ELSEVIER

Theoretical Computer Science 253 (2001) 95–118

Theoretical Computer Science

www.elsevier.com/locate/tcs

with BDDs

Sérgio Vale Aguiar Campos[a,*,1], Edmund Clarke[b,2]

[a] *Dept. de Ciencia de Computação, Universidade Federal de Minas Gerais, C.P.702, 30161-970 Belo Horizonte, Brazil*

[b] *Carnegie Mellon University, Pittsburgh, USA*

Abstract

There have been significant advances on formal methods to verify complex systems recently. Nevertheless, these methods have not yet been accepted as a realistic alternative to the verification of industrial systems. One reason for this is that formal methods are still difficult to apply efficiently. Another reason is that current verification algorithms are still not efficient enough to handle many complex systems. This work addresses the problem by presenting a language designed especially to simplify writing time-critical 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. A symbolic representation using BDDs, model checking and quantitative algorithms are used to check system timing properties. The efficiency of the representation allows complex realistic systems to be verified. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Symbolic model checking; Timed systems; BDDs; Verus

1. Introduction

Formal verification tools are becoming more and more efficient every day. Until recently, it was not possible to verify large industrial systems using formal methods.

[*] Corresponding author.

E-mail addresses: scampos@dcc.ufmg.br (S.V.A Campos), emc@cs.cmu.edu (E. Clarke).

[1] This research was sponsored in part by Conselho Nacional de Pesquisa e Desenvolvimento under the project "Métodos Formais para Verificaçáo de Sistemas Computacionais de Complexidade Industrial".

[2] This research was sponsored in part by the National Science Foundation under grant CCR-9217549, by the Semiconductor Research Corporation under contract 95-DJ-294 and by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under title "Research on Paral lel Computing", ARPA Order 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035.

PII: S0304-3975(00)00090-6

Today this scenario has been changed by the development of more efficient verification methods such as symbolic model checking [5, 22]. It is now possible to verify systems of realistic complexity such as the Futurebus cache coherence protocol [11] and the PCI Local Bus [8].

However, in spite of significant technical success, formal methods have yet to be recognized as a viable alternative to the verification of industrial systems. One reason is because even though 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 systems where *time* is a vital parameter such as controllers for industrial machinery, power plants or airplanes. In these systems, a late response can have serious or even fatal consequences. We will refer to this type of systems as *time-critical* systems.[3] Modeling time is difficult and frequently the time component is the bottleneck of the verification. Another difficulty in the use of formal verification is that most tools are not simple to use. Extensive knowledge about the verification method is frequently required. Also, the language used to verify the system is usually significantly different from 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. Moreover, the two goals of increasing the verification efficiency and the development of more expressive and simpler to use languages can be contradictory. A language with powerful constructs can be easy to program in, but the verification of those constructs can be expensive.

This work addresses these problems by presenting a new language used to describe time-critical systems called Verus. Verus provides a familiar environment for writing timed programs. Its syntax resembles the syntax of C, the language most frequently used to implement such systems. 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. This work describes Verus in detail and shows how time-critical programs can be efficiently represented and manipulated symbolically.

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, we have applied this method to the verification of several real systems, such as an aircraft controller [7], a robotics controller [9] and a distributed heterogeneous time critical system [10]. In all cases the examples verified are either actual systems or use components and protocols employed in current industrial products.

[3] They are also called real-time systems in the literature, but this term can be confusing since real-time also connotes the use of a continuous time representation.

1.1. The Verus language

The main goal of Verus is to allow engineers and designers to describe timed systems easily and efficiently. 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.

