

Control-based programming of electro-mechanical controllers

H. Chi Wong^{*} Markus Fromherz^{**} Vineet Gupta^{**} Vijay Saraswat^{**}

1 Introduction

We are developing techniques for model-based construction of software for computationally-controlled electromechanical systems, such as photocopiers. These systems are examples of *real-time, reactive* systems — those that react continuously with their environment at a rate controlled by the environment. Execution in a reactive system proceeds in bursts of activity. In each phase, the environment stimulates the system with an input (e.g. a sensor is tripped by a passing sheet), which starts a computation (e.g., a timer to activate a downstream gate) and responds with the output. The system then stays inactive until the next input comes in from the environment. In such a setting, real-time response is as important as standard liveness and safety requirements.

Our approach to constructing such software is based on the development of compositional and declarative models of the components that make up the electromechanical device. The intuition is that these models can be combined with software systems corresponding to specific tasks (such as simulation, control code generation etc.), via customized reasoning engines. These engines will take as input the specification of a system configuration, the software architecture and the model of the components to semi-automatically produce the target system. This general approach can be used for producing a variety of systems including controllers, simulators, testers, productivity analysers and diagnosis tools.

In this paper, we consider the nature of the language that should be used to describe the output control code. Even though the proper handling of time is an integral aspect of such software, control software has traditionally been programmed in conventional languages that do not support adequate constructs for handling time. Traditionally, real-time software has been written in untimed languages such as C augmented with some new primitive constructs for real-time operations. This has obvious drawbacks — it does not help in analysis, and anyway it is difficult to get real-time software correct. More recently, attempts have been made to provide an application and platform independent Real-Time Application Programmer's Interface (API), within a language such as C++. Typically, in a setting like this, real-time programs are written as C++ programs that make calls to run-time routines implementing timers, event handlers, event queues etc. Hardware I/O channels are associated with memory locations. Call-back procedures are registered with an event handler, to be called when an event is raised. Priority levels

^{*} Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa 15213, hcwong@gs88.sp.cs.cmu.edu

^{**} Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, Ca 94304 {fromherz, vgupta, saraswat}@parc.xerox.com

may be associated with task queues. Sometimes an implementation may provide multiple threads.

Designs such as these have traditionally been confronted with the tradeoff between modularity and efficiency. Modularity demands that the code representing logically distinct modules (e.g. tray-feeder and the elevator) be represented separately. Efficiency demands that there be no unnecessary run-time overhead associated with communication between various interacting components of code. Traditionally, this issue has been resolved in favor of efficiency. In response to an event, an event handler is invoked. Its body typically looks like a finite state machine, setting the values of various state variables, and checking them to determine what should be output, and what the current state should be changed to. There is usually no possibility of a call out to another event-handler during the process of responding to an event; hence concurrency in response to a stimulus is not supported. (If multi-threading is supported, different stimuli may still be responded to by different threads.) This makes the software difficult to develop, and even more difficult to maintain.

In this paper we compare the *control-oriented* paradigm with the *data-oriented* paradigm for programming reactive controllers. In the control-oriented paradigm, time is embodied in the control structure of the program; in the data-oriented paradigm, timed interactions are mediated through explicit data-structures representing clocks, timers and suspension lists. We argue, through a detailed example, that the control-oriented paradigm supported by synchronous programming languages such as TCC [SJG93, SJG94] is superior to the traditional data-oriented paradigm in programming control software, with respect to requirements such as modularity, efficiency, time handling, verification and real-time performance measurements. A result of integrating synchronous [BG92, HCP91, GBGM91, Har87] and constraint languages [SRP91], TCC incorporate timing constructs that express the *pattern of interaction*, over time, between the controller and the environment. TCC allows the expression of modular programs, exploiting powerful clocking primitives for timing. At the same time, programs can be compiled into real-time finite-state machines with minimal runtime overhead. Since programs can be viewed as formulas in a (temporal) logic, it also becomes possible to use standard techniques for proving properties.

