Journal of Universal Computer Science, vol. 13, no. 6 (2007), 682-700
submitted: 19/1/07, accepted: 22/3/07, appeared: 28/6/07 © *J.UCS*

CML: C Modeling Language

Frederico de Oliveira Jr., Ricardo Lima,
Márcio Cornélio, Sérgio Soares
(Pernambuco State University (UPE), Brazil
{fgaoj,ricardo,marcio,sergio}@dsc.upe.br)

Paulo Maciel, Raimundo Barreto,
Meuse Oliveira Jr., Eduardo Tavares
(Federal University of Pernambuco (UFPE), Brazil
{prmm,rsb,mnoj,eagt}@cin.ufpe.br)

Abstract: Non-functional requirements such as performance, program size, and energy consumption significantly affect the quality of software systems. Small devices like PDAs and mobile phones have little memory, slow processors, and energy constraints. The C programming language has been the choice of many programmers when developing application for small devices. On the other hand, the need for functional software correctness has derived several specification languages that adopt the Design by Contract (DBC) technique. In this work we propose a specification language for C, called CML (C Modeling Language), focused on non-functional requirements. CML is inspired on the Design By Contract technique. An additional contribution is a verification tool for hard real-time systems. The tool is the first application developed for CML. The practical usage of CML is presented through a case study, which is a real application for a vehicle monitoring system.
Key Words: specification language, non-functional requirements, c programming language
Category: D.2.1, D.2.4, D.3.3

1 Introduction

A software specification is intended to describe the structure and functionality required for a system[Gannon et al. 1994]. The specification is useful to understand the system and to eliminate errors in the later phases of the development cycle.

A number of specification languages have been designed to be annotated directly in the source code. Some of these languages adopts the Design by Contract (DBC) [Meyer 1992, Meyer 1997] technique. Contracts are a breakthrough technique to reduce the programming effort for large projects. Contracts are the concept of preconditions, postconditions, errors, and invariants. The idea of a contract is just an expression that must evaluate to true. If it does not, the contract is broken, and by definition, the program has a bug in it. Contracts form part of the specification for a program, moving it from the documentation to the code itself. And as every programmer knows, documentation tends to be

incomplete, out of date, wrong, or non-existent. Moving the contracts into the code makes them verifiable against the program.

The concepts of DBC were introduced in Eiffel [Meyer 1997]. The Java Modeling Language (JML) follows the design by contract paradigm. It is a specification language for Java programs, using Hoare style pre- and postconditions and invariants [Hoare 1969]. The specifications are added as annotation comments to the Java program, which hence can be compiled with any Java compiler. There are various verification tools for JML, such as a runtime assertion checker [Cheon and Leavens 2002] and the Extended Static Checker (ESC/Java) [Flanagan et al. 2002].

Small devices, such as PDAs and mobile phones, have constrained resources, like memory, processor power, and energy. Therefore, applications developed targeting such devices cannot ignore these resources limitations. The C programming language has been the choice of many programmers when developing application for small devices. The capacity for manipulating low level resources and the existence of efficient compilers justify the popularity of C for these applications.

Due to side effects caused by pointer manipulation is difficult to prove the functional correctness of C programs. Thus, although most C programmers like the idea of DBC, they abandon DBC because it is too inelegant, relies too much on macros, and is too weak without language support. On the other hand, C is widely adopted to implement application with stringent resource constraints (memory, time processing, communication cost, and energy consumption, for example). Therefore, it is intuitive to define a specification language to describe non-functional requirements of C programs. Indeed, most C programmers invent their own strategy to define such requirements, for instance, in form of comments, informally included in the source code. Additionally, the specification language should be associated with verification tools. It is important to automatically check if the non-functional requirement was fulfilled.

This work proposes a specification language for C focused on non-functional requirements, inspired in the DBC paradigm, called C Modeling Language (CML). Moreover, the paper contributes with a tool for hard real-time systems based on CML. The tool receives a C program, composed of several tasks, annotated with time restrictions, scheduling method (preemptive and non-preemptive), arbitrary inter-task relations (precedence and exclusion), task-to-processor allocation. It automatically looks for a feasible schedule. If a schedule is found, the tool generates a scheduler to control the tasks' execution. It is worth observing that this is pre-runtime scheduling policy, which is fundamental to satisfy timing requirements established in the CML specification.

The main contributions of this work are: the CML specification language focused on non-functional requirements; a tool for hard real-time systems based

on CML that analyzes C programs according to the defined specification; and a case study that validates the proposed language (CML) and the analysis tool in a real application.

2 CML Description

The C Modeling Language (CML) is a specification language developed to describe non-functional requirements of applications implemented through the C programming language. CML is particularly useful for applications with stringent constraints in terms of time, memory, area, power, and other limited resources.

Figure 1 depicts an overview of the CML environment. Similar to JML, programmer includes annotations in the C source code in form of comments. The CML compiler translates the annotated C code into the file format of the verification tool employed to check the system against the specified non-functional requirements.

