean formulas over program variables can be true or not in a given state. The value of a boolean formula in a state is obtained by substituting into the formula the values of the variables in that state. For example, the formula $(a \vee c)$ is true in all states shown above. The graph representation used by Verus is a direct consequence of this observation. Sets of states are represented by boolean formulas, where each formula represents the set of states in which the formula is true. For example, the formula *true* represents the set of all states, the formula *false* represents the empty set of states, and the formula $(a \vee c)$ represents the set of states in which a or c are true.

Transitions can also be represented by boolean formulas. A transition $T(s, s')$ is represented using two sets of variables, one for the current state and another for the next state. Each variable in the next state set corresponds to one variable in the current state set. If state s is represented by the formula f_s over the current state variables, and state s' is represented by formula $f_{s'}$ over the next state variables, then the transition $T(s, s')$ is represented by the formula $f_s \wedge f_{s'}$. For example, a transition from state $(\overline{a},\overline{b},\overline{c})$ to state $(\overline{a},b,\overline{c})$ is represented by the formula $\neg a \wedge \neg b \wedge \neg c \wedge \neg a' \wedge b' \wedge \neg c'$. The transition relation of a graph is the disjunction of all transitions in the graph. The meaning of the formula representing the transition relation is the following: there exists a transition from s to s' iff the substitution of the variable values for s in the current state variables and s' in the next state variables of the transition relation yields *true*.

The symbolic representation relies on the fact that states are represented by the values of the atomic propositions in those states. In order to guarantee that states can be identified uniquely, we must make the assumption that different states have different labeling of propositions. More formally, we assume that for any two states s_1 and s_2 in S, if $L(s_1) = L(s_2)$ then $s_1 = s_2$. This assumption does not, however, impose any restrictions on the model, since extra atomic propositions can be added in order to make $L(s_1) \neq L(s_2)$ for distinct states s_1 and s_2 [62].

Tracking the Control Flow — Wait Graphs

The execution of a Verus statement may change the value of one or more program variables. In general, it changes a given state s into another state s'. Executing a sequence of statements in a given state then generates a sequence of states. However, in Verus not all of those states are observable. The state of the program can only be observed at `wait` statements. When a `wait` is executed all changes caused by the execution of the block of statements since the previous `wait` take effect at the same time. Transitions in the graph occur only when `wait` statements are executed.

```
...
wait(1);
a = false;
c = true;
d = d + 1;
wait(1);
...
```

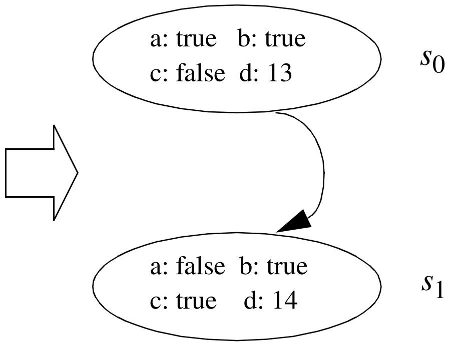


Figure 6. Wait statement example: if s_0 is the current state at the first `wait`, s_1 will be the current state at the second.

Each transition in the graph corresponds to time elapsing by one unit. The statement `wait(1)` corresponds to one transition in the graph. Longer waits are modeled by a sequence of unit transitions. This allows the programmer to specify exactly when time passes, and permits a more accurate model of time than possible if each statement takes one time unit to execute.