The rest of the paper is structured as follows. Section 2 describes a sample device and its control. Section 3 presents a data-oriented implementation of the control algorithm: it is in C++ and uses function calls from a library that supports event-driven computation. Section 4 presents briefly TCC languages, emphasizing their timing constructs and shows how they can be used effectively to model real-time, concurrent and reactive tasks. Section 5 presents a control-oriented program written in TCC. Section 6 mentions the various advantages of the control-oriented TCC program. Section 7 concludes with a report on the current status of our work and a discussion of future work.

2 A Sample Device and Its Control

We describe the hardware and the control algorithm for the paper feeder component of a simple photocopier. The feeder is the device which pulls sheets of paper into the paper path of a copier, in response to a signal from the scheduler.

2.1 The Paper Feeder

The Components For the purposes of this discussion, the paper feeder consists of (see Fig. 1):

- a paper tray (6), that holds a stack of paper;
- an elevator (5) attached to the paper tray, through which the height of the stack of paper can be regulated.
- an acquisition roll (4), that can rotate and can be dropped onto or lifted away from the stack of paper. Its state is determined by a solenoid that can be energized or de-energized. When the solenoid is turned on, the acquisition starts the rotational movement around its own axis and is dropped onto the stack of paper; when the solenoid is turned off, the acquisition roll stops rotating and is lifted from the stack;
- a stack height sensor, attached to the acquisition roll;
- a retard nip (3), constituted by a feeder roll (that rotates when the feeder motor is on) and a retard roll rotating in the opposite direction to prevent more than one sheet from being fed; a sensor (2) located at the wait station, where the sheet of paper being fed is held waiting for the purposes of synchronization with the rest of the machine functions.

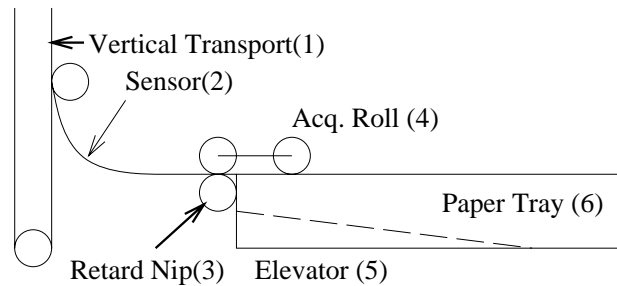


Fig. 1. The diagram of the feeder

Principle of operation. The **acquisition roll** contributes in paper transportation and paper stack height detection. When the roll is in contact with a sheet of paper, its rotational movement drags the sheet of paper into the retard nip. The paper stack height is determined by how much the acquisition roll has been stretched out to reach the stack of paper. This information is captured by the stack height sensor.

The **elevator** is important for height adjustment, since a sheet of paper can only be dragged into the retard nip when the acquisition roll is rotating and in contact with it, and the stack of paper is high enough so that its top sheet can get into the retard nip. The stack height should be adjusted before each feed.

The **feeder roll** transports a sheet of paper along the initial segment of the paper path. The retard roll pushes back any additional sheets of paper that may have been sticking to

the top sheet. The **wait station sensor** is responsible for detecting the arrival and the departure of sheets of paper in the feeder. It is important both for synchronization purposes and for jam detection.

2.2 The Paper Feeder Control

The feeder control provides the following basic functions:

1. Feeder initialization: the paper tray is set at an adequate height;
2. Feed function, with the following subfunctions:
 - preprefeed**: a sheet of paper lying completely inside of the tray has its leading edge inserted into the retard nip;
 - prefeed**: a sheet of paper that has its leading edge originally at the retard nip is brought to a position where its leading edge is at the wait station (sensor);
 - feed proper**: a sheet of paper that has its leading edge at the wait station is fed into the rest of the paper path. Note that we are using the word *feed* both for the whole cycle and for the 3rd subcycle of it. It should be clear from the context as to which we are referring.