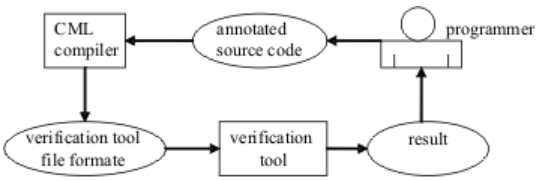

Figure 1: An overview of the CML environment

The CML specification is placed into comment blocks, between `/*!` and `*/` patterns. The pattern `/*!` indicates the beginning of the specification. The pattern `*/` establishes the end of a specification block. Figure 2 presents a simple example of a CML specification.

```
/*!
 * @attribute value
 */
```

Figure 2: A simple CML specification

The complete set of annotation elements proposed for CML is presented in Appendix A. Table 1 lists a subset of CML defined for specifying hard-real time systems. We will focus on this subset to illustrate the practical usage of CML.

Constructor	Description	Format
@task	task name	String
@processor	processor where the task will be executed	String
@scheduling	task scheduling model	NP (Non-preemptive) or P (Preemptive)
@phase	task phase time	Integer
@release	task release time	Integer
@wcet	task worst case execution time	Integer
@deadline	task deadline time	Integer
@period	task period time	Integer
@precedes	tasks preceded by this task	List of tasks between the tokens `{` and `}` separated by comma
@excludes	tasks excluded by this task	List of tasks between the tokens `{` and `}` separated by comma
@sends	message sent by this task	This attribute is followed by: 1. Message name 2. Bus name 3. Worst case communication time 4. Receiver Task

Table 1: Subset of CML for hard real-time systems

Figure 3 presents a CML specification for task T1 (`@task T1`), which belongs to a hard real-time system. According to the specification, T1 cannot be preempted (`@scheduling NP`) and must execute in processor P1 (`@processor P1`). The attributes `@release`, `@period`, `@phase`, `@deadline`, and `@wcet` are related to the task timing constraints and expressed in Time Task Units (TTUs). A TTU is the smallest indivisible granule of a task, during which a task cannot be preempted by any other task. The attributes `@precedes` and `@excludes` specifies the relation between tasks. In the specification example, T1 precedes tasks T2 and T5. Consequently, T2 and T5 can only start executing after T1 has finished (`@precedes {T2,T5}`). `@excludes {T3}` indicates that task T1 excludes T3. Therefore, no execution of T3 can start while T1 is executing. Eventually, message M1 is sent by T1 to the task T4 through the communication bus B1. The communication should take, in the worst case, 17 TTUs (`@sends M1 T4 17 B1`). Section 3 provides a detailed discussion about each attribute included in the CML specification for hard real-time systems.

The CML compiler developed in this work translates the CML specification into a XML file. This file is read by an application developed in this work, which is a software synthesis tool for embedded hard real time systems described in Section 3. The XML file equivalent to the CML specification in Figure 3 is presented in Figure 4.

```
/*!
 @task T1
 @scheduling NP
 @processor P1
 @release 1
 @period 9
 @phase 1
 @deadline 9
 @wcet 1
 @precedes {T2,T5}
 @excludes {T3}
 @sends M1 T4 17 B1
*/
 void T1(){ //task code }
```

Figure 3: Example of a task specification

3 A CML Based Software Synthesis for Embedded Hard Real-Time Applications

The practical usage of CML depends on the implementation of verification tools integrated with a C programming environment. An additional contribution of this work is the development of a software synthesis tool for embedded hard real-time applications.

Embedded hard real-time systems are dedicated computer applications having to satisfy stringent timing constraints, or rather, they must guarantee that critical tasks finish before their deadlines. A failure to meet deadlines may have serious consequences such as resources damage or even loss of human life. Software synthesis has become a key problem in design of embedded hard real-time systems, since the software is responsible for more than 70% of functions in such systems [Su and Hsiung 2002].

Scheduling is very important in embedded real-time systems. There are two general approaches for scheduling tasks: runtime and pre-runtime scheduling. The former approach computes schedules on-line as tasks arrive, through a priority-driven strategy. However, in some cases the runtime scheduler is unable to find a feasible schedule, even if such schedule exists [Xu and Parnas 1993]. On the other hand, a pre-runtime scheduler computes the schedule entirely off-line. This strategy improves processor utilization, reduces context switching, makes execution predictable, and excludes the need for complex operating systems.

This work focuses on embedded *hard* real-time systems. We decided to adopt a pre-runtime scheduling approach. To find a feasible schedule, we perform a state space exploration, since it presents a complete automatic strategy for verifying finite-state systems [Godefroid 1996]. The scheduled code is generated by traversing the timed Labelled Transition System (LTS), which represents a feasible schedule. Transition's instances visited are substituted by the respective

```
<realtime-table>
  <task release="1" period="9" phase="1"
        processor="P2" schedulingModel="NP"
   oid="1088076" name="T1">
  <time>
     <computing value="1"/>
     <deadline value="9"/>
  </time>
  <precedes>
       <task-ref name="T2"/>
       <task-ref name="T5"/>
  </precedes>
  <excludes>
       <task-ref name="T3"/>
  </excludes>
  </task>

       ⋮

  <message bus="B1" oid="30377347" name="M1">
     <time>
        <communication value="17"/>
     </time>
     <precedes>
       <task-ref name="T4"/>
     </precedes>
  </message>
</realtime-table>
```

Figure 4: Specification in XML format

code segments. Tasks can be distributed in several processors in order to achieve the time restrictions.