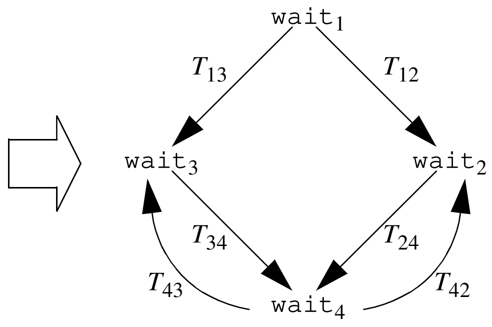
It is easier to understand the behavior of a Verus program by concentrating on its `wait` statements. This can be accomplished by translating the program into a *wait graph*. The wait graph corresponding to a Verus program is a graph in which the states are the `wait` statements in the program. It corresponds to an intermediate representation between the Verus program and the corresponding state-transition graph. It is used only to illustrate how this translation occurs and is not actually constructed. In the discussion below, to differentiate between distinct waits, $\mathtt{wait}_i$ represents the i^{th} occurrence of a `wait` statement in the program.

Subscripts have been added to the sample program below to aid the presentation, but no subscript exists in actual programs.

As discussed, each `wait` in the program is a state in the wait graph. Transitions between `wait`s are defined as follows. A transition between $\texttt{wait}_i$ and $\texttt{wait}_j$ exists iff $\texttt{wait}_j$ can be reached from $\texttt{wait}_i$ in the control flow of the program without going through intermediate `wait`s. Edges of the wait graph are labelled by a relation T_{ij} between any two states in the state-transition graph. This relation describes state changes caused by the execution of the statements between the corresponding `wait`s. The construction of this relation is described in the next section. Intuitively, given two states s and s', $T_{ij}(s, s')$ means that if program execution is in $\texttt{wait}_i$ and the current state is s, then there exists a path in the control flow leading to $\texttt{wait}_j$ without intermediate waits, and executing the statements on this path changes state s into state s'.

Notice that T_{ij} represents exactly all transitions from s to s' in the state graph such that s and s' are respectively the current state of the program before and after control is transferred from $\texttt{wait}_i$ to $\texttt{wait}_j$. This makes it possible to construct the state transition graph that corresponds to a given Verus program from its wait graph. The set of all relations between `wait` statements represents all transitions in the program. Consequently, the disjunction of all such transitions constitutes the transition relation of the state-transition graph of the program.

```
wait1(1);
S1;
while cond1 {
  if cond2 {
    S2;
    wait2(1);
    S3;
  } else {
    S4;
    wait3(1);
    S5;
  };
  S6;
  wait4(1);
};
```



($\texttt{S}_i$s correspond to blocks of assignments)

Figure 7. A Verus program and its corresponding wait graph

Wait Counters

Since each relation T_{ij} corresponds to a set of transitions, their disjunction should correspond to the transition relation of the program. However, this is not true because T_{ij} does not contain information about where it came from ($\texttt{wait}_i$) and where it leads to ($\texttt{wait}_j$). The disjunction of all relations would not maintain consistency of the values of variables after the execution of a sequence of `wait`s.

This problem is solved by creating an extra variable in the program to record this information, the wait counter `wc`. Each `wait` statement is preceded by an assignment `wc` = i, where i is the occurrence number of the `wait` statement (this assignment is introduced by the compiler; it is not part of the source code). The relation T_{ij} now contains information about where it leads to, since the assignment `wc` = j is introduced before $\texttt{wait}_j$. As detailed in the next section, the previous value of the wait counter indicates where this transition came from. Now T_{ij} has all information needed to maintain consistency across sequences of `wait` statements. The disjunction of all relations between `waits` is the transition relation of the program.

Determining the Initial State Set

The initial state set of a Verus program is the state it reaches just after executing the first `wait`. In order to compute the initial state set, Verus programs start with an initial `wait`, with the wait counter of 0 (introduced by the compiler). The state of the program at this point, S_0, is represented by the formula (wc = 0). The initial state set is defined as the set of states reached from S_0 in one transition. Alternatively, the initial state could be defined as (wc = 0). However, this can cause a non intuitive behavior because in the set of states defined by (wc = 0) no variable has been initialized. By defining the initial state set as the set of states reachable from (wc = 0) in one step we make sure that all variables have been initialized in the initial state of the program.