The initialization happens when a photocopier is turned off and then on, or when the tray of paper is pulled out and then in. In either case, the first thing to be done is height adjustment (of the paper stack). For that, the acquisition roll should be turned on. The amount of time for which it is kept on is determined by how fast the stack of paper reaches the adequate height (this, of course, requires some action from the elevator, if necessary) which, in turn, is determined by the amount of paper that exist in the paper tray. When the height is adjusted, the acquisition roll is turned off.

As part of the initialization, a *daemon* process responsible for the feeding process is started up and this process keeps watching for signals from the system scheduler that signals a feed.

Depending on whether or not it is the first feed after the copier is initialized, the feeder performs or skips the preprefeed subcycle. If it is the first feed, then it performs preprefeed followed by prefeed and feed; otherwise, it will go straight to the prefeed part of the cycle as the preprefeed subcycle was done as the last action of the previous feed cycle. After feeding (subcycle) a sheet of paper, the feeder performs preprefeed and adjusts stack height.

2.3 The Algorithm

Having in mind the high level functionality just described, the feeder hardware can be controlled by the following algorithm:

Initialization of the paper stack height: Occurs right after tray out/tray in and power off/power on:

1. Drop the acquisition roll onto the stack (to initialize the stack height);
2. Check if the stack height is low;

3. If the stack height is low, turn the elevator motor on immediately;
4. As soon as the stack height reaches an adequate level, turn the elevator motor off;
5. Lift the acquisition roll immediately after the elevator motor is turned off.

First prefeed: This code is activated when the feed is the first after initialization:

1. Turn the feed motor on to prefeed from the stack (this activation obeys the system timing and the signals from the scheduler);
2. Drop the acquisition roll (this should be done at the same time as the feed motor is turned on);
3. Lift the acquisition roll after a certain fixed time;
4. Turn the feed motor off (at the same instant the wait station sensor senses the arrival of the leading edge of the sheet).

Normal run: This algorithm is used when the requested feed is not the first following the initialization. The leading edge of the sheet should be at a short distance past the retard nip:

1. Turn the feed motor on to prefeed (this obeys the system timing and signals from the scheduler);
2. Turn the feed motor off when the leading edge of the paper reaches the wait station sensor;
3. Turn the feed motor on (for **Feed from wait station**. It is dictated by the system timing);
4. Drop the acquisition solenoid roll onto the stack (this is given by time elapsing after the feed motor is on for **Feed from wait station** and is the time at which the trailing edge of the traveling paper is at the leading edge of the stack);
5. Turn the feed motor off (given by a time elapsing after the feed motor is on for **Feed from wait station**, and is the time at which the trailing edge of the traveling sheet of paper is at a short distance beyond the retard nip);
6. Check if the stack height is low (this checking occurs a predetermined amount of time after the feed motor is turned off);
7. Lift the acquisition roll (end of the feed cycle) (triggered by a fixed time elapsing after **Feed from Wait Station**);
8. Turn the elevator on (to elevate the stack height. It is started immediately after the end of the stack height check, if the stack is low);
9. Turn the elevator off (to stop elevating. Given by time elapsing after the previous elevator turning on).

3 Data-centered implementation

The first implementation we discuss is written in C++ and uses function calls from a library that supports event-driven computation. Since the C++ implementation is several pages long, we present and analyze only parts of it in this paper.

Consider the reactive task **Feed Elevate** that controls the tray elevator motion during the feed cycle, based on sensor and timer activation. We give only a schematic version of the code, to illustrate the way in which timing information is being maintained.

```
FeedElevate::FeedElevate(SheetSensor*   heightSensor,
                        IoRegister*     elevator,
                        IoRegister*     solenoid,
                        TimeRelative*   elevateInRun)
: ReactiveTask("FeedElevate", 5),
  elevator (elevator),
  solenoid (solenoid),
  heightSensor (heightSensor),
  trayLow (false),
  elevateTime (elevateInRun)
{ startTimer = new Timer("start",rtClock);
  stopTimer = new Timer("stop",rtClock);
  elevateTimer = new Timer("elevate",rtClock);
  Attach(heightSensor->ChangedToSheet(), sensesSheet);
  Attach(startTimer->Expired(), startChecking);
  Attach(stopTimer->Expired(), stopChecking);
  Attach(elevateTimer->Expired(), stopElevating); }

void FeedElevate::StateMarker(Enum message, Natural count)
{
  switch (message)
  { case startChecking:
    if (*heightSensor == SheetState::noSheet)
    { trayLow = true;
      *stopTimer = stopCheckingSensor; }
    break;
    case stopChecking:
    if (trayLow)
    { *elevator = IoState::on;
      *elevateTimer = 50 + *elevateTime; }
    break;
    case sensesSheet:
    trayLow = false;
    break;
    case stopElevating:
    *elevator = IoState::off;
    StopResponding();
    break; } }
```

