A Language for Complex Real-Time Systems

ALEXANDER D. STOYENKO, THOMAS J. MARLOWE AND MOHAMED F. YOUNIS

Real-Time Computing Laboratory, Department of Computer and Information Science, New Jersey Institute of Technology, Newark, NJ 07102 USA
Email: alex@rtlab12.njit.edu

The new generation of real-time systems are characterized by multiple, conflicting non-functional desiderata on goals. Furthermore, the systems exhibit very large size and complexity—in both application structures and underlying software and hardware platforms. We argue that current high-level real-time languages do not meet the challenge of these complex real-time systems and introduce a new language—CRL—that we claim does. Relevant real-time features of CRL are discussed and a summary is provided *vis-à-vis* future features that would address non-functional goals other than timeliness. A current implementation status and how CRL fits into a rather ambitious environment for the construction of complex real-time systems (under construction in our Real-Time Computing Lab at NJIT) are briefly presented.

1. INTRODUCTION

In the past several years, there has been a modest yet visible trend toward addressing design and development of computer systems in which construction and execution is affected by multiple, simultaneous and interacting constraints and goals. We refer to these computer systems as complex—large applications, typically running on distributed, heterogeneous networks, driven by a number of distinct constraints and desiderata on goals such as performance, real-time behaviour and fault tolerance. These requirements frequently conflict; further, satisfaction of these design objectives interacts strongly with every stage of development, system engineering, construction and operation, including design/selection of hardware, environment and application software. The sheer difficulty of engineering of complex computer systems necessitates definition and creation of a tool suite remarkable for the number, nature and power of services provided.

In this paper, we would like to present one tool in this suite, namely, a new high-level programming language, Language for Complex Real-Time Systems, or CRL. While it is our intention to incorporate into CRL mechanisms to address a number of goals (see a brief discussion in Section 5), we focus here on features that make CRL suitable for developing large real-time applications.

The rest of the paper is organized as follows. Section 2 briefly discusses the existing language scene and motivates the creation of CRL. In Section 3 an overview of CRL is presented. Section 4 discusses key real-time details of CRL's design. In Section 5 we briefly summarize how we intend to provide CRL mechanisms to support other goals. Section 6 presents the CRL implementation, and indicates how the language fits into the tool suite for engineering of complex computer systems under construction at NJIT's Real-Time Computing Lab. In Section 7 we briefly conclude and outline future work. Finally, Appendices A and B contain, respectively, CRL examples and significant CRL BNF elements not fully defined or explained in the main text.

2. MOTIVATION

2.1. What is a real-time language?

An environment, such as a nuclear reactor, a manufacturing floor, or a ship, is real time when its correct operation includes adherence to timing constraints. For instance, when two chemical agents are mixed to form a medicine, the basic laws of chemistry may require that the agents be mixed for X seconds at a temperature of Y to achieve the correct result. Similarly, the rate of an assembly line, which is determined by the conveyor operator subject to the limits of the conveyor system, forces a step in the assembly to be undertaken in Z seconds or the parts involved will be lost or damaged. Real-time applications define a paradigm of computing very different from that of traditional computing. While in a traditional paradigm (such as interactive processing, for instance) the correctness of a program is independent of the timing characteristics of its execution, a real-time program must be its very nature adhere not only to functional semantic requirements, but also to the timing constraints of its application environment.

The motivation for developing real-time programming languages is to facilitate writing of correct and maintainable real-time programs, through more effective use of abstraction, compilation, and a priori analysis. Hundreds of languages have been designed to address not only such main traditional paradigms of computing as batch or interactive programming, but even specific areas within these paradigms. Thus, FORTRAN was designed specifically for scientific computing, COBOL for business applications and so forth. It is reasonable that languages should have been designed for real-time computing. While several languages have been designed or designated to be used in real-time computing, until recently they have lacked, to a significant extent or entirely, the notion of real-time as a first-class entity.

Schedulability analysis—a key requirement for a high-level real-time language—was originally defined by [1–5]. Schedulability analysis refers to any pre-execution or symbolic language-level analysis of programs that either determines whether the programs will meet their critical timing constraints when executed, or derives the programs' timing characteristics. While the concepts and details of how a real-time program or a set of extracted real-time tasks is/are analysed for schedulability are beyond the scope of this paper (but can be found in a number of sources [1, 3–23]), it is important to realize that schedulability analysis cannot in general be applied to programs written in a conventional language, even in a concurrent programming language such as Ada, since such languages do not provide syntactic/semantic mechanisms to define real-time processes and have constructs which take arbitrarily (and unpredictably) long to execute. Thus, conventional languages need to be either modified or extended with additional higher-level directives or "pragmas" to facilitate their use in real-time computing.

The very nature of schedulability analysis requires predictable system software and hardware behaviour. Ideally, the time taken for execution of each machine instruction should be known, and the hardware should not introduce unpredictable delays into program execution. In practical programs, however, it is typically the case that the execution times useful for schedulability analysis are for blocks of instructions long enough that small variations in execution times of individual instructions tend to average out within a block [24]. While we do not address this subject in this paper, not only can realistic systems closely fitting these assumptions be assembled from existing components, but a number of sources indicate that entire such system software and hardware systems used in time-critical real-time applications can be, should, and are being designed in this way [25–33]. A complete predictable computer architecture can be found in [28]. Among the features facilitating predictability are "regular" DMA access (implemented as for memory refresh), managing secondary storage operations predictably through swapping by unit of activity or preparing scheduling blocks (which is *de facto* state of the art in many existing time-critical applications, see for instance [34]); and providing hardware support for exactly timed input and output.

2.2. The real-time language scene

Historically, the real-time language scene has been anything but consistent. The need for real-time languages emerged in the earliest days of computing, and arguably the first high-level programming language, Plankalkül, was meant for real-time applications [35]. For those who endured through the Dark Ages of pseudo-code, assembly, and early computers in the 1940s and the 1950s, the 1960s and the 1970s brought a Renaissance with FORTRAN, ALGOL, PL/I, and others (see for example [36–38]), all incorporating novel mechanisms—at the language and system level—suitable for real-time computation. The 1970s were then perhaps a good time for various standardization and government bodies to step in, and help. They did, standardizing and helping in a number of cases: CCITT, PEARL and others.

Then came Ada—a "real-time" language with essentially no notion of time and few constructs suitable for real-time computation. Given the nature of R&D funding in the USA and other Western countries, the sheer size of this government-mandated effort had significant consequences. Experimental real-time language research which depended on such funding had to be either a spin-off of non-language research, or an "attachment" to an Ada program. By the mid- to late-1980s, it was hard to find any trace of any previous or new non-Ada real-time language work, save in a few exceptions at universities and research laboratories. The first language design which forced every program to express its timing constraints, defined a real-time process model, and which forbade or restricted all constructs that might take arbitrarily long to execute was Real-Time Euclid [1, 4]. Real-Time Euclid also included the first complete schedulability analyser [3, 5], and was the first language which guaranteed that every program was schedulability analysable. A single prototype of the language and its schedulability analyser was implemented and evaluated. Among other notable examples are Tomal [39], DICON [40], Flex [41], RTC++ [42], and High-Integrity PEARL [43]. Not surprisingly, a large community continued to work on Ada, or, more properly, on ways of using Ada in real-time applications (not an easy task, typically through run-time and pragma "tricks") and on ways of extending the Ada 1983 standard 1 (to the Ada '95 standard 2). Among those efforts, Baker's [6, 44, 45] and Sha and Goodenough's [17] notable.

Seemingly, the real-time language scene of the 1990s is not substantially different from that of the 1980s. There is not much real-time language design work that we can identify, though there is actually some solid work on supporting real-time program development, related to compilers, debuggers and other tools [12–15, 21–23, 46–60].

2.3. Why another real-time language and why CRL?

In our opinion, there are some major weak points in proposed real-time languages which affect their usability and suitability for complex real-time systems. Such weaknesses come from two sources: isolation from trends in computer programming languages, and lack of support for non-functional features of complex real-time systems.