2. Overview of Verus

This section provides an overview of the language by presenting a simple time-critical 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 and the `produce` variable signals the production of an item. After initialization, the program enters a nonterminating loop in which items are produced at a certain rate. Line 7 introduces a time delay of 3 units. Line 8 marks the production of an item and in line 9 `p` is updated appropriately. Line 10 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. As it will be seen in Fig. 1, if a `wait` is not introduced in line 10, line 11 would cancel the effect of the assertion of `produce` before it can be observed.

```
1   producer(p)
2   {
3     boolean produce;
4     p = 0;
5     produce = false;
6     while(!stop) {
7       wait(3);
8       produce = true;
9       p = p+1;
10      wait(1);
11      produce = false;
12    };
13  }
```

Fig. 1. Producer code.

2.1. Wait statements

In Verus time passes only on `wait` statements. For example, lines 4–6 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.

2.2. Nondeterminism

To illustrate another characteristic of Verus, let us 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 characterizes a nondeterministic choice, and can be modelled in Verus by changing line 9 to

```
9  p = select{p,p+1}.
```

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) (Figs. 2 and 3).

2.3. Process instantiation

In the `main` function, the `producer` and `consumer` processes are instantiated as can be seen in Fig. 4. 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. Parallel process composition is discussed in Section 3.4. Asynchronous behavior can be modeled by using *stuttering*, which introduces nondeterministic transitions and effectively models the desired behavior. We can use nondeterministic assignments to variables to determine if the system will wait for another step or not, as seen in the figure below. Notice that the individual statements can be hidden in a preprocessing step. This technique is described in detail in [6].

2.4. The main function

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`. Details about the verification method can be found in Section 4.

2.5. Periodic execution and deadlines

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. In this case CPU cycles are wasted, the processes are busy

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

Fig. 2. Consumer code.

```
wait(1);
r=select{false,true};
if (r) wait(1);
r=select{false,true};
if (r) wait(1);
```

Fig. 3. A stuttering transition of between 1 and 3 time units.

```
main()
{
  int p, c;
  process prod producer(p),
          cons consumer(p, c);
  spec AG(prod.produce to AF cons.consume)
        MIN[prod.produce, cons.consume]
        MAX[prod.produce, cons.consume]
}
```

Fig. 4. Producer/consumer main function.

waiting. For example, the `consumer` uses the processor even if the `producer` does not generate items. In real systems busy waiting is virtually never used. In order to model systems as realistically as possible, busy waiting should be avoided. In Verus this can done 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 Fig. 5. The first parameter of the

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

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

Fig. 5. Periodic producer.

`periodic` statement is the *start time*, which specifies how many time units the periodic code will idle before starting its execution for the first time. 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). Deadlines can also be defined independent of period using the `deadline`(n) statement.

2.6. Exceptions

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. When a deadline is missed the code designated as handler is executed. After the execution of the exception handler control is passed to the statement following the deadline statement. This can be, for example, the next instantiation of a periodic process.

Fig. 6 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.

2.7. Internal and external variables

There are two types of variables in Verus, *internal* and *external*. Unless assigned a specific value, the value of both types of variables is chosen nondeterministically from all possible values (*true* or *false* for booleans and $0..2^{\text{width}} - 1$ for integers). The two

```
producer(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;
    };
  };
}
```

Fig. 6. Exception handling.

types differ, however, regarding the rules that control when their value changes. The value of an internal variable changes only when assignments are executed. External variables on the other hand model the interaction of the model with the environment. They correspond to inputs from the outside world, and the program has no control over their value. Assignments to external variables are not allowed and their value can change nondeterministically at any transition of the model. The declaration of external variables is preceded by the `extern` keyword.

The data types allowed in Verus 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 allow a very efficient compilation into a state-transition graph. Smaller representations can then be generated, which is critical to the efficiency of the verification and permits larger examples to be handled.

2.8. Related work

There are several other languages for specifying finite-state time-critical systems. Esterel [3] is one such language. It is an imperative language, but its syntax may be very unfamiliar to most designers of time-critical systems, accustomed to programming in C or similar languages. For example, specifying the execution of a periodic process with a deadline is not as straightforward as in Verus. Process algebras are also used to specify time-critical systems [4, 14, 16, 25] but they are also frequently unfamiliar to designers. The disadvantage of using an unfamiliar language is that the designer needs to adopt his/her coding style to that of the new language. Forcing designers to do so can lead to loss of interest in the method, since extra effort has to be spent to use it. Frequently, designers give up on new tools because they cannot afford the time to learn it properly. By using a familiar language for verification we overcome this extra obstacle in making formal methods a practical tool to be used directly by designers.