3.1 A Method for Software Synthesis

This section describes a method for software synthesis considering embedded hard real-time applications. Our method comprises four main steps:

- *Specification:* describes the properties of each task in the system, including time restriction (phase time, release time, worst execution time, deadline, and period), scheduling method (preemptive and non-preemptive), arbitrary inter-task relations (precedence and exclusion), task-to-processor allocation, and task source code; tasks allocated in different processors communicate through a special task, called communication task; such a task is described by the worst communication time, communication channel, sender and receiver;
- *Modelling:* the specification is translated into a Time Petri Net (TPN) model [Merlin and Faber 1976]; each specification element is modeled through a

TPN building block; these blocks are composed to form the complete model [Tavares et al. 2005, Tavares 2006];

- *Pre-runtime Scheduler:* the next step searches for a feasible scheduling using the TPN model; the proposed scheduling algorithm performs a depth-first search on a finite timed Labeled Transition System (LTS) derived from a TPN model;
- *Software Synthesis:* the scheduled code is generated by traversing the timed LTS that represents a feasible schedule, if it exists, and substituting transition's instances by the respective code segments.

This work concentrates on the specification and software synthesis phases. More details about the modeling and pre-runtime scheduler phases may be found elsewhere [Tavares et al. 2005, Tavares 2006].

3.2 Specification of Hard Real-Time Systems

This subsection describes the specification elements included in CML for modelling hard real-time system.

A task is the basic element in the system. The specification is given in terms of temporal restrictions on task; the scheduling method adopted; inter-task relations; and inter-processor communications, which is modeled by a special task, called *communication task*.

Temporal Restrictions

In real-time systems, there are, generally, three types of tasks:

- *periodic tasks* perform a computation that are executed once in each fixed period of time;
- *aperiodic tasks* are activated randomly;
- *sporadic tasks* are executed randomly, but the minimum interval between two consecutive activations is known a priori.

Pre-runtime method performs scheduling decisions at compile time. It aims at generating a schedule table for a runtime component, namely, dispatcher, which is responsible for controlling the tasks during system execution. In order to adopt such method, the major characteristics of the tasks must be known in advance. This approach can only be used to schedule *periodic tasks*.

Definition 3.1 *(Periodic Task) Let T_i be a periodic task defined by T_i =(ph_i, r_i, c_i, d_i, p_i), where ph_i is the initial phase; r_i is the release time; c_i is the worst*

case computation time; d_i is the deadline; and p_i is the period. A periodic task samples objects of interest at a fixed rate. The phase (ph_i) is the delay associated to the first time request of task T_i after the system starting. $ph_i = 0$ whenever not specified. The release time (r_i) is the time interval between the beginning of a period and the earliest time to start the execution of task T_i. The computation time (c_i) is the worst case computation time required for executing task Ti. The deadline (d_i) is the time interval between the beginning of a period and the time instant at which task T_i must be completed (in each period). The period (p_i) is the time interval in which T_i must be executed.

The initial phase (ph_i) defines the point in time, after the system starts executing, when the task period can be allocated. The definition of ph_i is important, since non schedulable system may become schedulable when an initial phase is specified. For instance, considering two tasks, T_1 and T_2, having the same timing constraints (ph_1, r_1, c_1, d_1, p_1) = (ph_2, r_2, c_2, d_2, p_2) = (0, 0, 5, 5, 10). This system is not schedulable, since both takes 5 time units to execute and should finish at time unit 5. However, if an initial phase is specified, i.e. $ph_2 = 5$, the system becomes schedulable, because the period of T_2 is allowed to start 5 time units after the beginning of system execution. It is enough for task T_1 finishes. It is worth notice that the deadline is relative to the period and not to the entire system. Figure 5 presents a feasible schedule for the system.

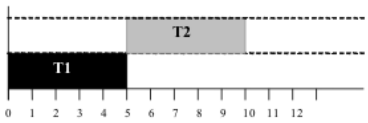

Figure 5: A feasible schedule for the system

Scheduling Method

The scheduling methods are all-preemptive and all-non-preemptive. In the all-preemptive scheduling method tasks are implicitly split into all possible subtasks. This scheduling method permits running other conflicting tasks, implying that one task could preempt another task. In turn, with the all-non-preemptive scheduling method processor is just released after finishing the entire computation.

Arbitrary Inter-Task Relations

A task T_i precedes task T_j, if T_j can only start executing after T_i has finished. In general, this kind of relation is suitable whenever a task (successor) needs

information that is produced by another task (predecessor). A task T_i excludes task T_j, if no execution of T_j can start while task T_i is executing. If it is considered a single processor, then task T_i could not be preempted by task T_j. Exclusion relations may prevent simultaneous access to shared resources. In this work it is considered that the exclusion relation is not symmetrical, that is, when A EXCLUDES B, not necessarily implies that B EXCLUDES A.