Current standard languages also do not readily support real-time programming. in today's software market place, the fastest growing language—C++—has essentially no real-time features. Given this fact, as well as the emergence of the POSIX and its real-time extensions (i.e. a real-time Unix, as for example in [76]) standard, there is no doubt that C++ will also become a widely used language in the real-time arena. Despite the decline of military spending, Ada is still a very formidable player. It is also the only language in existence which is massively funded by the government (or, more precisely, by governments). While the '95 version does make significant strives towards acceptable real-time functionality [61], these are hardly sufficient. In fact, arguably, ten years after Real-Time Euclid, Ada still accepts but a subset of that language's constructs, and suffers from a number of poorly designed original concepts (such as the semantics of the `TASK EXCEPTION`). Moreover, the relegation of many practical real-time features (or common packages and so forth) to the optional and "application-domain-specific" Real-Time Annex 3 means that Ada's famed portability and standardization of features will suffer. Significant European "players", such as PEARL [8, 43, 62] do not appear to be gaining in today's market.

On the other hand, while Real-Time Euclid was an advance for its time, changes in both the development environment and the nature of applications argue that a real-time language must now provide a number of features enabling secure, predictable, concurrent, maintainable, etc., real-time operation, and support a number of development objectives as well. These latter objectives are summarized as follows:

- Aiming for programming in the large in the modern world, the language should support a real-time variant of object orientation and other features.
- The language needs a more robust programming model than that allowed by restricted real-time languages. For instance, general loops and recursion may be allowed and then eliminated or restricted by the compiler (to ensure no construct executes arbitralily long).
- Timing constraints should be considerably richer than in older languages, allowing for much finer granularity of expressions.
- User assertions should be supported, both to provide hints to the schedulability analyser, and to allow more programs to execute, at the cost of run-time checks where the assertion cannot be validated by the compiler, or by the partial evaluator using system or link-time information.
- The language should compile to portable, standardized form.
- The language should fit into an integrated environment—a suite of tools (including non-functional objective analysis tools, assignment, allocation and migration tools, debuggers and monitors and so on).
- A good set of exception handling mechanisms is needed.
- The design should address not only real-time performance, but also other non-functional objectives such as fault tolerance, security, and quality of service.
- A good set of communication and synchronization mechanisms is needed.
- Finally, a language definitely needs to capture the features of existing real-time languages.

While the combination of features in CRL is unique, many of its features are supported in other language and compiler efforts, some of which are mentioned above. Various other approaches allow the use of user assertions, partial evaluation [53], and many of the other features of CRL. One of the most significant features of this paper, finer-grained timing constraints, is not a complete novelty. Several languages, including TCEL [63] allow specification of timing constraints between events or sets of events in a process. The work of Dietz and Chung [64] provides an embedding language in the CHaRTS compiler whose time-constraint specification facility is related to our own; their constraint syntax is also quite clean.

3. CRL OVERVIEW

CRL is sufficiently robust and expressive to capture most standard functionality, but also sufficiently structured in both native constructs and annotations to afford static analysis by reasonable techniques. The language serves as a vehicle for research and experimentation in both complex real-time languages and in schedulability analysis and assignment, and techniques for enabling efficient analysis and assignment of components of complex real-time programs.

CRL addresses several perceived problems in most existing real-time languages: (i) it has a well-defined and robust semantics for functional, real-time, and other non-functional behaviour; (ii) CRL allows but does not require fine-grained constraints relating any two given statements, with natural interpretations on constraints in conditionals and loops (however, the syntactic relationship of the two statements has to be constrained to allow efficient analysis); (iii) CRL allows, via annotations, static analysis and partial evaluation (as in Maruti [53]), the use of a full range of structured programming constructs, including general loops and recursion, while remaining largely schedulability analysable (and otherwise guarded by exceptions); (iv) CRL has among its basic constructs explicit parallelism (via threads) and object orientation, including natural inclusion of resources, and an exception mechanism.

CRL includes a language for constraints and assertions, on timing, recursion, iteration, data set size, constant values, etc. These are in general intended to support partial evaluation (as above) and static analysis. It is required that if subsequent code requires the truth of an assertion for correctness, that the assertion is marked as "invariant".

CRL serves as a foundation to our testbed for transformations and other investigations, as a language specifically for time-constrained complex computer systems.

3.1. Classes, concurrency, typing, binding

CRL exhibits a number of features aimed at facilitating the development of robust and reliable complex real-time programs. The usage of the class mechanism is restricted to disallow recursive definitions, and to encourage successive refinements of generics in place of fully dynamic binding. Formal parameters (in headers and class constructors) are declared as types (including classes) only. To allow call-time (semi-dynamic) binding of actuals, the language permits late instantiation of parameters (but their types must of course correspond exactly to or inherit solely from their corresponding formal parameters). Incorporating inheritance will be a future extension.

In CRL, concurrency is provided through threads within objects. For uniformity and orthogonality, threads are actually defined as special methods.

3.2. Timing assertions, observably timed statements, constraints

The set of program statements whose execution time may occur in timing constraints is specified syntactically. All critical section accesses, accesses to I/O, and synchronizations/messages between processes are, by default, observably timable, as are the beginning and the end of a process. The fork and join nodes governing one or more conditionally executable timable nodes are themselves observably timable (this is a mild exception to the syntactic nature of timability). Other statements may be labelled as timable. In theory, one of the statements above could be labelled as non-timable; we would not expect this to be common, except perhaps for code added to a process for logging, profiling, debugging or similar purposes.

The execution time of a timable statements can be distinguished by labels. A timable statement has the form

```
<timed_statement>::=[$<label.!]
                     <untimed_statement>[!<label>$]
```

where the first label represents the execution initiation time for `untimed_statement`, and the second label represents the execution completion time.

Constraints are either absolute, relating a statement in a thread or object to the beginning or end of the current frame, or relative, constraining the time interval between two observably timable and statically co-executable statements, each in or called by the same process, or involved in a single inter-process synchronization. Constraints are either max_time or min_time constraints, according as they place an upper or a lower bound on absolute or relative times; more complicated constraints are syntactic sugar for combinations of these types.

CRL uses timing assertions both as a programming aid, and as a facilitator/focuser for the transformer, conditional linker, schedulability analyser, assignment tool, and other tools, as briefly discussed in Section 6. In the spirit of orthogonality, these assertions are allowed in association with any statement or pair of statements or even expressions. (Since, however, a system of absolutely general constraints can cause problems for analyses—in fact, can even encode the halting problem—the compiler will check a global legality property relating the form and location of associated statements, and the form of the associated constraints). Meaningful expression forms include relative and absolute timing expressions, based on intervals. Eventually, time variables will be allowed.

CRL promotes reuse through object-based components. To assist in component storage, selection and other operations (and thus support libraries of reusable components), object-level timing properties are asserted in the same language as regular CRL timing assertions.

3.3. A foundation of "conventional" mechanisms and features

A complex-system higher-order language, such as CRL, will naturally include conventional language constructs and mechanisms, most typically based on an existing language paradigm, with support as above for functional and non-functional systems engineering objectives. CRL uses for its foundation a largely object-oriented paradigm with an imperative expression syntax for scalar types (much as C++).

However, CRL is not simply an object-oriented language with support for real-time and other systems engineering objectives added *a posteriori.* Rather, the syntax and semantics for functionality and for systems engineering objectives have necessarily co-evolved. The language and the constraint paradigms, and the basics of their syntax, were specified together; as we investigated support for non-functional systems objectives, we found it necessary to disallow or restrict certain standard mechanisms or features, such as arbitrarily long computation primitives (because of real-time requirements) or mechanisms that lead to poor or unsafe execution semantics (such as the ambiguity in the Ada's `TASK EXCEPTION`). Some of these decisions modified the existing language; others have guided the direction of later extensions.

There are several possible strategies in limiting a troublesome feature in a language's syntax to support system objectives: (i) it can be totally disallowed, as in Real-Time-Euclid; (ii) it can be made safe (for example, by adding timeouts, or by refining/redefining the semantics); or (iii) it can be permitted, provided the compiler, or another pre-run-time tool, can eliminate the problems. In CRL, we rely principally on the last of these, but any reasonable approach will probably have to use a combination of the three approaches.