Modechart [13, 20] and Statechart [23] are other examples of specification languages that can be used to model time-critical systems. They are graphical languages in which nodes represent states, and transitions are explicitly drawn between states. However, complex constructs such as periodic are difficult to draw. Moreover, many systems are too large to be naturally described using languages in which individual states are drawn in the program.

In this work, we use a discrete notion of time. In recent years, there has been considerable research on algorithms that use continuous time [1, 2, 18, 19]. Most of these techniques use a transition relation with a finite set of real-valued clocks and constraints on times when transitions may occur. It can be argued that such algorithms lead to more accurate results than discrete time algorithms. However, an uncountable infinite state space is required to handle continuous time, because the time component in the states can take arbitrary real values. Unfortunately, the representation of this infinite state space can be very expensive in practice. This makes it very difficult to verify many large complex systems using continuous time tools. Discrete-time tools, however, compromise accuracy for efficiency. It is possible to verify larger systems using discrete time, but with less accuracy. In many cases, however, the loss of accuracy is not a problem. For the verification of controllers for many mechanical [17], electrical [8] and chemical processes [24], for example, discrete time is acceptable since many other factors in these controllers already affect the accuracy of measurements. Some of these factors include the use of synchronous circuits, the granularity of operating system timer interrupts which affects the observability of events in the computer, or even the slow speed in which some of these processes operate. In such cases, it is preferable to use a discrete-time tool, since this may simplify verification, speeding it up considerably and possibly enabling the analysis of larger systems.

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. Also, in Section 3.1 the concept of *wait graphs* is introduced. Wait graphs are an abstraction used to keep track of the control flow of the program. The formal semantics is described in Section 3.2. The initial discussion is restricted to a single process, that is, only one flow of execution. The semantics of concurrency in Verus is not discussed until Section 3.4.

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. I_P and T_P will be seen shortly.

3.1.1. 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), $(a, \bar{b}, \bar{c})$ and $(\bar{a}, \bar{b}, c)$, where, for variable v, v means the variable is true in the state, and $\bar{v}$ 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. The size of the BDD representation for a set of states is not directly related to the number of states in it. Frequently, the BDD for a set of states is significantly smaller than another corresponding representation for the same set of states. This is one of the reasons for the efficiency of the method. However, in the worst case the size of the BDD can be exponential in the number of variables in the formula. In this case, the BDD representation is not smaller than an explicit representation for the states, possibly making verification impossible. This problem is known as the state explosion problem. Fortunately, there exist several efficient heuristics to manipulate BDDs that help avoid this exponential blowup of states in the majority of cases [22].

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 $(\bar{a}, \bar{b}, c)$ to state $(\bar{a}, b, \bar{c})$ 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*. Further details about this representation can be found in [6, 22].

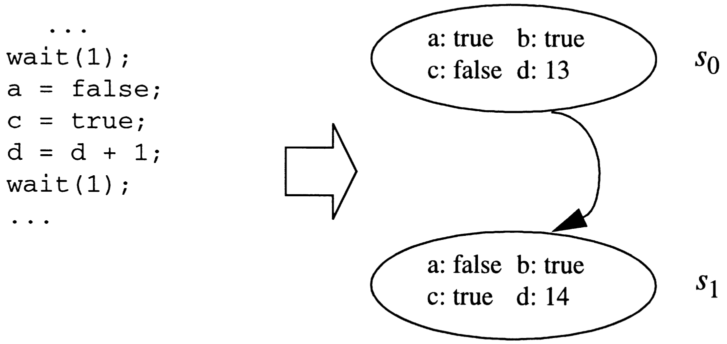
3.1.2. Tracking the control flow – wait graphs

In Verus, the program state 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. As seen in Fig. 7, transitions in the graph occur only when `wait` statements are executed. Each transition corresponds to time elapsing by one unit. Longer waits are modeled by a sequence of unit transitions.