3.2 Core Language Semantics

The state space of a Verus program is defined by a set of boolean variables. A state in the model is an assignment of values to the variables. The set of all states is *ST*. A relation between any two states belongs to *Relation* ≡ *Powerset*(*ST* x *ST*).

The semantics is defined as a function of the program text. The meaning of a program is a transition relation between states such that there is a transition from state s to state s' iff there exists an execution sequence leading from $\texttt{wait}_i$ to $\texttt{wait}_j$ without intermediate `wait` statements that changes state s into s'.

The function R given below constructs the relations between `wait` statements. Intuitively, given a relation r describing the program until the execution of statement *Stmt*, the function R will produce the relation r' describing the program after executing *Stmt*. The function R also constructs another relation t by accumulating the relations constructed for all `wait` statements. Function R is defined by

$$R\text{: STMT} \rightarrow \textit{Relation} \text{ x } \textit{Relation} \rightarrow \textit{Relation} \text{ x } \textit{Relation}$$

where pairs of relations are $\langle r, t\rangle$, r being the relation containing changes to the program state since the last `wait` statement, and t being the transition relation of the program, that is, the disjunction of the relations between all pairs of `wait` statements.

The state-transition graph corresponding to a program P is constructed as follows. Given program P, function R constructs $\langle r, t\rangle = R[\![P]\!]\langle wc = 0, \varnothing\rangle$, where t is the transition relation of the state-transition graph corresponding to P, and the initial state set is constructed from t as discussed above.

Additional definitions are needed before presenting the semantic functions:

- There are only boolean variables in the program. Integer variables are encoded in binary and substituted for the corresponding boolean variables.
- V and V' are two sets of boolean variables such that for each variable `v` in the program there are corresponding variables $v \in V$ and $v' \in V'$. The variable $v \in V$ represents the value of the program variable `v` in the current state, and the variable $v' \in V'$ represents its value in the next state. A transition is a relation between variables in V and V'.
- A variable wc is introduced in the model. It is used to keep information about the previous and next `wait` statements in the control flow. An assignment `wc = i;` is assumed to exist just before statement $\texttt{wait}_i$.
- An initial `wait` (with wait counter value of 0) is the second statement of the program, preceded by an assignment `wc = 0`.
- All programs are assumed to have as the last statement:

  ```
  while (true) wait(1);
  ```

  There are two reasons for this requirement. Transitions are only generated at `wait` statements (see $R[\![$ `wait` $]\!]$) and the existence of a final `wait` guarantees that transitions will be generated for all programs. Moreover, this loop also guarantees that even when the program terminates and no more statements are executed, the state of the program can still be observed. Intuitively the loop means that after the program terminates its state will remain unchanged.

Expressions

The meaning of a Verus expression is a boolean formula corresponding to the syntactic expression. Since the core language only allows boolean expressions, the translation is straightforward; it is described below by the function E:

Primary Expressions

$E[\![\ \texttt{true}\ ]\!] = true$
$E[\![\ \texttt{false}\ ]\!] = false$

$E[\![\ \texttt{v}\ ]\!] = v'$, where `v` is an internal variable and $v' \in V'$.
$E[\![\ \texttt{v}\ ]\!] = v$, where `v` is an external variable and $v \in V$.

Internal variables are represented by their *next* state value, while external variables are represented by their *current* state value. This choice of representation significantly affects the behavior of each type of variable, as described below.

Initially, lets consider how internal variables behave. All references to an internal variable will be denoted by its next state variable. For example, a reference to variable `v` in the left-hand of an assignment (as in `v = false`) will be denoted by the next state variable v', and therefore the assignment will change the value of v' in the current relation (see semantics of assignments). This is the expected behavior, since an assignment determines the value of the variable in the next state.