In general, declarations of new timers look as follows:

```
startTimer = new Timer("start",rtClock);
```

Notice that they are referenced to a real-time clock.

To be of any use, however, the timers need to be assigned values, and this is done, for example, in:

```
*startTimer = startCheckingSensor;
```

where `startCheckingSensor` is a defined numerical constant.

Also, if timer expirations are to trigger any action, they need to be registered as a valid event before they can be handled by an event handler. And this is done by:

```
Attach(startTimer->Expired(), startChecking);
```

There are also other types of events such as those raised by environment sensors:

```
Attach(heightSensor->ChangedToSheet(), sensesSheet);
```

The core for `Feed Elevate` is implemented as an event handler (`StateMarker`) that responds to messages from the outside:

```
messages = {startChecking, stopChecking, sensesSheet,  
            stopElevating}
```

State variables record various timers that may be active at any particular time (e.g. `failureTimer`). Other variables correspond to input (e.g. `heightSensor`) or output values for the controller (e.g. `elevator`).

Examining the code in C++, we can see that this version of the controller is structured in terms of a `case` statement in which several *events* can be handled. Inside each event handler, typically we have timer setting, activation and deactivation of actuators and generation of further events. The events, in their turn, are raised either by time expiration or are signals from hardware sensors. In other words, the management of time is done by counter-like *data structures* and represents a significant effort in writing the control code, not to mention that the related code is spread throughout the program.

In addition to the amount of overhead in coding, the implementation of the *real-time clock* in terms of data structures means that the control flow is dynamic information, known only at the execution time.

4 Timed Concurrent Constraint (TCC) Languages

In this section, we introduce TCC languages [SJG94], describing briefly the computational model, the basic constructs and some of the combinators. Those interested in more detailed information, including the formal syntax, the denotational and the operational semantics should refer to [SJG93].

TCC languages resulted from the integration of synchronous and constraint programming. More precisely, they are a *timed* extension of *Concurrent Constraint* (CCP) languages [SRP91]. We will show how to write a range of constructs for expressing timeouts, preemption and other complicated patterns of temporal activity in the basic model and language-framework.

CC Languages

The CCP paradigm is based on the idea of a collection of (possibly distributed) agents communicating by imposing and checking pieces of partial information - constraints - on shared variables. The basic departure from the conventional imperative languages is, therefore, the replacement of the notion of store as valuation of variables, central to von Neumann computing, with the notion that the store is a constraint, that is, a collection of pieces of partial information about the possible values that variables can take. Computation progresses via the accumulation of constraints. Concurrency arises because any number of agents may simultaneously interact with the store. The usual notions of “read” and “write” are replaced by *ask* and *tell* actions. A *tell* operation takes a constraint and conjoins it with the constraints already in the store; *tells* are executed asynchronously. Synchronization is achieved via the *ask* operation: *if c then A* takes a constraint *c* and uses it to probe the structure of the store: if the store contains enough information to entail *c*, it succeeds and starts *A*; if the store is not strong enough to entail *c*, it waits.

TCC Languages

In order to have a language for real time systems, it is not only necessary to detect when an event happens (“positive information”), but also when an event did not happen (“negative information”). The problem is that since the CC framework is inherently monotonic, information can never be withdrawn. Negative information, on the other hand, is inherently non-monotonic — it can be invalidated by later information.