4. CRL DETAILS

The language of real-time systems consists of a set of top-level objects (some with threads of control), possibly running on a distributed network, accessing a set of resources managed as other objects, and synchronizing via calls and messages (messages are not yet implemented).

Fundamentally, we need both a high-level general-purpose language, and a powerful expression mechanism for timing and other constraints. The language provides the standard core of a high-level language, including array and record-type constructors (but not at present pointers), and function calls with limited recursion. The start and end of any atomic statement can be used as temporal reference points, allowing an extremely powerful language for temporal constraints; however, the semantics of the language restricts the pairs of co-referenced statements, and the meaning of certain constraints, to permit efficient and precise analysis.

For real-time programs, time-unbounded higher-level constructs such as `while` loops will have to be transformed—either automatically or with the aid of the programmer—into time-bounded constructs using the full power of such constructs requires a more powerful constraint/assertion language, including constraints and/or assertions on iteration counts and recursion depth.

The syntax uses single-line prefixed comments (rather than delimited comments); the comment character is %. Any input between a comment character and the next end of line is ignored by the compiler.

4.1. Programs, declarations, classes, objects, visibility

A program is a collection of type declarations, constant declarations and variable declarations. Type declarations include classes, and variable declarations include class objects. Currently, all declarations are static, to facilitate predictability and schedulability analysis. Declaration before use is required; scoping is static. A declaration persists exactly for the lifetime of its declaring scope; thus, program declarations and declarations in scopes that do not depend on subprograms or blocks live for the entire program. A declaration in a scope which does depend on a subprogram or block lives for the lifetime of the corresponding innermost subprogram or block encompassing (perhaps transitively) the scope of the declaration.

Manifest integer expressions are used for initialization in arrays and ranges. This aids both the compiler and the programmer by forbidding type constructors in variable declarations (only scalar and used-defined type name are allowed there). Upon object creation, only IN parameters may be passed to constructors of objects. Object creation is currently allowed only through static declarations. Likewise, only IN parameters are currently allowed when calling a thread (this corresponds to thread activation—see a more detailed discussion below).

A class consists of an interface, together with internal and exported methods and threads. Methods and threads have parameter lists, where each parameter is identified as IN, OUT, or INOUT. Parameters are passed as in Ada: IN parameters by value, OUT parameters by result, and INOUT parameters by value-result. It is currently illegal for two parameters to be aliased, unless at least one is an IN parameter; further, by construction, there are no global variables. Thus, as in Ada, pass by reference can be used for OUT or INOUT parameters for local procedure calls, and value-result for remote procedure calls, without affecting the semantics.

Only types and methods defined and exported explicitly from one class, and imported explicitly into another, may be visible in the latter class. Every method (including threads) declared in the class (or in an object of the class) implicitly imports all imports of the class (object). This import rule permits the method to declare variables and parameters to be of imported types (classes), and to consequently use these variables and parameters. Note that this rule permits run-time binding of actual parameters (to formals), while neither completely enabling nor forbidding dynamic binding of parameter types. The same implicit import rule does not apply to classes declared (directly or indirectly) inside other classes. Rather, as already stated, all imports into classes must be explicit; moreover, direct or indirect recursive imports are forbidden. Blocks in the bodies of either methods or threads implicitly import everything imported by their owner methods or threads.

A constructor or a destructor are special methods which may not be exported or imported, and are invoked implicitly only, during object declaration elaboration and de-elaboration respectively. When an object is declared, its constructor is passed an actual parameter for every formal parameter in the constructor method interface. Only IN parameters may be passed to constructors; no parameters are passed to or from destructors. Since constructors and destructors are invoked implicitly, they use no names. We are still debating the utility of labels and time constraints in constructors and destructors, but the following is the current definition:

```
<constructor>::=[$<label>!] constructor
                [<time_constraint>] <in_param_list>
                <decls> [<statements>]
                endconstructor [!<label>$]
<destructor> ::=[$<label>!] destructor
                [<time_constraint>] <decls>
                [<statements>] enddestructor
                [!<label>$]
```

A method is either a conventional ("regular") method or a thread. Not surprisingly, both possibilities allow for time constraints and time expressions:

```
<method>         ::=<regular_method>|<thread>
<regular_method>::=[$<label>!] method <name>
                   [<time_constraint>]
                   <param_list> <decls>
                   [<statements>] endmethod
                   [!<label>$] <name>
```

```
<thread>          ::= [$<label>!] thread <name>
                      <activ_deactiv_constraint>
                      <[param_list> <decls>
                      [<statements>] endthread
                      [!<label>$] <name>
```

The balance of BNF pertinent to this section can be found in Appendix B1.

4.2. Statements

CRL takes a set of conventional imperative statements and augments it with a number of features:

```
<statements>      ::={<statements>
                     [<time_constraint>]}*
<statement>       ::=<null> | <timed_statement>
<timed_statement> ::=<untimed_statement> |
                     <untimed_statement> !<label>$
                     | $<label>! <untimed_statement>
                     | $<label>! <untimed_statement>
                     !<label>$
<untimed_statement> ::=<assignment> | <if>|<loop>
                      |<exit>|<call>|<block>
                      |<activation>|<read>|write>
<null>        ::=null
 /* an explicit no-op, as opposed to an empty
statement */
<assignment>::=<rhs>
<if>          ::=if <boolean_expr> then <statements>
                 {elseif <boolean_expr> then
                 <statements>}* [else <statements>]
                 endif
<loop>        ::=loop <iteration_constraint_or_
                 assertion> <statements> endloop
<exit>        ::=exit
<call>        ::=call <object_name> . <method_name>
                 ([<actual_param_list>])
                 [<recursion_constraint_or_
                 assertion>]
<actual_param_list> ::=<actual_parameter>
                      {,<actual_parameter>}*
<actual_parameter>  ::=<name> | <expr>
<block>       ::=block <decls> <statements> endblock
<activation>::=<call>
<read>        ::=IOread (<file _name> <var_ref>)
                 [<length_assertion>]
<write>       ::=IOwrite (< file_name_> <var_ref>)
```

A statement may be associated with a timed constraint, a loop with an iteration constraint and a call with a recursion constraint (as discussed in Sections 4.5 and 4.6). There is no separate thread activation statement; rather, a call to the thread of an object activates the thread. Currently, such a call is non-blocking, and only IN parameters are allowed. If the thread is not eligible to become active yet (typically because its previous frame has not expired), then the thread will become active at the end of the frame. A corollary of this is that calls to periodic threads have no effect (and should be flagged by the compiler).

4.3. Types, variables, scalars and names

The language is strictly typed, and type compatibility is by name. All declarations are explicit, static and unique within their scopes. Redeclaration is forbidden. Declaration before use is required. Currently, only compile-time knowable integer intervals are allowed as array ranges. There are (at this time) two distinct name spaces, one for labels, and one for all other names.

All language-defined types are implicitly imported and thus visible everywhere, and may not be redefined. User-defined types need to be imported in accordance with the rules presented below.

The pertinent BNF is found in Appendix B2.

4.4. Expressions and assertions

A detailed syntax of CRL expressions is provided in Appendix B3. Among the more interesting features is the following:

```
<expr>              ::=...
<length_assertion>  ::=assert length <manifest_rhs>
                       | ?assert length <var_name>
```

where <length_assertion> can be used to restrict the length of data entered into a variable providing possibly larger string or array storage—the value can then be used to refine out-of-bounds checks, and to prove bounds on loops and recursion. ?assert length requires compile-time or link-time specification of length. Specification provides a value for <var_name> as input to the partial evaluator (see below), which must be a declared and otherwise unused integer variable; it is a compile-time/link-time error for the value to (be negative or to) exceed the declared length of the string or array. The IO class handles input and output to files and other devices <file_name> may be mapped to a device).[1]

Currently, there are three kinds of assertions: iteration and recursion assertions, and length assertions on read data. The first two are always specified in line; the last has two forms—one in line and one entailing input at compile or link time. It is legal for no value to be provided for the latter type of assertion if a default value (e.g. the declared length of the identifier) can be supplied for such an assertion.