Inter-Processor Communication

When adopting a multiprocessing environment, all inter-processor communications have to be taken into account, since these communications affect the system predictability. An inter-processor communication is represented by a special task, namely, communication task, which is described as follows.

Definition 3.2 *(Communication Task) Let $\mu_m \in M$ be a communication task defined by $\mu = (T_i, T_j, ct_m, bus_m)$, where $T_i \in T$ is the sending task, $T_j \in T$ is the receiving task, ct_m is the worst case communication time, and $bus_m \in B$ is the bus, where B is the set of buses.*

It is worth observing that the bus is an abstraction for a communication channel used for providing communication between tasks from different processors.

3.3 Scheduled Code Generation

The proposed method for code generation includes not only tasks' code (implemented through C functions), but also a timer interrupt handler, and a small dispatcher. Such dispatcher automates several control mechanisms required during the execution of tasks. Timer programming, context saving, context restoring, and tasks' calling are examples of such additional controls. The timer interrupt handler always transfers the control to the dispatcher, which evaluates the need for performing either context saving or restoring, and calling a specific task.

An array of registers (`struct ScheduleItem`) is created to store the schedule table. Each input represents the *execution part* of a task instance. In case of preemption, a task instance may have more than one *execution part*. The register `struct ScheduleItem` contains the following information: (i) start time; (ii) flag, indicating if the task was preempted before; (iii) task id; and (iv) a pointer to a function (the respective task code). Figure 6 depicts the schedule table for a preemptive application. It includes two instances of `TaskA`, two instances of `TaskB`, two instances of `TaskC`, and one instance of `TaskD`. `TaskA1` and `TaskA2` are preempted in time 4 and 20, respectively. `TaskB1` is preempted twice: first in time 6 and, then, in time 10. Therefore, the schedule table contains 11 entries. Figure 7 presents the respective timing diagram.

```
struct ScheduleItem scheduleTable [SCHEDULE_SIZE] =
{{ 1, false, 1, (int *)TaskA},  TaskA1 starts
 { 4, false, 2, (int *)TaskB},    TaskB1 starts and preempts TaskA1
 { 6, false, 3, (int *)TaskC},    TaskC1 starts and preempts TaskB1
 { 8, true,  2, (int *)TaskB},    TaskB1 resumes executing
 {10, false, 4, (int *)TaskD},    TaskD1 starts and preempts TaskB1
 {11, true,  2, (int *)TaskB},    TaskB1 resumes executing
 {13, true,  1, (int *)TaskA},    TaskA1 resumes executing
 {18, false, 1, (int *)TaskA},    TaskA2 starts
 {20, false, 3, (int *)TaskC},    TaskC2 starts and preempts TaskA2
 {22, false, 2, (int *)TaskB},    TaskB2 starts
 {28, true,  1, (int *)TaskA}     TaskA2 resumes executing
};
```

Figure 6: Example of a Schedule Table

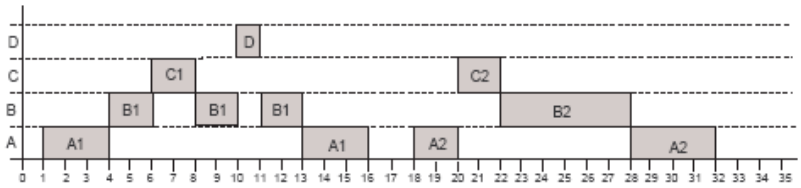

Figure 7: Timing Diagram for Schedule Table in Figure 6

A brief explanation of the dispatcher (Figure 8) is as follows: before calling a task, the dispatcher check some situations: (a) If the current task was preempted, the dispatcher saves its context (line 4); (b) If the next task has been preempted, and now it is being resumed, the dispatcher restores the context (line 5); and (c) If it is a new task instance, the dispatcher just stores the function pointer (line 7) in the variable `taskFunction` that will be called by the interrupt handler. Additionally, the table representing the feasible schedule is accessed as a circular list (line 9). The timer is automatically programmed using the start time of the next task instance to be called (line 10). After all these activities, the timer is activated to interrupt at the start time of the next task (line 11).

Whenever considering hard real-time embedded system design based on multiple processor platforms, the mechanism for processors synchronization is an important concern. A specific architecture was designed for this purpose. A *master processor* performs the time counting. It periodically sends synchronization messages to *slave processors*. There is no runtime scheduler running in each processor, but a runtime dispatcher. The scheduling is performed using a pre-runtime approach, as described before. Master processor is only responsible for performing the time counting and for notifying slave processors. It does not execute any task, but only a dispatcher. In addition, slave processors do not have their own real-time clocks. The real-time clock only resides in the master processor.

```
1 void dispatcher() {
2   struct ScheduleItem item = scheduleTable[scheduleIndex];
3   globalClock = item.clock;
4   if(currentTaskPreempted) { // context saving }
5   if(item.isPreemptionReturn) { // context restoring }
6   else {
7    taskFunction = item.functionPointer;
8   }
9   scheduleIndex = ((++scheduleIndex)%SCHEDULE_SIZE);
10  programTimer(scheduleTable[scheduleIndex].clock);
11  activateTimer();
12}
```

Figure 8: Simplified Version of the Dispatcher

In the proposed approach, all communication tasks are also taken into account in the code generation. Each communication task (μ_m) is translated into two special tasks: `sendMm` and `receiveMm`. Both tasks are executed at the same moment for guaranteeing the correct data transmission. `sendMm` and `receiveMm` are considered in the pre-runtime schedule table and both cannot be preempted. Section 4 presents an example of a system considering inter-processor communication.