It is easier to understand the behavior of a Verus program by concentrating on its `wait` statements. This is done 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 is 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, $\texttt{wait}_i$ represents the ith occurrence of a `wait` statement in the *source program.* Traversing the same `wait` statement more than once does not change its number. Subscripts have been added to the sample program below to aid the presentation, no subscript exists in actual programs.

As discussed, each `wait` in the program is a state in the wait graph. Transitions between `waits` 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 `waits`. Edges of the wait graph are labelled by a relation T_{ij} between any two states in the state-transition graph. 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'.

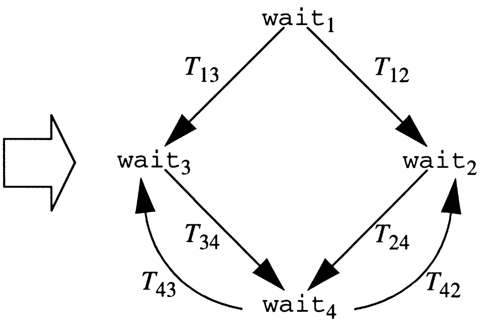
Notice that T_{ij} represents exactly all transitions from s to s' in the state graph such that s and s' are, respectively, the current state of the program before and after control is transferred from $\texttt{wait}_i$ to $\texttt{wait}_j$. This makes it possible to construct the state transition graph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



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

Fig. 7. Wait statement example: if s_0 is the current state at the first `wait`, s_1 will be the current state at the second.

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



(S_is correspond to blocks of assignments)

Fig. 8. A Verus program and part of its corresponding wait graph.

relations between wait statements represents all transitions in the program and their disjunction constitutes the transition relation of the state-transition graph.

3.1.3. 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 ($wait_i$) and where it leads to ($wait_j$). The disjunction of all relations would not maintain consistency of the values of variables after the execution of a sequence of waits.

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 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 (Fig. 8).

3.1.4. 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 nonintuitive behavior because in the set of states defined by $(wc = 0)$ no variable has been initialized. Defining the initial state set as the set of states reachable from $(wc = 0)$ in one step ensures that all variables have been initialized in the initial state.

3.1.5. *Efficient representation of time*

Time is represented by transitions in the state-transition graph. Each transition represents one time unit. This representation is extremely simple, but also extremely efficient. Even though each time unit is individually described in the source program, it is not necessarily explicitly represented internally. The BDD representation used by Verus minimizes the boolean formulas that correspond to these transitions. This usually generates small representations even for very long transitions.

Two cases can be considered when dealing with long transitions represented by a sequence of unit transitions. If at the intermediate steps nothing else happens except for the passage of time, the corresponding BDD will be small since the only event to represent is the increment of the wait counter variable value. If many events happen at intermediate steps, their representation may be complex and the corresponding BDD large. But in this case, it would be necessary to represent these events regardless of which representation of time is used. For example, timed automata use clock variables to represent time [1, 2, 18, 19]. Their value is not necessarily incremented by one, so long time delays *can* be represented by one transition. However, if other events may occur during one such long delay, they must be taken into account, making the verification of such systems considerably more expensive. This is one of the main reasons for the complexity of the verification of time critical systems, represented using timed automata, BDDs or any other method.

The representation proposed in this work does not add a significant overhead to this problem, and it takes advantage of the efficiency of BDD manipulation algorithms. This efficiency can be attested by the various systems verified such as the PCI local bus [8], a heterogeneous time critical system in which multimedia data travels over several different types of communication links [10] and others. For example, the full aircraft controller example described in Section 5 has 15 concurrent processes and we have been able to produce counterexamples with depth greater than 2000 states in minutes using a PC workstation. The depth of a counterexample is relevant because it measures the depth of the breadth first search performed to verify the property. In this example, we have been able to represent the parallel composition of 15 processes and perform an extremely deep breadth first search on a state space with more than 10^{15} states.