However, once a computation is over, it is safe to conclude that some information did not appear, and it is only then that negative information can be inferred. Thus we wait for the *quiescent* points of a computation, which are states of the system in which no more positive information is being generated. All negative information is inferred at this point, and can be used in future. Now *time* can be introduced by identifying quiescent points as the markers that distinguish one time step from the next. This allows the introduction of primitives that can trigger an action in the *next* phase of computation (time tick) if some event did not happen through the extent of the *previous* phase. In other words, the absence of a signal in a period can be detected only at the end of a period, and hence, can trigger activity only in the next interval.

Operationally, this is what happens: consider the situation in which computation in the current time instant has quiesced. This means that all reductions that could have been caused because of the entailment of Positive Asks have been done, all Negative Asks whose antecedents are entailed have been eliminated, and computation has not aborted. Computation can now progress to the next time instant. The active agents at the next time instant are the bodies of the remaining Negative Asks agents or the bodies of agents within a Unit Delay. All other agents and the current store will be discarded.

To summarize, computation in TCC proceeds in intervals. During the interval, positive information is accumulated and detected asynchronously, as in CCP. At the end of the interval, the absence of information can be detected, and the constraints accumulated in the interval are discarded. No mechanism is provided for the implicit transfer of positive information across time boundaries to maintain the bounded size of the constraint store; this must be done explicitly by the programmer by using the basic combinators.

We now identify basic processes and process combinators in the model. They fall into two categories:

- **CCP constructs** (they do not cause extension over time): Tell (c), Parallel Composition ($[,]$) and Timed Positive Ask ($\text{if } c \text{ then } A$).
Tell adds a constraint c to the current store. *Parallel Composition* allows two agents to run concurrently. *Timed Positive Ask* is the process that checks if the current store is strong enough to entail c ; and if so, behaves like A . Timed positive ask and Tell combine to allow synchronization capabilities.
- **Timing constructs** (cause extension over time): Timed Negative Ask ($\text{if } c \text{ else next } A$), Unit Delay ($\text{next } A$), and Abortion (abort).
Unit Delay starts a process to be started in the next time instant. *Timed Negative Ask* is a conditional version of Unit Delay, based on detection of negative information. It causes a process to be started in the next time instant if, on quiescence of the current time instant, the store was not strong enough to entail some information. *Abort* shuts down all computation in the system.

Since TCC languages are defined as an algebra of processes, one can define a number of combinators from the basic constructs. We now show some of the interesting ones that capture natural patterns of temporal activity.

1. Unconditional persistence: $\text{always } A$ is the agent that behaves like A at every time instant.
2. Conditionals: $\text{if } c \text{ then } A \text{ else } B$ is the agent that behaves like A if c holds in the current time instant; otherwise it behaves like B from the *next* time instant onwards.
3. Extended Wait: $\text{whenever } c \text{ do } A$ is the agent that suspends until the first time instant at which c is true; it then behaves like A .
4. Watchdogs: $\text{do } P \text{ watching } c$ behaves like P until a time instant when c is entailed; when c is entailed, P is killed from the next time instant onwards. $\text{do } P \text{ watching } c \text{ timeout } B$ activates a handler B when P is killed.
5. Delays: $\text{next } (\text{Num}, \text{Signal}) \text{ do } A$ waits for Num occurrences of Signal and then starts doing A .

In TCC it is not possible (due to semantic reasons) to specify that a computation be aborted in the current instant on receipt of positive information from the environment. One can, however, describe a construct that aborts the computation at the next stable instant, that is at the next moment of quiescence.

Before proceeding to the next section, we should mention that the logic underlying TCC languages is intuitionistic linear time temporal logic. Also, since the presentation is done for TCC languages in general, it is valid for the particular language we will use: TIMED GENTZEN (TG). TG is a concrete TCC language instantiated over the constraint system GENTZEN, which consists of a set of uninterpreted tokens, with the trivial entailment relation: $c_1, \dots, c_n \vdash d$ iff $d = c_i$ for some $i : 1 \leq i \leq n$.