4 A Case Study: Vehicle Monitoring System

This section presents a real application case study to illustrate the practical usage of CML for modeling a vehicle monitoring system. The example is particularly useful to demonstrate the application of CML for modeling systems executing in a multiple processing environment where inter-processor communication is required.

The system is composed of a set of sensors employed to verify whether the car components are working properly. If a component fails or works erroneously, the system notifies the driver through the dashboard. The vehicle monitoring system relies on multiple processors, since several sensors are considered and the microcontroller adopted (8051) contains only four 8-Bit I/O ports. In this way, two processors are used for interfacing with the sensors.

4.1 The specification model

A set of tasks were defined to check the status of the engine (TV and TR), breaks (TB), water (TW), gearing (TG), and temperature (TT). Finally, the data processed is sent to the task TRA, which is responsible for notifying the driver. Table 2 details the system specification, which is composed of 14 tasks, including the communication task M1. It implements the communication between

TaskID	Task Name	Release	Comp.	Deadline	Period	Proc. /Bus	From	To
TV0	ReadVelocity	0	231	20000	120000	P1	-	-
TV1	ProcVelocity	20000	5487	40000	120000	P1	-	-
TB0	ReadBreaks	20000	221	40000	120000	P1	-	-
TB1	ProcBreaks	40000	236	60000	120000	P1	-	-
TR0	ReadRPM	40000	232	60000	120000	P1	-	-
TR1	ProcRPM	60000	238	80000	120000	P1	-	-
TRA	Notifier	80000	2444	120000	120000	P1	-	-
TW0	ReadWater	0	237	20000	120000	P2	-	-
TW1	ProcWater	20000	241	40000	120000	P2	-	-
TT0	ReadTemperature	20000	259	40000	120000	P2	-	-
TT1	ProcTemperature	40000	234	60000	120000	P2	-	-
TG0	ReadGearing	40000	224	60000	120000	P2	-	-
TG1	ProcGearing	60000	236	80000	120000	P2	-	-
M1	-	-	1700	-	B1	TG1	TRA	-

Table 2: Task Timing Specification

processors P1 and P2. The implementation splits task TV into two subtasks: one (TV0) reads the sensor; and the other (TV1) processes the information. The same is done for tasks TR, TB, TW, TG and TT.

Figures 9 and 10 presents the CML specification for the vehicle monitoring system. The communication between processors P1 and P2 is implemented through the communication task M1. Two functions are generated to implement M1: `receiveM1`, in the receiving side; and `sendM1`, in the sending side. B1 is the bus used to transmit the message, which takes 1700 Time Task Units (TTUs) in the worst case. Task TG1 is responsible for sending the message from processor P2 to processor P1. The message sent to the processor P1 is received by the task TRA, which notifies the driver about the vehicle status.

The system was verified using the tool presented in Section 3. A feasible schedule was found after visiting 78 states. Figure 11 presents the timing diagram with the schedule for the vehicle monitoring system. The schedule was automatically generated by the tool.

More applications, such as Pulse Oximeter and Heated-Humidifier, have already been developed with CML support. These applications are described in [Barreto 2005].

5 Related Work

Meyer introduced the concept of *Design by Contract* [Meyer 1992, Meyer 1997], a lightweight formal method that allows for dynamic runtime checks of specification violation. Design by Contract establishes that a relationship between a class and its clients is viewed as a formal agreement, which expresses each party's right and obligations. A precondition states the properties that must hold when a routine is called; the postcondition states the properties that the

/*! * @task TV0 * @processor P1 * @scheduling NP * @phase 0 * @release 0 * @wcet 231 * @deadline 20000 * @period 120000 * @precedes {TV1} */ void TV0() {...}	/*! * @task TV1 * @processor P1 * @scheduling NP * @phase 0 * @release 20000 * @wcet 5487 * @deadline 40000 * @period 120000 */ void TV1() {...}	/*! * @task TB0 * @processor P1 * @scheduling NP * @phase 0 * @release 20000 * @wcet 221 * @deadline 40000 * @period 120000 * @precedes {TB1} */ void TB0() {...}	/*! * @task TB1 * @processor P1 * @scheduling NP * @phase 0 * @release 40000 * @wcet 236 * @deadline 60000 * @period 120000 */ void TB1(){...}
/*! * @task TR0 * @processor P1 * @scheduling NP * @phase 0 * @release 40000 * @wcet 232 * @deadline 60000 * @period 120000 * @precedes {TR1} */ void TR0(){...}	/*! * @task TR1 * @processor P1 * @scheduling NP * @phase 0 * @release 60000 * @wcet 238 * @deadline 80000 * @period 120000 */ void TR1(){...}	/*! * @task TRA * @processor P1 * @scheduling NP * @phase 0 * @release 80000 * @wcet 2444 * @deadline 120000 * @period 120000 */ void TRA(){...}	void receiveM1(){...}