However, other references to `v` (as in `x = !v`) also refer to v'. In the assignment `x = !v` the value of x' in the current relation will be assigned the negation of the value of v'. Two cases must be considered. If variable `v` has been assigned a value previously, this assignment has updated the value of v' in the current relation. Consequently, the assignment to `x` uses the most recent value assigned to `v`. To illustrate how this affects the behavior of the program, consider the program fragment below. The value assigned to `x` in line 4 is *false*, because in the current relation of the program at line 4 the value of variable v' is *true*.

```
1  v = false;
2  wait(1);
3  v = true;
4  x = !v;
```

Figure 8. Example of the behavior of internal variables

In the case that variable `v` has not been assigned any value, the current relation enforces that the value of internal variables do not change via the clause ($\wedge_{v \in \text{ internal variables}}\ v = v'$) introduced in the current relation at `wait` statements (see $R[\![$`wait`$]\!]$). This clause guarantees that the current and next state variables of internal variables have the same value (the clause is automatically overridden if an assignment is made). This has the effect that the value of an internal variable does not change if no assignments are made. This is true even across `wait`s. For example, if line 3 did not exist in the fragment above, the value of v' would be *false* in the current relation because of this clause, and therefore the value assigned to `x` would be *true*.

External variables, on the other hand, are not included in the `wait` statement clause introduced in the current relation. This is because their value is not maintained across `wait` statements. External variables may change value nondeterministically at `wait` statements and they cannot be assigned to. The value an external variable has at any point in the program is the value it had in the previous `wait` statement, since no assignments exist. This value is represented by its current state variable.

Boolean Expressions

$E[\![\ expr_1\ \texttt{||}\ expr_2\ ]\!] = expr_1 \vee expr_2$

$E[\![\ expr_1\ \texttt{\&\&}\ expr_2\ ]\!] = expr_1 \wedge expr_2$

$E[\![\ \texttt{!}\ expr\ ]\!] = \neg\ expr$

3.3 Statements

Assignments

$R[\![\ \texttt{v = } expr\ ]\!]\langle r, t\rangle = \langle(\exists\ y\ [\ v = Expr\ ^y/_v \wedge r\ ^y/_v]), t\rangle,$

where $v = E[\![\ \texttt{v}\ ]\!]$, $Expr = E[\![\ expr\ ]\!]$ and y is a new variable.

This expression computes the strongest post-condition for the assignment `v = ` *expr* given r as a pre-condition. If r is the set of valid transitions in the graph since the last `wait` statement, the expression above determines the largest set of transitions that satisfy the assignment and that satisfy r for variables other than v. Intuitively, this expression substitutes the previous value of v in r for *Expr*, while maintaining the values of other variables.

$$R[\![\ \texttt{v = select\{}expr_1\texttt{, }expr_2\texttt{\}}\ ]\!]\langle r, t\rangle = \textbf{let}\langle r', t\rangle = R[\![\ \texttt{v = } expr_1\ ]\!]\langle r, t\rangle,$$

$$\langle r'', t\rangle = R[\![\ \texttt{v = } expr_2\ ]\!]\langle r, t\rangle\ \textbf{in}$$

$$\langle r' \vee r'', t\rangle$$

The relation for a nondeterministic assignment is the disjunction of the expression for each possible assignment. In other words, a nondeterministic assignment is true if any possible value is assigned. The extension of R for the case in which more than two expressions exist is a simple extension of the disjunction shown, and is omitted for brevity.

Sequential Execution

$R[\![\ S_1\ ;\ S_2\ ]\!]\langle r, t\rangle = R[\![\ S_2\ ]\!](R[\![\ S_1\ ]\!]\langle r, t\rangle)$

Wait Statements

$R[\![\ \texttt{wait}_i\texttt{(1)}\ ]\!]\langle r, t\rangle = \langle((wc = i) \wedge \wedge_{v \in IV}\ v = v'), (t \vee r)\rangle,$

where IV is the set of internal variables in the program.

Function R for the `wait` statement changes the previous relation in two ways. At this point in the program transitions that lead to $\texttt{wait}_i$ are generated. These transitions are represented by relation r before the `wait` is executed, which is then disjointed with the previous transition relation t. This is the only statement that changes the value of t.