5 Control-Oriented Programming

We now present a program that directly exploits the combinators of TCC. The program is divided up into logical pieces, each corresponding to a physical activity (e.g. `feed`, `prefeed_from_stack`) etc. Each piece consists of one or more assertions running in parallel. We present the program for the entire feeder mechanism here.

In the following, we use the following conventions. Names without hyphens are reserved for names of signals, and with hyphens for names of agents. The external signals are `tick` (the clock ticks), `tray:in` and `tray:out`, `sync` and `paperAtS1` (from the wait station).

```
feeder ::
  whenever tray:in do
    do [init_stack_height,
        whenever sync do [
            first_feed,
            next waiting_to_feed_cycle]]
    watching tray:out
    timeout [{feedElevator:off},
            {acqSolenoid:off},
            {feedMotor:off}].

init_stack_height(init) ::
  [{acqSolenoid:on},
   next(startCheckingStkHtInit, tick) do [
     if lowTray:false then {acqSolenoid:off},
     if lowTray:true then [
       next(endCheckingStkHtInit, tick) do [
         if lowTray:false then {acqSolenoid:off},
         if lowTray:true then [
           {feedElevator:on},
           whenever lowTray:false
             do {feedElevator:on, acqSolenoid:on}]]]]].

first_feed :: prefeed_from_stack, feed.
waiting_to_feed_cycle :: always if sync then regular_feed.
regular_feed :: prefeed_from_retard, feed.

prefeed_from_stack :: [
  {feedMotor:on},
  {acqSolenoid:on},
  next(prefeedFromStackPeriod, tick) do {acqSolenoid:off},
  whenever paperAtS1 do {feedMotor:off}].

prefeed_from_retard :: [
  {feedMotor:on},
  whenever paperAtS1 do {feedMotor:off}].

feed :: [
```

```

next(feedFromWaitTime) do [
  {feedMotor:on},
  next(startPreprefeedTime, tick) do {acqSolenoid:on},
  next(endPreprefeedTime, tick) do [
    {feedMotor:off},
    next(startCheckingStkHtRun, tick) do
      if lowTray:true then
        next(endCheckingStkHtRun, tick) do
          if lowTray:true then [
            {feedElevator:on},
            nex(elevatingPeriod, tick) do {feedElevator:off}]],
          next(liftAcqSolenoidTime, tick) do {acqSolenoid:off}]].

```

The TCC code does not make use of any timer. Instead, time handling is mainly done by the combinator **next(interval) do A**, which “hardwires” the notion of time passage into the program structure (this is exactly the characterization of *control-based* paradigm). Furthermore, the pattern **whenever c_1 do A_1 watching c_2 timeout A_2** , which appears in several places, captures precisely the basic interaction between the controller and its environment, namely,

Upon the raising of the condition c_1 , handle it with the agent A_1 . And in the case A_1 performs an action that extends over time, interrupt it when the condition c_2 is raised, and handle it with the agent A_2 .

6 Analysis of Control-Oriented Programs

Apart from brevity, there are several advantages to the control-based paradigm. We discuss some of these below.

Time analysis. One of the requirements that real-time control systems need to fulfill is that of *time criticality*. This means that, in addition to being correct, they need to produce the right outputs within time constraints established by the specification of the system. Thus, given any program, it is necessary to check whether it satisfies this requirement. This is made considerably easier in the control-oriented style of writing code, where the time-critical components are separated from the rest of the code through `next(interval, sig) do A`. For example, we have

```

next (10, tick) do next (20 tick) do A =
next (30, tick) do A

```

In the data-oriented style, the C++ code, time is treated just like another data item. Thus verifying the timing properties of the program involves a data flow analysis of the whole C++ program, which is quite difficult. Thus, since in most such programs the time component of the program is quite simple, by separating it from the data-handling the control-oriented program is much more amenable to time analysis.