Figure 9: CML specification for tasks in processor P1

/*! * @task TW0 * @processor P2 * @scheduling NP * @phase 0 * @release 0 * @wcet 227 * @deadline 20000 * @period 120000 * @precedes {TW1} */ void TW0() {...}	/*! * @task TW1 * @processor P2 * @scheduling NP * @phase 0 * @release 20000 * @wcet 241 * @deadline 40000 * @period 120000 */ void TW1(){...}	/*! * @task TT0 * @processor P2 * @scheduling NP * @phase 0 * @release 20000 * @wcet 259 * @deadline 40000 * @period 120000 * @precedes {TT1} */ void TT0(){...}	/*! * @task TT1 * @processor P2 * @scheduling NP * @phase 0 * @release 40000 * @wcet 234 * @deadline 60000 * @period 120000 */ void TT1(){...}
/*! * @task TG0 * @processor P2 * @scheduling NP * @phase 0 * @release 40000 * @wcet 224 * @deadline 60000 * @period 120000 * @precedes {TG1} */ void TG0(){...}	/*! * @task TG1 * @processor P2 * @scheduling NP * @phase 0 * @release 60000 * @wcet 236 * @deadline 80000 * @period 120000 * @sends M1 B1 1700 TRA */ void TG1(){...}	void sendM1(){...}	—

Figure 10: CML specification for tasks in processor P2

routine guarantees when it returns. As a consequence, pre and postconditions enforce behavior with contracts, which is also expressed by the term *software contract*.

JML [Burdy et al. 2005, Leavens et al. 2006] is a notation used to formally specify the behaviour and interfaces of classes and methods written in Java [K. Arnold and J. Gosling 1996], which follows the "software contract" concept introduced in the Eiffel language. By using JML, one can specify both the in-

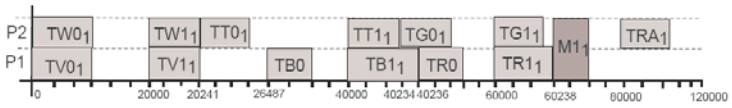

Figure 11: The schedule for the vehicle monitoring system

terface of methods and classes as well as their behavior. Interface specification usually includes type information and visibility modifiers, for instance. We can specify the interface of methods (including name, visibility, number of arguments, return type, and so on), attributes (name, type and modifiers), and types (name, modifiers, whether it is a class or interface, its supertypes and so on). In fact, JML uses Java syntax to specify all such interface specification.

The behavior of a method or type specifies the transformations that are performed by them. The style of specification of JML is usually called model-oriented [Wing 1990]. Indeed, specifications written in JML follow the style of refinement calculus [Morgan 1994]. The attributes to which we can assign in a method are described by the method's *frame-axiom*. States to which methods are defined are formally described by means of assertions (the method's *pre-condition*); states that may result from the normal execution of methods are described by logical assertions, are called the method's *normal postcondition*. The relationship between the state in which a method is called and the states that may result from throwing an exception are described by the method's *exceptional postcondition*. We can also specify class invariants in JML, describing properties that hold in all visible states [Leavens et al. 2006]. We can also deal with refinement in JML.

A distinguishing feature of JML is the range of tools available. We can check the correctness of JML specifications using the runtime assertion checker *jmlc*, the JML compiler. Unit test is partially automated by the *jmlunit* tool that generates test classes that rely on the JML runtime assertion checker. The *jml-doc* tool produces HTML browsable pages in the style of pages generated by *javadoc*. Other tools with distinct purposes are also available: *ESC/Java* (static checker) [Flanagan et al. 2002], *LOOP* tool [Jacobs et al. 1998] (compilation to PVS [Owre et al. 1992] theories), and *JACK* (program checker) [Burdy et al.].

Although most of the JML annotations are directed to functional requirements, there are some annotations, which are related with expressions that can be used to express non-functional requirements. For instance, a duration expression describes the specified maximum number of virtual machine cycle times needed to execute a method call. Also, a space expression describes the amount of heap space allocated to an object given as argument. The CML handles this kind of assertions in a higher-level way, dealing not simply with time associated

to machine cycles, but to the execution of tasks and scheduling. Besides that, we deal with memory use, and energy consumption.

The *Spec#* programming system [Barnett et al. 2004] provides a complete infrastructure, including libraries, tools, design support, and integrated editing capabilities for C# programs specification. Part of this system is composed by the Spec# programming language, which is a subset of the existing object-oriented .NET programming language C#. Like JML, the Spec# language focuses on specification of functional aspects. On the other hand, CML deals with specification of non-functional aspects of programs written in the C language.

The tool *Jass* [Bartetzko et al. 2001] translates Java annotated programs into pure Java programs in which compliance with the specification is dynamically tested during runtime. Assertions are written as comments into Java code; they are simply boolean expressions. Different kinds of assertions are allowed: method pre and postconditions; class invariants; loop invariants and variants; refinement checks; and trace assertions that specify the intended dynamic behavior of objects in time, describing allowed traces of events. In CML, we can also express the order in which tasks must be executed, but we go further since we can also express requirements on time, power and memory, for instance. On the other hand, in CML we cannot check tasks refinements.