Other kinds of assertion will be added to the language as time goes by, principally for use by the analysis/transformation engine; for example, if the semantics of OUT and INOUT parameters is generalized, as an assertion assert noalias <var_list> on a call, may be used to assert that the parameters in <var_list> will not be aliased at a call site (or at entry).

4.5. Constraints

As indicated above, constraints are (currently) of four types: time constraints, iteration constraints, constraints on activation and deactivation of threads and constraints

[1] Whether <file_name> is local or a system name will be resolved when class IO is defined.

on (direct) recursion. The first three are discussed in this section; recursion constraints are treated in the following section. Constraints can either be verified at compile time, or viewed as run-time checked assertions. The keyword assert is reserved to distinguish the latter type; assert will later also be used (possibly with modification) to label expected link-time input to the partial evaluator, but this is not currently implemented.

A time constraint is specified as:

```
<time_constraint>::=  timeconstraint
        {<absolute_duration>|<relative_duration>
        |<absolute_bound>|<relative_bound>}*
        endtimeconstraint
<absolute_duration>::= use <abs_time_expr>
<relative_duration>::= userelative <rel_time_expr>
<absolute_bound>    ::= <nosoonerthan_absolute>
  |<nolaterthan_absolute>
  |<nosoonerthan_absolute> <nolaterthan_absolute>
<nosoonerthan_absolute>::= nosoonerthan
                           <abs_time_expr>
<nolaterthan_absolute> ::= nolaterthan
                           <abs_time_expr>
<relative_bound>    ::= <nosoonerthan_relative>
  |<nolaterthan_relative>
  |<nosoonerthan_relative> <nolaterthan_relative>
<nosoonerthan_relative>::= nosoonerthanrelative
                           <rel_time_expr>
<nolaterthan_relative> ::= nolaterthanrelative
                           <rel_time_expr>
```

Time expressions have the following syntax:

```
<abs_time_expr>  ::= (<manifest_integer>)
                    |(<manifest_intefer>, <label>)
<rel_time_expr>  ::= (<label> <manifest_integer>
          <flags>) | (<label> <manifest_integer>
          <flags>, <label>)
```

<label>s must of course be declared labels, and the statements which they label must be appropriately related to the current statement (see below). In the second form for each type of time expression, the second label must be a label of the current statement. <flags> will be used to distinguish classes of constraints involving a statement in a loop, or a statement in one procedure and a label in another.

An iteration constraint is:

```
<iteration_constraint_or_assertion> ::=
  <iteration_constraint>|<iteration_assertion>
<iteration_constraint> ::= <nomorethan_iterations>
<iteration_assertion> ::= assert
  <iteration_constraint> | <nofewerthan_iterations>
  <nomorethan_iterations>
<nofewerthan_iterations> ::= nofewerthaniterations
                             <manifest_integer>
<nomorethan_iterations> ::= nomorethaniterations
                            <manifest_integer>
```

An activation/deactivation constraint is

```
<activ_deactiv_constraint> ::=
          activationdeactivationconstraint
          <type_activ_deactiv_constraint>
          endactivationdeactivationconstraint
<type_activ_deactiv_constraint> ::= periodic
          <absolute_duration> <first_activation>
          | atevent frame <absolute_bound>
<first_activation> ::= firstactive <absolute_bound>
          | firstactive atevent <event_name>
```

For now, deactivation is implicit, and deadlines are defined by the end of the frame or period. Clearly, an exact time may be defined as bounded between two identical times. Currently all time constraints are to the end of the statement. It is likely that this constraint syntax will later be extended.

Constraints should be statically manageable without combinatorial explosion; this entails that timing constraints between conditionally executed statements be restricted in form, and that minimum absolute timing constraints of a conditionally executed statement be satisfied by non-execution of a given instance of the statement.

4.6. Recursion

A program is said to have direct recursion only if for each function (method or thread) *f*, any invocation of *f* occurring during a call to *f* occurs as a result of a call to *f* in the body of *f*. In a program with direct recursion only, recursion can be handled by a construct similar to iteration constraints, i.e.:

```
<recursion_constraint_or_assertion>::=
                             <recursion_constraint>
                             | <recursion_assertion>
<recursion_constraint> ::= <nomorethan_rec_depth>
          | <rec_flag> <nolessthan_rec_depth>
          <nomorethan_rec_depth>
<recursion_assertion>  ::= assert
                             <recursion_constraint>
<nolessthan_rec_depth> ::= minrecursions
                             <manifest_integer>
<nomorethan_rec_depth> ::= maxrecursions
                             <manifest_integer>
<rec_flag>             ::= ε | any | all
```

For a recursion constraint which is not statically verifiable (possibly modulo earlier checked assertions, such as data length), the compiler will add an extra parameter to the function, and a test on each base case. Some care will need to be taken both in the definition (using <rec_flag>) and in checking if the function makes multiple calls to itself.

As indicated, recursion is currently limited to direct recursion. Indirect recursion (detectable by reachability analysis of the call graph) poses a greater difficulty if recursion constraints cannot be statically verified, since recursion state will have to be passed between threads, each possibly with its own recursion constraint. Possible solutions include keeping track of recursion only at some locations, using a single artificial thread to manage the recursion and keeping counters as local state, or extending the parameter list to include recursion depth for all functions in the call-graph strong component. Our proposed solution is to require that indirect recursion with recursion constraints not verifiable at compile time be required to have a reducible call graph, and to keep track of recursion at region head nodes (back-edge targets); the compiler may be permitted to perform a limited amount of node splitting to create a reducible region.

4.7. Static semantics

The language admits of standard Pascal-like static restrictions on type consistency, function arity, name conflict and so on. A few similar constraints on the use of names (such as the second label in time expressions) are sketched above. The compiler will also insert checks for array bounds (although some may be removed by optimization) and other similar enforcement mechanisms.

There are two additional semantic restrictions. First, there is a restriction on the reference of the first label on a time expression, relative to the location of the current statement (see below); this constraint can be verified by standard attribute grammar techniques. Second, inherent in the systems engineering nature of the language, constraints and assertions naturally impose requirements for compile-time verification or run-time checking.

In general, timing constraints must be verified at compile time. Thread activation/deactivation constraints are enforced by the scheduler and the run-time system. Iteration and recursion constraints must either be verified at compile time (possibly in the partial evaluator) or checked at run time; a compiler flag will be used to require verification or permit run-time checks of various types of constraints. Assertions are assumed true at compile (or link) time, and may be used by the partial evaluator and subsequent analyses and transformations. It is however required that they be checked at run time, and it is a fatal error for a run-time check of a constraint or assertion to fail.

4.8. Other features: exception handling and IPC

We are working on the definition of exception handling and intertask communication. On the former, we have defined a preliminary model of synchronous exceptions and their handling. To enable predictable real-time performance, we use a static program analysis related to the program summary graph 9. Currently, expressions of constants, scalars, and some I/O are allowed [65].

For the latter, we have defined messages, handled by a class `Channel`, as one form of expression. We are looking to extend this model with rendezvous.

5. AN OPEN-ENDED SUITE OF CONFLICTING OBJECTIVES

"Conventional" programming languages have neglected standard system engineering objectives, and concentrated primarily on compilability and on computational paradigms (imperative, functional, declarative, object-oriented, data-flow ...). In the field of real-time computing, specialized languages have been developed (compare [66]) which accommodate the real-time objective to varying extents, through libraries, or, more properly, through language-level mechanisms. In specific application domains, specialized languages have also emerged, including features specific to those domains (such as the `grip` primitive for a manufacturing robotics language).

What is clearly missing is a suite of non-functional objectives, together with a means of associating objectives to application entities, provided, as part of language design and implementation, in a domain-independent fashion. Hundreds of such objectives have been identified (see, for instance, [67]). In what follows, we focus on some of the important systems objectives we have addressed or intend to address in CRL.

5.1. "Quality of service"

"Quality of service" is a measure of the quality of observable results computed by processes. This is perhaps the only functional criterion on the list of system objectives of immediate concern. Given unlimited resources, including time, and a fault-free environment, every function can be implemented and subsequently executed to produce correct and maximally precise results. This, a function could return an arbitrarily precise value for π, trace the complete effects of one atom's vibration on another (either classically or quantum-mechanically), determine the age of the visible universe predicted by a given model, or do something more mundane, such as automatically and perfectly adjusting control surfaces on a landing aircraft. Unfortunately, the world is not fault-free, computer resources are not unlimited, and time is of essence in many applications; thus, we frequently have to accept some degradation in the timeless, precision or availability of results.