3.2. *Formal 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 $Relation \equiv Powerset(ST \times ST)$.

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*, 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

$$R : \mathrm{STMT} \to \mathit{Relation} \times \mathit{Relation} \to \mathit{Relation} \times \mathit{Relation},$$

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. Relations r and t are represented by boolean formulas as explained previously.

The state-transition graph corresponding to a program P is constructed as follows. Given program P, function R constructs $\langle r,t\rangle = R[\![P]\!]\langle wc = 0, false\rangle$, 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 value of program variable `v` in the current state is represented by $v \in V$, and in the next state by $v' \in V'$. A transition is a relation between variables in V and V'.
- A variable wait counter (wc) is introduced in the model. An assignment `wc = i;` exists just before statement $\mathtt{wait}_i$.
- Programs start with the sequence: `wc = 0; wait_0;`
- All programs are assumed to have as the last statement:

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

  This statement guarantees that transitions will be generated for all programs, since transitions are only generated at wait statements. It also ensures that after the program terminates its state will remain unchanged.

3.2.1. Primary 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:

$$E[\![\mathtt{true}]\!] = true, \quad E[\![\mathtt{false}]\!] = false,$$

$$E[\![\mathtt{v}]\!] = v' \quad \text{where } \mathtt{v} \text{ is an internal variable and } v' \in V',$$

$$E[\![\mathtt{v}]\!] = v \quad \text{where } \mathtt{v} \text{ is an external variable and } v \in V.$$

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. 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 the assignment will change the value of v' in the current relation (see semantics of assignments). This is expected, since assignments determine the value of variables 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`. In the case that variable `v` has not been assigned any value, the current relation enforces that the value of internal variables does not change via the clause ($\bigwedge_{v\in \text{ internal variables}} v=v'$) introduced in the current relation at `wait` statements (see $R[\![\texttt{wait}]\!]$). 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.

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.

A final case that must be considered is what happens when the value of the next state variable v' changes after an assignment that refers to its old value. For example, in the code `x = !v`; $v=false$; we must be sure that the new value of v' does not affect the value assigned to x'. This does not happen, however, because during the assignment to v', its old value is assigned to a new variable y which is substituted for v' in r, eliminating cross referencing between the old and new values of v'. This can be seen in detail in the semantics for the assignment statement below.

3.2.2. Boolean expressions

$$E[\![expr_1|\ |expr_2]\!] = E[\![expr_1]\!] \vee E[\![expr_2]\!],$$

$$E[\![expr_1\ \&\&\ expr_2]\!] = E[\![expr_1]\!] \wedge E[\![expr_2]\!],$$

$$E[\![!expr]\!] = \neg E[\![expr]\!]$$

3.3. Statements

3.3.1. Assignments

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

where $v = E[\![\mathtt{v}]\!]$, $Expr = E[\![expr]\!]$ and y is a new variable.This expression computes the strongest post-condition for the assignment $\mathtt{v} = expr$ given r as a pre-condition. If r is the set of valid transitions in the graph since the last `wait`, the expression above determines the largest set of transitions that satisfy the assignment and that satisfy r for variables other than v. Intuitively, it substitutes the previous value of v in r for $Expr$, while maintaining the values of other variables.

$$\begin{aligned} R[\![\mathtt{v} = \mathtt{select}\{expr_1, expr_2\}]\!]\langle r,t\rangle = \mathbf{let}\langle r',t\rangle &= R[\![\mathtt{v} = expr_1]\!]\langle r,t\rangle,\\ \langle r'',t\rangle &= R[\![\mathtt{v} = expr_2]\!]\langle r,t\rangle \mathbf{in}\\ \langle r' \vee r'',t\rangle \end{aligned}$$

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 this disjunction and is omitted for brevity.