The Bandera Specification Language [Corbett et al. 2000] is a source-level, model-checker independent language for expressing temporal properties of Java programs action and data. An assertion sublanguage allows programmers to define constraints on programs by writing pre- and postconditions. A temporal property sublanguage provides support for defining predicates on control points (method call and return) and data (object instantiation) present in Java programs. This sublanguage allows specifying temporal properties between system actions. The language provides *specification patterns* that describe properties like precedence. The patterns mentioned in [Corbett et al. 2000] can also be described in CML. However, we do not deal with pre and postconditions as in the assertion language of Bandera.

6 Conclusions

Non-functional requirements (NFR) has long been recognized as a fundamental issue in software development. The popularity of small devices and embedded systems increases the importance of NFR. Due to the limited amount of resources, the correctness and quality of software targeting these platforms relies equally on functional and non-functional aspects. This work presented CML (C Modeling Language), a specification language to describe non-functional requirements of C programs. We decided to focus on the C language because it is widely employed to develop applications with severe non-functional restrictions, such as performance, memory allocation, and energy consumption.

The development of CML was inspired in the Design By Contract (DBC) technique. Thus, specifications are added as annotation comments to the C program, which hence can be compiled with any C compiler. We believe that this tight integration between specification and implementation languages contributes for motivating programmers to use the specification language in practice. It is worth notice that, CML does not follow the DBC technique. For instance, the concepts of preconditions, postconditions, and invariants have no interpretation in CML. On the other hand, a CML specification establishes a kind of contract, in the sense that: from the NFR perspective, if the specification was respected, the system will work properly. We believe that CML can contribute to improve the quality of software systems. In particular, those with stringent resource limitations and performance requirements.

In addition to the proposition of CML itself, this paper contributed with the first verification tool based on CML The tool receives a CML specification for embedded hard real-time systems and checks if exists a feasible schedule for the system. If the answer was yes, the schedule is automatically generated and the code for the system is synthesized considering a pre-runtime scheduling strategy.

In order to show the practical feasibility of the proposed software synthesis method, we specified a vehicle monitoring system. This is a multiple processing application, which was useful to demonstrate the modeling of inter-processor communication in CML. The system was then analyzed through the tool and the scale was found after visiting 78 states.

6.1 Future Works

The current CML implementation does not cover the whole language constructors. Through an incremental approach, we first implemented the elements that address hard-real time systems. The implementation was then validated using a real application. Other verification tools considering the remaining set of CML constructors (see Appendix A) will be released in near future.

We are currently working to implement a verification tool considering two constructors (`@pesaccess` and `@optaccess`) related to concurrent access policies, in order to guarantee safe execution, and therefore, data consistency. The concurrent access policies are responsible to control a function execution in order to avoid undesirable interferences in a concurrent environment. There are two possible access policies: pessimistic (`@pesaccess`) and optimistic (`@optaccess`). In the pessimistic access policy the function access is made through a critical section using a default or a used-defined lock variable. This technique guarantees that while a thread/task is executing a function with this annotation no other thread/task can execute the same function. This is basically the synchronization of all functions execution. Since this synchronization might be too restrictive and decrease performance, an alternative access policy (`@optaccess`) considers

the function semantics in order to decide when block and when allow concurrent access to a function. This optimistic policy considers that in some cases it is possible to define when two or more threads/tasks executions might conflict to each other [Soares and Borba 2001]. For instance, a function might not be concurrently executed if both executions are using the same arguments values.

We are constantly reviewing the language to include/remove constructors and constructors' parameters. New versions of CML will be released soon.

The tool presented in this work as well as more information about CML can be found at `www.dsc.upe.br/cml`.

References

[Barnett et al. 2004] Barnett, M. et al. (2004). The Spec# Programming System: An Overview. In *CASSIS'2004*, pages 49–69, New York, NY, USA. ACM Press.

[Barreto 2005] Barreto, R. (2005). *A Time Petri Net-Based Methodology for Embedded Hard Real-Time Systems Software Synthesis*. PhD thesis, Informatic Center - Federal University of Pernambuco.

[Bartetzko et al. 2001] Bartetzko, D., Fischer, C., Möller, M., , and Wehrheim, H. (2001). Jass — Java with Assertions. *Eletronic Notes in Theoretical Computer Science*, 55(2):103–117. RV'2001 - Workshop on Runtime Verification, (in connection with CAV '01).

[Burdy et al. 2005] Burdy, L. et al. (2005). An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232.

[Burdy et al.] Burdy, L., Lanet, J.-L., , and Requet, A. JACK (Java Applet Correctness Kit). Technical report. Last visit: 31 March 2007.

[Cheon and Leavens 2002] Cheon, Y. and Leavens, G. (2002). A runtime assertion checker for the Java Modeling Language. Software Engineering Research and Practice (SERP'02), pages 322328. CSREA Press. citeseer.ist.psu.edu/cheon02runtime.html.