Thus, there is an obvious trade-off between quality of service (also a recent term) and other objectives. In computer systems, this trade-off has been addressed through truncation/approximation error, ever since the advent of numerical analysis (thus, well before there were computers); through exception handling in languages; and more recently through an attempt to extent uniprocessor scheduling theories under the names of "imprecise computation" and "multiversioning".[2]

To accommodate quality of service at the programming language level, we propose to use extensions in the spirit similar to those found in High-Integrity PEARL (HI-PEARL) [43] or Flex [41]. Specifically, HI-PEARL allows providing multiple versions of task bodies, ranked by quality. (Typically, a lower ranked version will employ a less time-consuming or less potentially-faulty

[2] There is also a relationship to strategies for distributed computation, particularly in heterogeneous multiprocessor systems, including cloning, migration and dynamic allocation of resources. While we expect also to provide language support for these decisions, we do not here consider this aspect further.

algorithm that delivers somewhat less exact results.) Since what constitutes a unit of algorithmic computation depends on the language (as well as the language's paradigm), we believe it to be reasonable to aim for expressibility of quality of service per first-class entity. Thus, the definition of any function in a functional language, any object in an object-oriented one, and so forth, could be augmented with a quality of service description (as a specification, an attribute, a compiler or run-time pragma ...).

In some cases, we may also wish to provide quality of service annotations for some subordinate entities, such as methods of an object, computation of an intermediate result, etc. This imposes, however, a requirement of a semantic policy for "inheritance" of service quality, such as "If an entity A whose quality of service is required to be x calls another entity B, B must statically be able to provide, and must dynamically provide, service of quality at least x." Other policies, and mechanisms such as explicit relaxation through pragmas/assertions, can be formulated.

This leaves the question of what exactly constitutes quality, and how one can evaluate or rank it. These are not trivial issues. For one, the measure of quality may well be (conceptually) continuous, where for instance the accuracy of an iterative algorithm may improve the more iterations the loop has time to perform.

Even with multiversioning, there may well be interesting nuances. For example, a procedure that seeks argument values that result in zero, for a parameter specifying an arbitrary real-valued differential function, can be implemented by the more time-consuming per iteration but more quickly converging Newton–Raphson method or by the less time-consuming per iteration but more slowly converging bisection method. However, there are anomalous situations which destabilize Newton's method (nearly horizontal tangent lines) or which slow the rate of convergence (multiple roots), for which the standard speed-of-convergence guarantees for Newton's method fail. Should such a situation be encountered, it may be that bisection actually produces an acceptably accurate result with less cost than for Newton's method. Thus, maintaining an expected quality of service may require dynamic decisions, and may not be implemented simply as a fixed choice among a fixed number of independent implementations.

In addition, as the previous example suggests, where there are two or more dimensions, either in terms of quality of service, via resource constraints (e.g. on computation), or a combination, such as iteration cost and number of expected iterations to achieve a given tolerance, it may be reasonable to have a two- or higher-dimensional space of alternatives, indexed by the resources available and service provided in each applicable dimension. Selection will then be guided by maximization of some combination of objectives, subject to maximum constraints provided by available resources, and minimum constraints on some desiderata on quality.

5.2. Real-time performance

In complex computer systems, the issue of real-time performance is of immense significance and difficulty.[3] Moreover, while strict ("hard") deadlines associated with program units are still of major significance, other measures of a "softer" or more aggregate nature—proportion of deadlines made, allowable sequence of failures, and so on—are as important in complex applications such as multimedia. Moreover, while systematic, automated schedulability analysis [3–5] has aided in developing modern real-time systems, modern complex system engineering tools also need to accommodate assignment and allocation [68–70], design structuring and other decision processes. Again, for these other steps, measures (and predictors!) of real-time performance beyond hard deadlines will be needed [71].

While we recognize that complexity and computability impose limitations on all formalisms, including timing constraint languages and their interpretations, nevertheless we are tempted to propose that as rich a set as possible of real-time performance expression primitives be devised. Later, interpreters, compilers, schedulability analysers, assignment and allocation tools can indicate which expressions can and which cannot be analyzed, resolved, unified, transformed ... or otherwise processed. Some sort of heuristics, in the spirit of Prolog's cuts, can probably be included. Every applicable unit of computation (variable access, expression evaluation, call, statement, method evaluation, task activation and execution ...) or combination of units of computation should therefore and "in principle" be allowed to be augmented with some expressions of timing requirements. Particular measures may include deadlines, intervals, laxities, bounded elapsed and response times, in the hard sense and in the soft sense (i.e. general time-value functions), per unit of computation and in an aggregate fashion. Research will be needed to determine what temporal relationships between units will allow a reasonably accurate pre-run-time analysis of feasibility. To a significant extent, we have already discussed in this paper how real-time performance is addressed in CRL.

5.3. Fault tolerance

Fault tolerance for traditional programming languages

[3] Note this is not equivalent to the scheduling-theoretic model of more-or-less independent pure computation tasks executing on a simple network, with simple temporal constraints. These latter situations, addressed in the OR community in the 1940–50s, were readdressed with some extensions by the real-time community in the past twenty years. In those efforts, the focus has been on provably correct scheduling predictions and schedule construction—which however only makes sense for well-understood and very simple cases (essentially a uniprocessor with a set of independent pure computation tasks). While these results applied to the standard computing model of the 1940–50s, and also to job-shop and other non-computer applications, they can in general be used only as first approximations (by no means provably either feasible or optimal) for the systems of 1970–80s, and even less for the complex computer systems of the 1990s and the future.

is addressed by a number of techniques, including exceptions, rollback and replication, but these are unlikely in and of themselves to be sufficient for complex systems. Some limited language support is available for replication and rollback; however, both of these have global implications, and must be used with care in a complex composable system. We suggest adopting current and proposed pragmas supporting replication and optimistic/speculative execution, while where possible deferring decisions based on those pragmas until we have a better picture of both global and local system resources and requirements.

Exception-based mechanisms are promising, but current language support is arguably insufficient. First, there is a tendency to address exception propagation only at certain sites, such as an object-method interface, and most user exceptions are handled as returns from calls. Second, while some languages provide mechanisms for intercepting some system exceptions, most do not, and existing interception mechanisms (as in C/UNIX) are system dependent and often not comprehensible. Third, most languages support synchronous exceptions only. Fourth, except in a few cases (such as Real-Time Euclid), most languages consider timing exceptions "primitive" events that occur in hardware.

We propose that the basic timed-computation language model be augmented with an exception detection, propagation and handling model. Every unit of computation in the new model may trigger an exception, detectable through some form of asserts, axiomatic or other specifications. Exception handling will depend on the availability of suitable handlers within the appropriate scope. The actual handling would be done by user-defined code, employing both conventional and extended mechanisms (such as rolling a value back to a previous value and so forth). System exceptions will be interceptible within the model. Propagation will proceed with control and data flow (such as through return parameters or static variables)—this aspect of the model should in fact be kept as synchronous as possible. Rules will be provided to augment existing visibility and encapsulation rules, while both providing for adequate propagation and preserving the philosophy of normal execution flow as much as possible. An approach to augmenting the basic timed-computation language model with an exception detection, propagation and handling model is described in [65].

5.4. Security

We have begun considering how security objectives can be expressed in an integrated manner via annotations. While security violations can be extremely subtle, at least some forms of security can be expressed and handled within a constraint mechanism; in particular, we consider access to secure data/code on untrusted platforms, and transmission of secure messages across untrusted links. We propose that a constraint mechanism can be used to identify secure data, code and messages (conceptually at the same level of granularity as our other annotations, although in practice probably largely at a higher level), to guide assignment of these units to secure devices. We can then check at compile time for this type of security violation, and focus security safeguards during implementation, translation, and execution, for units requiring the most security, or for the most vulnerable communications.