3.3.2. Sequential execution

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

3.3.3. Wait statements

$$R[\![\mathtt{wait}_i\mathtt{(1)}]\!]\langle r,t\rangle = \left\langle \left((wc = i) \wedge \bigwedge_{v \in IV} v = v' \right),\ (t \vee r) \right\rangle$$ 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 $\mathtt{wait}_i$ are generated. These transitions are represented by relation r before the `wait` is executed. r is then disjointed with the previous transition relation t. This is the only clause that changes the value of t.

Moreover, the current relation after the execution of $\mathtt{wait}_i$ must reflect the fact that a new set of transitions will be computed. The new relation specifies that transitions start in $\mathtt{wait}_i$ with the formula $(\mathtt{wc} = i)$. The destination of the new set of transitions will be established when the next wait statement is found. At that point, the assignment $\mathtt{wc} = j$ before $\mathtt{wait}_j$ 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 $\bigwedge_{v \in IV} v = v'$ into the current relation. For internal variables this expression guarantees that unless assigned a new value, internal variables maintain their previous value across transitions. 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 an assignment).

3.3.4. 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.

3.3.5. Loops

The representation of a `while` loop can be seen as unrolling the loop into nested if statements: `if` *cond*$\{S_1;$ `if` *cond*$\{S_1;\ldots\}\}$;. We describe below this recursive structure using the *fix* operator, which returns the least fixpoint of the functional given as its argument.

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

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 [26], 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 an infinite loop without waits is the relation $false$, which corresponds to nontermination. But since there are no `waits` in the program, time does not pass. Nontermination 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 nonterminating 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 nonterminating `while` programs are always observable.

3.3.6. 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. Notice that by using multiple `timer` variables it is possible to nest `periodic` and `deadline` statements.

3.3.7. 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.

3.4. Parallel process composition

Given a set of processes defined by their state-transition graphs, it is possible to construct a global state-transition graph corresponding to the environment in which all processes execute concurrently. The concurrency model implemented in Verus is synchronous, that is, one transition in the global model corresponds to exactly one transition in each process.

Given two processes defined by their state transition graphs $G_1 = (S_1, I_1, T_1)$ and $G_2 = (S_2, I_2, T_2)$ we can construct a global state transition graph $G = (S, I, T)$ by:

- $S = \{(s_1, s_2) \mid s_1 \in S_1, s_2 \in S_2$ and $s_1\langle v\rangle = s_2\langle v\rangle$, for all shared variables $v\}$, where $s\langle v\rangle$ denotes the truth value of variable v in state s.

Each state in the global model contains one component in each process. However, one constraint must be satisfied. If a variable is referenced in more than one process, its value in each component of the global state space must be the same. This model guarantees consistency of the values of shared variables.

- $I = \{(i_1, i_2) \mid i_1 \in I_1, i_2 \in I_2$ and $(i_1, i_2) \in S\}$
  An initial state in G is a state in the global model that is an initial state in all processes.
- $T((s_1, s_2), (t_1, t_2))$ iff $T_1(s_1, t_1)$ and $T_2(s_2, t_2)$
  A transition in the global model exists iff it corresponds to existing transitions in each component. Symbolically T is constructed by conjuncting T_1 and T_2. The meaning of the formula representing the global transition relation is that a transition exists if transitions exist in *all* components.

4. The verification algorithms

4.1. CTL and RTCTL model checking

Verus allows the verification of untimed properties expressed as CTL formulas [22] such as `AG (prod.produce -> AF cons.consume)`. This property means that it is an invariant of the system (the `AG` part) that a `produce` is always followed by a consume (the `AF` part). Timed properties can be expressed as RTCTL (real-time CTL formulas [15]. CTL formulas allow the verification of properties such "p will eventually occur", or "p will never be asserted". However, it is not possible to express bounded properties such as "p will occur in less than 10 ms" directly. RTCTL model checking overcomes this restriction by allowing bounds on all CTL operators to be specified [15].