Moreover, the current relation after the execution of $\texttt{wait}_i$ must reflect the fact that a new set of transitions will be computed. The new relation specifies that transitions start in $\texttt{wait}_i$ with the formula $(wc = i)$, that is, the wait counter value of the current state variable in the transition will be i. The destination of the new set of transitions will be established when the next `wait` statement is found. At that point the assignment `wc` = j before $\texttt{wait}_j$ introduces the formula $(wc' = j)$ in the current relation, specifying where the transition leads to. Because of these two conditions, all transitions specify a value for both the current and next state wait counters.

Finally, it is necessary to introduce the expression $\wedge_{v \in IV}\, v = v'$ into the current relation. For internal variables this expression guarantees that unless assigned a new value, internal variables maintain their previous value across transitions. External variables may change value during transitions, and therefore are not included in this expression. This allows the use of the next state variable as the semantic value of internal program variables. Whenever an internal variable is referenced, its next state variable will have its previous value (via the clause $v = v'$ above) or its new value (via the assignment expression described previously) in the current relation.

Conditionals

$$
\begin{aligned}
R[\![\, \texttt{if}\ cond\ S_1\ \texttt{else}\ S_2\,]\!]\langle r, t\rangle = \mathbf{let}\, & \langle r', t'\rangle = R[\![\, S_1\,]\!]\langle (r \wedge cond), t\rangle, \\
& \langle r'', t''\rangle = R[\![\, S_2\,]\!]\langle (r \wedge \neg cond), t\rangle\ \mathbf{in} \\
& \langle r' \vee r'', t' \vee t''\rangle
\end{aligned}
$$

Each branch in the `if` statement is executed by restricting its parameter to the set of transitions that satisfy the appropriate conditional — S_1 receives those transitions satisfying *cond*, and S_2 receives transitions not satisfying *cond*. In this way, if control reaches the `if` statement through a state that satisfies the condition, control will proceed to S_1. If the state does not satisfy the condition, control proceeds to S_2. The representation of a conditional is the disjunction of the representation of its branches.

Loops

$$
\begin{aligned}
R[\![\, \texttt{while}\ cond\ S_1\,]\!]\langle r, t\rangle = \mathbf{let}\, & \langle r', t'\rangle = R[\![\, \texttt{while}\ cond\ S_1\,]\!](R[\![\, S_1\,]\!]\langle (r \wedge cond), t\rangle), \\
& \langle r'', t''\rangle = \langle (r \wedge \neg cond), t\rangle\ \mathbf{in} \\
& \langle r' \vee r'', t' \vee t''\rangle
\end{aligned}
$$

The representation of a `while` loop can be seen as unrolling the loop into nested `if` statements: `if` *cond* {S_1; `if` *cond* {S_1; ...}};. However, even though simple to understand, function R for a `while` statement cannot be computed as shown above, because it is circular. A more accurate definition can be seen below. It uses the *fix* operator, which returns the least fixpoint of the functional given as its argument.

$$
\begin{aligned}
R[\![\, \texttt{while}\ cond\ S_1\,]\!] = fix(\lambda f\, \lambda\langle r,t\rangle.\mathbf{let}\, & \langle r', t'\rangle = f(R[\![\, S_1\,]\!]\langle (r \wedge cond), t\rangle), \\
& \langle r'', t''\rangle = \langle (r \wedge \neg cond), t\rangle\ \mathbf{in} \\
& \langle r' \vee r'', t' \vee t''\rangle)
\end{aligned}
$$

The operations performed by the functional above are projection (from the result of the application of f into r' and t'), disjunction (of r', r'' and t', t'') and pairing (of the results of the disjunctions). Since these operations are continuous [76], any functional constructed from them is also continuous. By being continuous, the functional is also monotonic, and therefore it has a fixpoint.