[Corbett et al. 2000] Corbett, J. C., Dwyer, M. B., Hatcliff, J., and Robby (2000). A Language Framework for Expressing Checkable Properties of Dynamic Software. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 205–223, London, UK. Springer-Verlag.

[Flanagan et al. 2002] Flanagan et al. (2002). Extended static checking for Java. *SIGPLAN Not.*, 37(5):234–245.

[Gannon et al. 1994] Gannon, J. D., Purtilo, J. M., and Zelkowitz, M. V. (1994). *Software Specification: A Comparison of Formal Methods*. Ablex Publishing Co., 355 Chestnut Street, Norwood, NJ 07648.

[Godefroid 1996] Godefroid, P. (1996). *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA.

[Hoare 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

[Jacobs et al. 1998] Jacobs, B. et al. (1998). Reasoning about Java classes. In *OOPSLA'98*, pages 328–340.

[K. Arnold and J. Gosling 1996] K. Arnold and J. Gosling (1996). *The Java Programming Language*. Addison-Wesley.

[Leavens et al. 2006] Leavens, G. et al. (2006). *JML Reference Manual*.

[Merlin and Faber 1976] Merlin, P. and Faber, D. J. (1976). Recoverability of communication protocols: Implications of a theoretical study. *IEEE Transactions on Communications*, 24(9):1036–1043.

[Meyer 1992] Meyer, B. (1992). Applying "Design by Contract". *Computer*, 25(10):40–51.

[Meyer 1997] Meyer, B. (1997). *Object-Oriented Software Construction*. Prentide-Hall, second edition.

[Morgan 1994] Morgan, C. C. (1994). *Programming from Specifications*. Prentice Hall, second edition.

[Owre et al. 1992] Owre, S., Rushby, J. M., and Shankar, N. (1992). PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607, pages 748–752, Saratoga, NY.

[Soares and Borba 2001] Soares, S. and Borba, P. (2001). Concurrency Manager. In *First Latin American Conference on Pattern Languages of Programming — SugarLoafPLoP*, pages 221–231, Rio de Janeiro, Brazil. Published in UERJ Magazine: Special Issue on Software Patterns.

[Su and Hsiung 2002] Su, F.-S. and Hsiung, P.-A. (2002). Extended quase-static scheduling for formal syn- thesis and code generation of embedded software. In *International Symposium on Hardware/Software Codesign (CODES'02)*.

[Tavares 2006] Tavares, E. (2006). A Time Petri Net Based Approach for Software Synthesis in Hard Real-Time Embedded Systems with Multiple Processors. Informatic Center - Federal University of Pernambuco.

[Tavares et al. 2005] Tavares, E. et al. (2005). A time petri net based approach for embedded hard real-time software synthesis with multiple operational modes. In *SBCCI '05: Proceedings of the 18th annual symposium on Integrated circuits and system design*, pages 98–103, New York, NY, USA. ACM Press.

[Wing 1990] Wing, J. M. (1990). A Specifier's Introduction to Formal Methods. *Computer*, 23(9):8–24.

[Xu and Parnas 1993] Xu, J. and Parnas, D. L. (1993). On Satisfying Timing Constraints in Hard-Real-Time Systems. *IEEE Trans. Softw. Eng.*, 19(1):70–84.

A The Complete Set of CML Constructors

Constructor	**Description**	**Format**	**Program/ Function †**	**Requirement**
@task	task name	String	F	–
@processor	processor to task allocation	String	P and F	resource allocation
@scheduling	task scheduling strategy	NP (Non-preemptive) or P (Preemptive)	F	scheduling
@phase	task phase time	Integer	F	temporal restrictions
@release	task release time	Integer	F	temporal restrictions
@wcet	worst case execution time	Integer	P and F	temporal restrictions
@deadline	task deadline time	Integer	F	temporal restrictions
@period	task period time	Integer	F	temporal restrictions
@precedes	tasks preceded by this task	List of tasks between the tokens { and } separated by comma	F	inter-task relation
@excludes	tasks excluded by this task	List of tasks between the tokens { and } separated by comma	F	inter-task relation
@sends	message sent by this task	This attribute is followed by: 1-message name, 2-bus name, 3-worst case communication time, 4-receiver task	F	inter-task communication
@usedmemory ⋆	maximum amount of memory to use	Integer (Kb)	P and F	memory use
@codesize ⋆	generated binary	Integer (Kb)	P and F	memory use
@cachehit ⋆	maximum number of accesses to cache (L1 and L2 levels)	L1:Integer, L2:Integer	P and F	performance
@cachemiss ⋆	maximum number of cache misses (L1 and L2 levels)	L1:Integer, L2:Integer	P and F	performance
@pagefault ⋆	maximum number of page faults	Integer	P and F	performance
@power ⋆	maximum energy consumption	Real (Joules)	P and F	Power
@pesaccess ⋆	no concurrent access is allowed; optionally, specify a lock variable	String (var)	F	Safety/data consistency
@optaccess ⋆	some concurrent access is allowed; specify a variable or parameter list as the function semantics	List of variables between the tokens { and } separated by comma	F	Safety/data consistency

† *Constructor granularity: P→ whole program; F→ single function.*

⋆ *CML constructors not considered in the current implementation*