Code Optimization Another advantage of the control-oriented programs is that they manipulate time within the control structure of the program, which is available to the compiler. In data-oriented programs, time is treated as data, and is manipulated by data structures, which are set up and interpreted at runtime. This makes static analysis of code much more complex, and rules out several easy optimizations. Thus for example if we needed to delay an operation in the code, the data-oriented program would set up a timer, which uses variables which are decremented at runtime. However the control-oriented program would set up an automaton with several states following each other to represent the passage of time. These states are created by the compiler, and the resulting automaton easily shows the exact delay. This information can be used for further analysis by the compiler. In the data-oriented version, this information is “lost in the data”, and does only appear at runtime through the trace of an execution.

Compositional Specification. Given the problem of composing parallel finite-state machines, we have seen two solutions: either explicitly compose them to achieve a fine granularity of interaction, or effectively execute their transitions sequentially. The latter has the problems of development and maintainability mentioned above; the latter is only applicable where control processes don't have to cooperate. Thus the data-oriented style prevents us from writing code modularly. The control-oriented style allows us to write and reason about separate pieces of code, however, at runtime it is compiled into a single automaton, getting us both efficiency and determinacy.

Formal verification. The compilability of TCC programs into FSMs enables the use of model checking techniques which allow the proof of properties such as liveness, safety, etc. This is extremely useful for real-time systems, since traditional methods like simulation can sometimes overlook bugs in the code. Verification of programs written in traditional languages such as C++ is nearly impossible.

One of the existing performance analysis techniques is *Real-time Model Checking* [CCMM94], an extension of the original *Symbolic Model Checking* [BCMDH90, McM92]. This enables us to compute quantitative information about finite-state real-time systems, and can be used to determine which conditions hold at which time, and whether a system can be scheduled properly. We see in *control-oriented* TCC programming an opportunity for getting performance measures of a system out of its TCC prototype, since its compiled code (FSMs that have all the timing behaviour “hardwired” in themselves) is the input required by *real-time symbolic model checking* algorithms.

7 Conclusion

In this paper, we report on insights obtained from using a TCC language to program real-time control systems. One can use either the data-oriented paradigm or control-oriented paradigm. They differ from each other in the way they control and register the passage of time. In the data-oriented paradigm, one uses data structures to this end. Timers are implemented and used explicitly. In the control-oriented paradigm, language combinators are used to “hardwire” timing passage into the program structure itself. One of the advantage of this second approach is that it is possible to apply real-time model checking techniques to the compiled code and obtain performance measures of the system.

Currently, we are working with a photocopier, modeling its paper path controller and the simulation environment necessary to execute it. The near-future goal for this part of the project is to use model-checking techniques to prove the correctness of the system, as well as to obtain its performance measures.

References

- [BCMDH90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, 1990.
- [BG92] G. Berry and G. Gonthier. The ESTEREL programming language: Design, semantics and implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.
- [CCMM94] S. Campos, E. Clarke, W. Marrero, and M. Minea. Computing quantitative characteristics of finite-state real-time systems. Technical report, CMU-CS-94-147, Computer Science Department, Carnegie Mellon University, May 1994.
- [GBGM91] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with SIGNAL. In *Special issue on Another Look at Real-time Systems*, Proceedings of the IEEE, Special issue on Another Look at Real-time Systems, September 1991.
- [HCP91] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous programming language LUSTRE. In *Special issue on Another Look at Real-time Systems*, Proceedings of the IEEE, Special issue on Another Look at Real-time Systems, September 1991.
- [Har87] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8:231–274, 1987.
- [McM92] K. L. McMillan. *Symbolic model checking — an approach to the state explosion problem*. PhD thesis, SCS, Carnegie Mellon University, 1992.
- [SJG93] Vijay Saraswat, Radha Jagadeesan, and Vineet Gupta. Foundations of timed concurrent constraint programming. Proceedings of the IEEE Symposium on Logic in Computer Science, Paris, July 1994.
- [SRP91] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of Eighteenth ACM Symposium on Principles of Programming Languages, Orlando*, January 1991.
- [SJG94] Vijay Saraswat, Radha Jagadeesan, and Vineet Gupta. Proceedings of the NATO ASI Workshop on Constraint Programming, Springer-Verlag.