However, not all programs with while statements have well behaved semantics. For example, a fixpoint characterization for the program below is the relation *false*, which corresponds to non-termination, as is expected from the program. But since there are no `waits` in the program, time does not pass. Non-termination in this case means that if the program is in state *s* when the code below is executed, *there will be no outgoing transition from s*, that is, the non-terminating behavior is not observable. In order to avoid this anomalous behavior, we impose the rule that all execution sequences inside all `whiles` in the program must execute at least one `wait` statement. This ensures that even non-terminating `while` programs are always observable and that no states without outgoing transitions will be created.

```
while(true) {
  a = !a;
}
```

Figure 9. A while program with a trivial fixpoint

Null Statement

$R[\![\, \texttt{null} \,]\!]\langle r, t\rangle = \langle r, t\rangle$

Schedule Statements

schedule_statement ::=
  `deadline (` *constant* `)` *compound_statement*

The deadline statement is translated into the Verus core language by creating an integer variable `timer`. At the `deadline` keyword an assignment `timer = 0` is inserted. Within the scope of the deadline, each `wait(`*n*`)` statement is preceded by `timer = timer +` *n*`;` and by a check `if (timer >= deadline)` *error_code*, where the exception handler defines *error_code*.

schedule_statement ::=
  `periodic (` *constant*`,` *constant*`,` *constant* `)` *compound_statement*

The periodic statement is handled in a similar way. The difference is that an infinite loop is inserted enclosing the periodic statement, and once the periodic statement has finished executing, a loop is inserted to enforce the periodicity:

```
while (timer < period) {
  timer = timer + 1;
  wait(1);
};
```

A similar loop is inserted before the main loop at the beginning of the periodic statement to account for the initial offset.

Exception Handling

schedule_statement ::=

`handler` *compound_statement* `for` *compound_statement*

The first compound statement is the exception handler, and the second is the scope of the handler. The exception handling statement `handler` S_1 `for` S_2 is translated by substituting the *error_code* created by deadline statements in S_2 for: S_1 `else {`. The compound statement S_1 is executed in case of a missed deadline, and the `else` clause guarantees that the rest of the `deadline` statement is skipped in case of a missed deadline. The `{` after the `else` is closed at the end of the deadline statement.

4 Conclusions

5 References

[1] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Symposium on Logics in Computer Science*, 1990.

[2] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. In: *Science of Computer Programming*, vol. 19, 1992.

[3] S. V. Campos and O. Grumberg. Selective quantitative analysis and interval model checking: verifying different facets of a system. In: *Computer Aided Verification,* 1996.

[4] S. V. Campos, E. M. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *IEEE Real-Time Systems Symposium*, 1994.

[5] S. V. Campos, E. M. Clarke, W. Marrero and M. Minea. Timing analysis of industrial real-time systems. In: *Workshop on Industrial-strength Formal specification Techniques*, 1995.

[6] S. V. Campos, E. M. Clarke, W. Marrero and M. Minea. Verifying the performance of the PCI local bus using symbolic techniques. In: *ICCD*, 1995.

[7] S. V. Campos, E. M. Clarke, W. Marrero and M. Minea. Verus: a tool for quantitative analysis of finite-state real-time systems. In: *Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1995.

[8] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus$^+$ cache coherence protocol. In *Proceedings of the 11th CHDL*, 1993.

[9] P. Clements, C. Heitmeyer, G. Labaw, and A. Rose. MT: a toolset for specifying and analyzing real-time systems. In *IEEE Real-Time Systems Symposium*, 1993.

[10] F. Jahanian and D. Stuart. A method for verifying properties of modechart specifications. In: *IEEE Real-Time Systems Symposium*, 1988.

[11] K. L. McMillan. *Symbolic model checking - an approach to the state explosion problem*. Ph.D. thesis, SCS, Carnegie Mellon University, 1992.