6. IMPLEMENTATION STATUS

As stated above, it is our intention to make the language as widely distributed and used as possible. For this reason, we target CRL to a C++-based platform.

The front end of our CRL compiler translates CRL code into a subset of C++ plus a number of auxiliary tables. The C++ code is generated in two forms with labels and comments: a compact, more-or-less statement-by-statement form for user access, and to reflect high-level structure, and a three-operand-like form for low level timing analysis and transformation. Auxiliary information includes a line-level correspondence among the three forms, and databases of constraints and assertions (on timing, iteration, recursion, activation, etc.). Constraints and assertions are tied to the C++ code via line numbers and labels. In general, constraints will require validation at compile or link time, and a program in which there is an unproven constraint will not be allowed to run; assertions will typically translate into run-time checks, much like checks on array bounds, although some assertions may be provable or redundant (implied by the success of previous assertions), in which case the corresponding run-time tests may be elided.

The two C++ forms are used to derive timing information, and to construct a number of auxiliary representations, including the control flow graph of each procedure. Standard CRL types (boolean, integer, array, ...) are provided as C++ classes. Since CRL is translated into a restricted subset of C++ (plus assertions) it should be possible to use or modify some C++ classes and routines, and for a developer to write certain types of functions in C++ modules which can then interact with the CRL code.

The run-time system for CRL takes care of architectural dependences, assignment, scheduling, activation, and message-passing. The design is robust, and allows various disciplines to be used in each case.

Our intention is to ultimately provide a flexible and extensible platform where different topologies, network properties, node architectures, kernel and OS-disciplines and architectures can be used, in a "pluggable" manner. As a minimum, we are currently looking to provide both basic well-known features as well as those developed within the Real-Time Computing Laboratory (such as for instance both EDF and RMF, and the Least-Space-Time-First (developed in our Lab [72]) scheduling disciplines.

6.1. Progress to date

The language is still to some extent in transition, and exceptions have not yet been added. A scanner and parser exist for the full language syntax (based on LEX and YACC), and attribute-grammar-based semantic analysis routines for type inference, constant recognition, and timing analysis exist for a core of the language. There is a basic distributed runtime language kernel, with network support, and with hooks for time measures, rudimentary logging and monitoring, and of course, language support. A window-based user interface is being developed. For the purpose of debugging and monitoring, we have augmented the language with a non-timed statement `dump <actual_param_list>` that dumps the variables and expressions—including character strings as in

```
''Hi there'~*&^%$& etc etc ...''
```

—to standard output.

6.2. Integrating the language into the suite of tools for engineering of complex computer systems

The language (and its implementation) is but one tool in the suite of tools. The tools associated with the language (compiler, runtime kernel and monitor, schedulability analyser and so on), and their subtools, are also part of this suite. A common user interface should be used, and the tools should be mutually and complementary invocable. Thus for instance, it should be possible to run the assignment tool for a while, then, perhaps change the program, recompile it, do some reanalysis, and rerun the assignment tool.

Let us now consider in general terms how application development might proceed in this environment. The developer begins with a high-level requirements or specification document, translating into a graphical formalism such as CaRT-Spec [77], or a textual formalism, or a combination. As code is developed in CRL (or perhaps partly in C++), we can, under user control, extract timing information, check on constraints/assertions, preprocess for schedulability, attempt local transformations, extract interface descriptions, and perhaps translate boundary-crossing constraints into demands on the interface.

Once a program core has been developed, sample data, platforms, and missing modules (or modules which cannot be executed because of the nature of their side-effects) can be provided using the workload generator (and profiling or related tools for object-code system routines). The developer can then consider various assignment algorithms, perform more thorough analysis and transformation, check for schedulability, and combine simulation and execution in varying measures for debugging, testing, profiling, etc. The results of the analyses, transformations and simulation can be displayed to the developer, who can use them to identify problems or bottlenecks and modify code.

Throughout the development process, the user will be able to estimate whether the program will satisfy its constraints, and how well the program is meeting its objectives. In the later stages, these evaluations will rely on both analysis and simulation/execution/profiling. In simulation or symbolic execution, we will need to rely on assignment construction, and scheduler simulation, as well as simulation of the code itself.

While each of the facets we describe above is an interesting research topic in itself, in which success will significantly extend the state of the art, the interaction and synthesis between them can be expected to provide further benefits to a software development environment and continuous research and technology advancements in the field.

6.3. Interaction with scheduling

Evaluation of costs for assignment selection in a system with complex non-functional constraints, including real-time constraints, may not be possible at compile time without some prediction of the schedule to be used. Program transformation may also be difficult without some knowledge of the scheduler. However, an approximate schedule can be used. Gerber and Hong assume, for work on transformations [73], that a scheduler such as rate-monotonic scheduling is used. We currently make use of a CPM/PERT-like scheduler for evaluating assignments; we also expect to use the LSTF scheduler [72].

Figure 1 depicts the integrated environment, which consists of interacting tools, as well as other auxiliary tools, "virtual files" and so forth. To appreciate why the

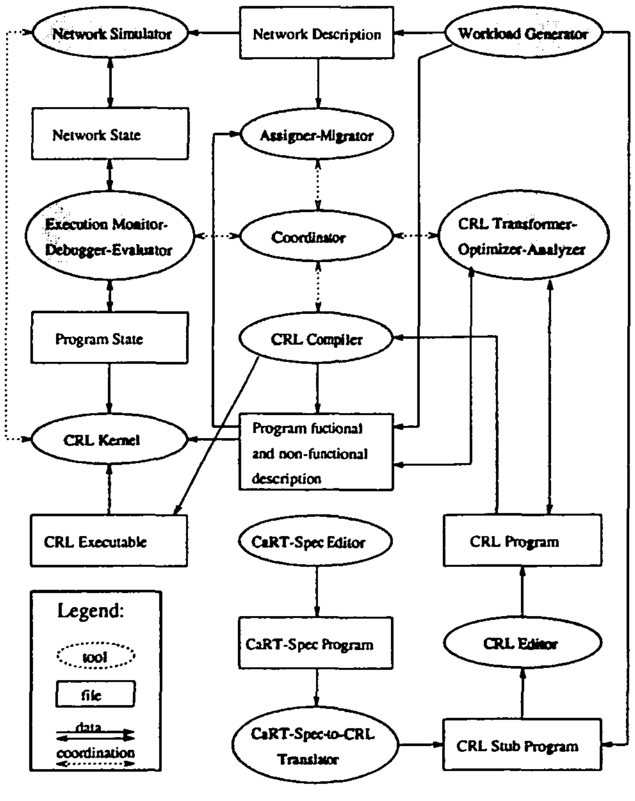

FIGURE 1. The integrated environment.

environment is constructed in this particular way, it is important to understand the inherent interactions among individual tools. Some interactions are in the rather conventional producer-consumer form. Thus, for instance, it is natural to expect the CRL compiler to take the output of the CRL editor or the CRL editor and CaRT-Spec-to-CRL translator, and in turn provide output to the CRL run-time kernel.[4] Other interactions, however, are more interesting. Thus, for instance, the transformations and concomitant performance predictions undertaken by the CRL transformer-optimizer-analyser are dependent on the assignment of CRL's modules to the processing elements in the network. Not surprisingly, the inverse dependency exists as well. Consequently, the assigner-migrator and the transformer-optimizer-analyser must coordinate their actions. In the integrated environment, all such coordination is implemented through the coordinator tool.

The low-level implementation details of operation and coordination in the integrated environment (distributed operation vs. lock-step or co-routines vs. multi pass ...) are to be decided upon later and are not presented in this paper. Referring to the Figure 1, we now briefly explain the role of each depicted item. Among the items, network simulator and CRL kernel can be used during run-time only. The rest of the tools can be used both before and during run time.[5]

The CaRT-Spec editor is used to develop, modify and manage CaRT-Spec specification programs. These programs are in turn processed by the CaRT-Spec-To-CRL Translator, which outputs stub CRL programs. The latter consist of CRL object, thread and method definitions and stub bodies. In those bodies, there are calls to methods and threads, but non-call code is substituted with cycle burners and dummy parameters are used in calls. The burners and dummy parameters are already largely sufficient to project various performance and other properties of the eventual program or to possibly test a new assignment, analysis or transformation algorithms. Should the detailed program be developed, naturally the burners and dummy parameters will be changed to their functional code counterparts.