Many important properties of timed systems can be verified using both CTL and RTCTL model checking. For example, we have used it to show the existence of priority inversion in a time-critical system [6]. In this example, we have modeled a simple system in which processes communicate in a nonregular pattern. The main objective is to determine which problems can arise from this communication and how to avoid them. The bounded until operator allows us to determine the existence of priority inversion, and to check that the solution implemented, priority inheritance, avoids the problem.

4.2. Quantitative algorithms

Most verification algorithms assume that timing constraints are given explicitly. 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.

Verus implements algorithms that determine the minimum and maximum length of all paths leading from a set of starting states to a set of final states. It also has 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 system behavior.

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 time critical and non time critical systems. More information about the verification algorithms can be found in [6, 8–10].

5. A more complex example: an aircraft controller

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 aircrafts [21]. Some examples of the Verus code used to model the system are also shown. We conclude with a brief analysis of the results obtained. A full analysis can be found in [7].

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 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. Examples of 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: Processes data received from radars. Identifies/positions targets.
- Display: Updates information on the pilot's screen.

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; processes communicate only through data servers called *monitor tasks*. The time to access shared data is included in the process execution time.

The code below models a process with a 200 ms period and a 3 ms execution time.

```
radar_control()
{
  boolean radar_activate, data_available;
  radar_activate = false;
  data_available = 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 process running time. Initially both variables are false.

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

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

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

Variable `radar_activate` is deasserted, and the process ends this instantiation.

```
  data_available = true;
  wait(1);
  data_available = 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 radar_activate, data_available;

```

```
 7   req1 = false;
 8   radar_activate = false;
 9   data_available = 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     radar_activate = true;
14     while (granted != 1) {\tt }}
15       wait(1); radar_activate = false;
16     };
17     wait(1);
18     radar_activate = false;
```

When execution starts the request variable is asserted and the process must wait until being granted the processor (lines 14–16) before continuing. Line 17 corresponds to the process executing for one time unit.

```
19     while (granted != 1) wait(1);
20     wait(1);
21     while (granted != 1) wait(1);
22     wait(1);
```

The process executes until completion in lines 19–22. But before each step it must check to see if it still has the processor.

```
23     data_available = true;
24     req1 = false;
25     wait(1);
26     data_available = false;
27   };
28 }
```

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

```
29 scheduler(req1, req2,..., time, granted)
30 {
31   time = 0;
32   while (true) {
33     if (req1) granted = 1; else
33     if (req2) granted = 2; else
35     ...
36     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.

```
37      wait(1);
38      if (time < 199) {
39        time = time + 1;
40      } else {
41        time = 0;
42      };
43    };
44 }
```

Line 37 makes the granted variable observable. The scheduler then increments the time variable and repeats the cycle.

In the main module all processes are instantiated, and their schedulability is checked using quantitative timing analysis. A time-critical 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 [7]. 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.

```
45 main()
46 {
47    process P1 p1(time, granted, req1),
48            P2 p2(time, granted, req2),
49            ...
50            SCH scheduler(req1, req2,...,
51                          time, granted);}
52
53    spec
54      MIN(P1.start, P1.end)}
55      MAX(P1.start, P1.end)}
56      MIN(P2.start, P2.end)}
57      MAX(P2.start, P2.end)}
58      ...
59 }
```

6. Conclusions

This work presents a new language to be used in the formal verification of time critical systems, the Verus language. Verus provides a familiar environment to verify time critical systems. The syntax is similar to the syntax of the C language, simplifying the use by nonexperts. It has special constructs to express the timing characteristics of the program naturally and accurately.

Verus programs are compiled into state-transition graphs, which provide a simple and extremely efficient representation of time. The discrete model of time allows the use of fast symbolic algorithms for verification. The simplicity of the internal representation does not restrict the language, however, as attested by the examples of systems that have been verified. Several large complex time critical systems have been verified using Verus. Most examples are either existing industrial applications or use components employed in real systems.