6 Appendix

6.1 The Verification Algorithms

In this work we use a new method for specifying and verifying real-time systems. The system being verified is compiled into a state-transition graph. Algorithms derived from symbolic model checking are used to compute *quantitative* information about the model. The information produced allows the user to check the temporal correctness of the model: schedulability of the tasks of the system can be determined by computing their response time; reaction time to events and several other parameters of the system can also be analyzed by this method. This information provides insight into the behavior of the system and in many cases it can help identify inefficiencies and suggest optimizations to the design. The same algorithms can then be used to analyze the performance of the modified design. The evaluation of how the optimizations affect the design can be done *before* the actual implementation. This can significantly reduce development costs.

Most verification algorithms assume that timing constraints are given explicitly in some notation like temporal logic. Typically, the designer provides a constraint on response time for some operation, and the verifier automatically determines if it is satisfied or not. Unfortunately, these techniques do not provide any information about how much a system deviates from its expected performance, although this information can be extremely useful in fine-tuning the behavior of the system.

In this work we use algorithms that determine the minimum and maximum length of all paths leading from a set of starting states to a set of final states. We also use algorithms that calculate the minimum and the maximum number of times a specified condition can hold on a path from a set of starting states to a set of final states. Our algorithms provide insight into *how well* a system works, rather than just determining whether it works at all. They enable a designer to determine the timing characteristics of a complex system given the timing parameters of its components. This information is especially useful in the early phases of system design, when it can be used to establish how changes in a parameter affect the global behavior of the system.

Several types of information can be produced by this method. Response time to events is computed by making the set of starting states correspond to the event, and the set of final states correspond to the response. Schedulability analysis can be done by computing the response time of each process in the system, and comparing it to the process deadline. Performance can be determined in a similar way. The algorithms have been used to verify several real-time and non real-time systems.

6.2 A Larger Example

This section presents a more realistic application of the Verus tool than the producer/consumer program described above. We will briefly describe an aircraft controller system that is based on controllers employed in existing military aircraft []. Some examples of the Verus code used to model the system are also shown. We conclude with a brief analysis of the results obtained by our verification algorithms. A full analysis of this example (except for the Verus code) can be found in [].

The control system for an airplane can be characterized by a set of sensors and actuators connected to a central processor. This processor executes the software to analyze sensor data and control the actuators. Our model describes this control program and defines its requirements so that the specifications for the airplane are met. The model is similar to the one described in [60].

The aircraft controller is divided into systems and subsystems. Each system performs a specific task in controlling a component of the airplane. The most important systems are implemented in our model to provide a realistic representation of the controller. The systems being controlled are:

- Navigation: Computes aircraft position. Takes into account data such as speed, altitude, and positioning data received from satellites or ground stations.
- Radar Control: Receives and processes data from radars. It also identifies targets and target position.
- Radar Warning Receiver: This system identifies possible threats to the aircraft.
- Weapon Control: Aims and activates aircraft weapons.
- Display: Updates information on the pilot's screen.
- Tracking: Updates target position. Data from this system are used to aim the weapons.
- Data Bus: Provides communication between processor and external devices.

Each system is composed of one or more subsystems. Timing constraints for each subsystem are derived from factors such as required accuracy, human response characteristics and hardware requirements. There are 15 subsystems in our example. Concurrent processes are used to implement each subsystem. Processes execute periodically and are defined by their period and the execution time of each instantiation. Periods range from 25 to 200 ms, and execution times range from 1 to 9 ms. Communication among the various processes is done indirectly. No data are shared directly by two subsystems. Processes communicate only through data servers called *monitor tasks*. The time to access data from the monitor tasks is included in the execution time of the process.