Output from either the CRL programmer user, the CaRT-Spec-to-CRL translator or the workload generator (see below) is processed by the CRL editor, which is used to develop, modify and manage CRL programs. This editor may in fact simply pass a stub program on to the next step, which is naturally the CRL compiler.

The compiler outputs "executable" CRL code and a program description consisting of functional (such as flow graphs) and non-functional (such as desired performance objectives, timing annotations ...) parts. The compiler thus combines the functions of the actual compiler and the extractor of non-functional information (including the traditional timing extractor, which is of course the front-end schedulability analyser). The compiler interacts (through the coordinator tool) with other tools. Essentially, complete or partial (incremental) recompilation may trigger more transformations, optimization, analysis, assignment, and migration. On the other hand, code transformations, or migrations[6] may trigger recompilation. Moreover, network monitoring debugging and performance evaluation may necessitate migrations, transformations and recompilation.

The CRL transformer-optimizer-analyser is driven by primarily the functional and non-functional program description, as output by the compiler. Naturally, to enable any transformations, this description is linked (through symbol tables and so forth) to the CRL code.[7] As has been discussed already, the transformer-optimizer-analyser performs transformations in the sense of (i) reduction of the problem space that needs to be considered during analysis (ii) real-time conforming versions of traditional speed-up optimizations, (iii) optimistic code specializations such as speculative, shadow, partial evaluation, and others. As transformations are done, the CRL code is changed accordingly. Such changes trigger recompilations, as well as possibly reassignments (migrations). Indirectly, such changes also lead to changes in run-time execution, as monitored, debugged and evaluated elsewhere in the system (the latter changes may necessitate additional transformations). The transformer-optimizer-analyser also performs the traditional back-end schedulability analysis as well as other non-functional analyses (geared at safety, security and so forth). As each analysis is done, the transformer-optimizer-analyser augments appropriately the non-functional program description. Again these changes may trigger changes requested by other tools in the system (e.g. performance predictions may now differ more from observed performance or a deadline may now be missed). Also, an analysis may indicate that an objective will not be met, forcing a CRL program to be possibly re-written.

As was already mentioned, the workload generator outputs CRL programs (naturally, in stub form, similarly to that of the CaRT-Spec-to-CRL translator). The generator also outputs a program description (such as the one output by the compiler) and a network description (topology, speeds, capacities ...).

[4] CaRT-Spec [76] is our specification language. This language is resource-algebraic and should support constraints and annotations which are at least as rich as CRL's. The details of CaRT-Spec are beyond the scope of this paper.

[5] This includes all seemingly "compilation"-time tools, such as the CRL, compiler, the CaRT-Spec Editor ...

[II] Really C++—architecturally, this is irrelevant, though there are some interesting engineering issues in the implementation.

[6] A migration may trigger stub regeneration for distributed calls or even create new calls or eliminate existing calls. These actions would consequently require recompilation.

[7] In the figure it indicates that we transform the source CRL code. Whether it is so, or not (the alternative is of course to transform the executable code) is currently considered an implementation issue. Conceptually, we feel that the choice matters rather little.

The network description is used by the network simulator and the assigner-migrator.

The assigner-migrator derives and triggers, at compile and run time, an allocation of program objects (as presented in the program description and physically found in the executable CRL code) to processing elements in the network (as presented in the network description). The allocation is driven by the functional and non-functional objectives as requested in the program description, subject to network constraints and on the basis of functional and non-functional properties found in the program description. The assigner-migrator may realize that an allocation according to a particular objective may not be possible. In this case, the tool will recommend program changes (through the coordinator), triggering action by other tools or even the programmer. On the other hand, actions by other tools may of course also trigger at reallocation.

The CRL kernel executes CRL code, subject to its functional and non-functional objectives. There is a virtual replica of the kernel at every processing element in the network. The kernel coordinates directly with the network simulator on issues of remote call preparation and execution, and associated scheduling decisions (remote calls may prove to be natural preemption points, for instance). The kernel maintains a program state at every processing element.

The network simulator essentially supports network messaging, as required by CRL calls, and CRL object migration, as required by the assigner-migrator. When the integrated environment stabilizes and becomes ready for transfer, the simulator may be replaced by a real network manager. The network simulator uses the network description and maintains a global network state (in the sense of messages rather than execution CRL objects, where the latter are of course tracked through the program state maintained by the CRL kernel). As already stated, the simulator coordinates with the CRL kernel, to support remote calls.

As its name implies, the execution monitor-debugger-evaluator is used to monitor, debug and evaluate runtime execution of CRL programs. Naturally, this tool makes use of program and network state information, as provided by the CRL kernel and the network simulator. The monitor-debugger-evaluator may be requested to perform a particular function (such as reporting which conditional branch is executed), as a result of a compilation, a transformation, or an assignment of a CRL object. On the other hand, the tool may trigger a recompilation, a reassignment, or more transformations, on the basis of its observations. As expected, all such coordination with other tools is done through the coordinator.

A number of auxiliary or detail items are omitted from but are implicit in the Figure 1. User interfaces are naturally provided for the tools. The CRL compiler outputs C++ code which is post-processed in preparation for consequent compilation and linkage with CRL type and other libraries, and linkage with the CRL kernel. A rudimentary database of non-functional constraints will be provided. We may also decide to support transformations and detailed analysis of CaRT-Spec specifications (some already defined). We may choose to support a mode where the workload generator outputs CRL program descriptions, and not actual code. Consequently, these descriptions would be symbolically executed in a symbolic mode by the kernel, and evaluated/monitored accordingly. Other detailed items are likely to include platform OS and GUI interface and so forth.

7. A LAST WORD

We have presented a new high-level language—CRL—aimed at the new generation of complex real-time systems. Arguably, while no existing set of tools, including real-time languages, is entirely adequate for construction of such systems—an undoubtedly tall order—CRL we feel rather strongly is a definite step in the right direction. It is our ambition to strengthen and build CRL. While timing specifications are already provided, and fault-tolerance and quality of service mechanisms too are being incorporated, we would like to also extend the list of supported objectives with security, human factors and others. Once in a reasonable form, the language will be tested and widely distributed. It is our intention to engage in significant experimentation with this language and its concomitant tools. The experimentation should include constructing at least one realistic computer system, such as a VR/multimedia application [74].

8. ACKNOWLEDGEMENT

We are indebted to generous support partially provided for this work under the US ONR Grants N00014-92-J-1367 and N00014-93-1-1047, the US NSWC Grants N60921-93-M-1912, N60921-93-M-3095, N60921-94-M-1250 and N60921-94-M-1426, by Siemens Corporate Research collaborative grants, and by the AT&T UEDP Grant 91-134. In addition to the authors, significant contributions to the implementation of CRL have also been made by M.-H. Akşu, Y. Cheng, C. Kline and B. Liang. This work has benefited from a productive collaboration the team members of the NSWC project DESTINATION led by C. Nguyen, as well as Real-Time Computing Lab's participation in research and technology projects at Siemens Corporate Research, IBM T.J. Watson Research and Honeywell Systems Development Center. The authors are very grateful for the many constructive comments made by the four anonymous referees as well as by the following Real-Time Computing Lab members: C. Amaro, B.-C. Cheng, A. Ganesh, M. Harelick, P. Laplante, A. Silberman and P. Sinha.

REFERENCES

[1] Kligerman, E. and Stoyenko, A. D. (1986) Real-Time

Euclid: a language for reliable real-time systems," *IEEE Transactions on Software Engineering*, **SE-12** (9) 940–949.

[2] Stoyenko, Alexander D. (1984) *Turing goes Real-Time ...* Internal programming Languages Report, Department of Computer Science, University of Toronto.

[3] Stoyenko, A. D. (1987) A schedulability analyzer for Real-Time Euclid, *Proceedings of the IEEE 1987 Real-Time Systems Symposium*, December.

[4] Stoyenko, A. D. (1987) *A Real-Time Language with A Schedulability Analyzer*, PhD Thesis, Department of Computer Science, University of Toronto.