References

[1] R. Alur, C. Courcourbetis, D. Dill, Model-checking for real-time systems, Proc. 5th Symp. on Logics in Computer Science, 1990, pp. 414–425.

[2] R. Alur, D. Dill, in: Automata for modeling real-time systems, Lecture Notes in Computer Science, 17th ICALP, Springer, Berlin, 1990.

[3] G. Berry, G. Gonthier, The ESTEREL synchronous programming language: design, semantics, implementation, Sci. Comput. Programm. 19 (1992).

[4] T. Bolognesi, F. Lucidi, in: A timed full LOTOS with time/action tree semantics, Theories and Experiences for RealTime System Development, World Scientific, Singapore, 1994.

[5] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, J. Hwang, Symbolic model checking: 1020 states and beyond. 5th Symp. on Logics in Computer Science, 1990.

[6] S.V. Campos, A quantitative approach to the formal verification of real-time systems, Ph.D. thesis, SCS, Carnegie Mellon University, 1996.

[7] S.V. Campos, E.M. Clarke, W. Marrero, M. Minea, H. Hiraishi, Computing quantitative characteristics of finite-state real-time systems, Real-Time Systems Symp., 1994.

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

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

[10] S.V. Campos, O. Grumberg, Selective quantitative analysis and interval model checking: verifying different facets of a system, Comput. Aided Verification (1996).

[11] E.M. Clarke, E.A. Emerson, in: Synthesis of synchronization skeletons for branching time temporal logic, Logic of Programs: Workshop, Yorktown Heights, NY, May 1981., Lecture Notes in Computer Science, Vol. 131, Springer, Berlin, 1981.

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

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

[14] J. Davies, S. Schneider, A brief story of timed CSP, Theoret. Comput. Sci. 138 (2) (1995) 243–271.

[15] E.A. Emerson, A.K. Mok, A.P. Sistla, J. Srinivasan, in: Quantitative temporal reasoning, Lecture Notes in Computer Science, Computer-Aided Verification, Springer, Berlin, 1990.

[16] A.N. Fredette, R. Cleaveland, RTSL: a language for real-time schedulability analysis, IEEE Real-Time Systems Symp., 1993.

[17] V. Hartonas-Garmhausen, S. Campos, E. Clarke, A. Cimatti, F. Giunchiglia, Verification of a safety-critical railway interlocking system with real-time constraints, 28th IEEE Internat. Symp. on Fault Tolerant Computing, 1998.

[18] T. Henzinger, P. Ho, H. Wong-Toi, HyTech: the next generation, IEEE Real-Time Systems Symp., 1995.

[19] T. Henzinger, X. Nicollin, J. Sifakis, S. Yovine, Symbolic model checking for real-time systems, Proc. 7th Symp. on Logic in Computer Science, 1992.

[20] F. Jahanian, D. Stuart, A method for verifying properties of modechart specifications, IEEE Real-Time Systems Symp., 1988.

[21] C. Locke, D. Vogel, T. Mesler, Building a predictable avionics platform in Ada: a case study, IEEE Real-Time Systems Symp., 1991.

[22] K.L. McMillan, Symbolic model checking – an approach to the state explosion problem, Ph.D. Thesis, SCS, Carnegie Mellon University, 1992.

[23] J. Ostroff, Formal methods for the specification and design of real-time safety critical systems, J. Systems Software 18 (1) (1992).

[24] S.T. Probst, Chemical process safety and operability analysis using symbolic model checking, Ph.D. Thesis, Dept. of Chemical Engineering, Carnegie Mellon University, 1996.

[25] J. Quemada, D. Frutos, A. Azcorra, TIC: a timed calculus, Formal aspects computing 5 (3) (1993) 224–252.

[26] G. Winskel, The Formal Semantics of Programming Languages, an Introduction, The MIT Press, Cambridge, MA, 1994, pp. 135–139.