The code shown below is the model for a process with period of 200ms and execution time of 3ms.

```
p1()
{
```

```
  boolean start, end;

  start = false;
  end = false;
```

Initially we declare two boolean variables that flag the beginning and end of execution of each instantiation of the process. They are used when checking how long the process can execute for. The initial value of both variables is false.

```
  periodic(0, 200, 0) {
    start = true;
    wait(1);
```

We then start the periodic execution with a period of 200 ms and flag the beginning of the execution. A one time unit wait is inserted so other processes can observe that `start` is true.

```
    start = false;
    wait(2);
```

The variable `start` is then deasserted, and the process executes for the remaining of its execution time.

```
    end = true;
    wait(1);
    end = false;
  };
}
```

Finally the end of execution is asserted and the periodic loop is iterated. The code for the other 14 processes is similar. The code for the scheduler completes the model. But before presenting the scheduler, it is important to understand the interaction between processes and scheduler. Processes request execution by asserting a `request` variable which is read by the scheduler. Upon deciding which processes executes the scheduler asserts a variable `granted` read by all processes. To make this interaction clear we repeat the code for the same process, but now translated into the core language to show the features discussed. Another modification has been implemented below, moving the periodic statement into the scheduler. In this way more than one process can make use of the same variable used to implement the periodicity.

```
int time, granted;
boolean req1, req2,... ;
```

Variable `time` is the counter used to enforce periodicity. The variables `granted` and `req`*i* are used by the scheduler.

```
p1(time, granted, req1)
{
  boolean start, end;

  req1 = false;
  start = false;
```

```
9     end = false;
10    while (true) {
11      while (time != 0) wait(1);
```

The periodic statement has been replaced by an infinite loop that only starts when `time` is `0`. Notice that when waiting on line 11 variable `req1` is false, and therefore `p1` does not request execution at this point.

```
12      req1 = true;
13      start = true;
14      while (granted != 1) {wait(1); start = false;};
15      wait(1);
16      start = false;
```

When execution starts the request variable is asserted and the process must wait until being granted the processor (line 14) before continuing. Line 15 corresponds to the process executing for one time unit.

```
17      while (granted != 1) wait(1);
18      wait(1);
19      while (granted != 1) wait(1);
20      wait(1);
```

The process executes until completion in lines 17 to 20. But before each step it must check to see if it still has the processor.

```
21      end = true;
22      req1 = false;
23      wait(1);
24      end = false;
25    };
26  }
```

The end of the execution is similar to the previous one.

```
27  scheduler(req1, req2,..., time, granted)
28  {
29    time = 0;
30    while (true) {
31      if (req1)  granted = 1;  else
32      if (req2)  granted = 2;  else
33      ...
34      granted = 0;
```

The scheduler executes an infinite loop which starts by assigning a value to the `granted` variable. It checks requests in the priority order (highest priority first) and grants the processor to the higher priority requesting process. In this example priorities are static, they are defined by the order in which requests are tested.

```
35      wait(1);
36      if (time < 199) {
37        time = time + 1;
```

```
38        } else {
39          time = 0;
40        };
41      };
42    }
```

Line 35 makes the `granted` variable observable. The scheduler then increments the `time` variable and repeats the cycle.

```
42    main()
43    {
44      process P1 p1(time, granted, req1),
45              P2 p2(time, granted, req2),
46              ...
47              SCH scheduler(req1, req2,..., time, granted);
48
49      spec
50        MIN(P1.start, P1.end)
51        MAX(P1.start, P1.end)
52        MIN(P2.start, P2.end)
53        MAX(P2.start, P2.end)
54        ...
55    }
```

In the main module all processes are instantiated, and their schedulability is checked using quantitative timing analysis. A real-time system is schedulable if all processes finish execution before their deadline. Usually the deadline is the same as the period, that is, processes must finish before their next instantiation. We have been able to determine that the system is schedulable. We have also been able to determine several other properties of the system such as the response time of the weapons subsystem. Whenever the pilot presses the firing button a complex sequence of events occurs. We have been able to determine its fastest and its slowest response times. The complete analysis of this example can be found in []. The final model has about 10^{15} states, and the transition relation uses approximately 4600 BDD nodes. Properties have been computed in seconds in all cases.