[5] Stoyenko, A. D., Hamacher, V. C. and Holt, R. C. (1991) Analyzing hard-real-time programs for guaranteed schedulability, *IEEE Transactions on Software Engineering*, **SE-17** (8) 737–750.

[6] Baker, T. P. and Shaw, A. (1989) The cyclic executive model and Ada, *The Real-Time Systems Journal*, **1** (1), 7–26.

[7] Haase, V. H. (1981) Real-time behavior of programs, *IEEE Transactions on Software Engineering*, SE-5 (7): 494–501.

[8] Halang, W. A. (1984) A proposal for extensions of PEARL to facilitate formulation of hard real-time applications, *Informatik-Fachberichte 86*, Springer-Verlag, 573–582.

[9] Leinbaugh, D. W. (1980) Guaranteed response times in a hard-real-time environment," *IEEE Transactions on Software Engineering*, **SE-6** (1) 85–91.

[10] Leinbaugh, D. W. and Yamini, M.-R. (1982) Guaranteed response times in a distributed hard-real-time environment *Proceedings of the IEEE 1982 Real-Time Systems Symposium*, December.

[11] Leinbaugh, D. W. and Yamini, M.-R. (1986) Guaranteed response times in a distributed hard-real-time environment, *IEEE Transactions on Software Engineering*, **SE-12** (12) 1139–1144.

[12] Mok, A. K., Amerasinghe, P., Chen, M. and Tantisirivat, K. (1989) Evaluating tight execution time bounds of programs by annotations, *IEEE Workshop on Real-Time Operating Systems and Software*, Pittsburgh, PA, 74–80.

[13] Niehaus, D. (1991) Program representation and translation for predictable real-time systems, *IEEE Real-Time Systems Symposium*, San Antonio, TX, December.

[14] Park, C. and Shaw, A. C. (1990) Experiments with a program timing tool based on a source-level timing schema, *IEEE Real-Time Systems Symposium*, Orlando, FL, December.

[15] Puschner, P. and Koza, C. (1989) Calculating the maximum execution time of real-time programs, *Journal of Real-Time Systems*, **1** (2), 159–176.

[16] Rate Monotonic Analysis for Real-Time Systems Project (1992) *Handbook of Real-Time Systems Analysis (DRAFT)*, Software Engineering Institute, Carnegie-Mellon University.

[17] Sha, L. and Goodenough, J. B. (1990) Real-time scheduling theory and Ada, *Computer 23* (4), 53–62.

[18] Shaw, M. (1979) *A Formal System for Specifying, Verifying Program Performance*, Carnegie-Mellon University, Computer Science Department, Technical Report CMU-CS-79-129.

[19] Shaw, A. C. (1989) Reasoning about time in higher-level language software, *IEEE Transactions on Software Engineering*, **SE-15** (7), 875–889.

[20] Shaw, A. C. (1990) *Deterministic Timing Schemata for Parallel Programs*, University of Washington, Department of Computer Science and Engineering, Technical Report 89-05-06.

[21] Stoyenko, A. D. and Marlowe, T. J. (1991) Schedulability, program transformations and real-time programming, *IEEE/IFAC Real-Time Operating Systems Workshop*, May, Atlanta, GA.

[22] Stoyenko, A. D. and Marlowe, T. J. (1992) Polynomial-time transformations and schedulability analysis of parallel real-time programs with restricted resource contention, *Journal of Real-Time Systems*, **4** (4).

[23] Stoyenko, A. D., Marlowe, T. J., Halang, W. A. and Younis, M. (1993) Enabling efficient schedulability analysis through conditional linking and program transformations, *Control Engineering Practice*, **1** (1).

[24] Harmon, M., Baker, T. and Whalley, D. (1992) A retargetable technique for predicting execution time, *Proceedings of the IEEE Real-Time Systems Symposium*, IEEE, December.

[25] Chroust, G. (1980) Orthogonal extensions in microprogrammed multiprocessor systems: a change for increased firmware usage, *EUROMICRO Journal*, **6** (2), 104–110.

[26] Halang, Wolfgang A. (1986) On methods for direct memory access without cycle stealing, *Microprocessing and Microprogramming*, **17** (5).

[27] Halang, Wolfgang A. (1986) Implications on suitable multiprocessor structures and virtual storage management when applying a feasible scheduling algorithm, *Hard Real-Time Environments, Software—Practice and Experience*, **16** (8), 761–769.

[28] Halang, Wolfgang A. and Stoyenko, Alexander D. (1991) *Constructing Predictable Real-Time Systems*, Kluwer Academic Publishers, Dordrecht-Hingham (1991).

[29] *KE-Handbuch* (1981), Periphere Computer Systeme GmbH, Munich.

[30] Schleisiek-Kern, K. (1990) Private Communication, DELTAt, Hamburg.

[31] Schrott, G. (1986) *Ein Zuteilungsmodell fuer Multiprozessor-Echtzeitsysteme*, PhD Thesis, Technical University, Munich.

[32] Tempelmeier, T. (1979) A supplementary processor for operating system functions, *1979 IFAC/IFIP Workshop on Real-Time Programming*, Smolenice, June.

[33] Tempelmeier, T. (1984) Operating system processors in real-time systems-performance analysis and measurement, *Computer performance*, **5** (2), 121–127.

[34] Baker, T. P. and Scallon, G. L. An architecture for real-time software systems, *IEEE Software*, May 1986, 50–59; reprinted in tutorial *Hard Real-Time Systems*, IEEE Press (1988).

[35] Zuse Konrad, Foreword to Halang, Wolfgang A., Stoyenko, Alexander, D. *Constructing Predictable Real-Time Systems*, Kluwer Academic Publishers, Dordrecht-Hingham (1991).

[36] BCS Specialist Group (1967) On-line computers and their languages—a language for real-time systems, *Computer Bulletin*, No. 3, 202–212.

[37] Mensh, M. and Diehl, W. (1968) Extended FORTRAN for process control, *IEEE Transactions on Industrial Electronics and Control Instrumentation*, **IECI-15**, 75–79.

[38] Pickett, M. S. (1979) *ILIAD Reference Manual*, Computer Science Department, General Motors Research Laboratories, Warren, Michigan, Research publication GMR-2015B.

[39] Kieburtz, R. B. and Hennessy, J. L. (1976) TOMAL—a high-level programming language for microprocessor process control applications, *ACM SIGPLAN Notices*, **11** (4), 127–134.

[40] Lee, I. and Gehlot, V. (1985) Language constructs for distributed real-time programming, *Proceedings of the IEEE 1985 Real-Time Systems Symposium*, December.

[41] Lin, K.-J. and Natarajan, S. (1988) Expressing and maintaining timing constraints in FLEX, *Proceedings of the IEEE 1988 Real-Time Systems Symposium*, December.

[42] Ishikawa, Y., Tokuda, H. and Mercer, C. W. (1990) *Object-Oriented Real-Time Language Design: Constructs for Timing Constraints*, Department of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-90-111.

[43] Stoyenko, A. D. and Halang, W. A. (1993) High-integrity PEARL: a language for industrial real-time applications, *IEEE Software*, July.

[44] Baker, T. P. (1991) Stack-based scheduling of real-time processes, *The Real-Time Systems Journal*, **3** (1), 67–100.

[45] Giering, E. W. III and Baker, T. P. (1989) Toward the deterministic scheduling of Ada tasks, *Proceedings of the IEEE Real-Time Systems Symposium*, December, 31–40.

[46] Chapman, R., Wellings, A. and Burns, A. (1994) Integrated program proof and worst-case timing analysis of SPARK Ada, *Proceedings of the Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June.

[47] Gopinath, P. and Gupta, R. (1994) Correlation analysis techniques for refining execution time estimates of real-time application, *IEEE Workshop on Real-Time Operating Systems and Software*.

[48] Gupta, R. and Spezialetti, M. (1994) *Busy-Idle Profiles and Compact Task Graphs: Compile-Time Support for Interleaved and Overlapped Scheduling of Real-Time Tasks*, University of Pittsburgh Technical Report TR-94-24.

[49] Marlowe, T. J. and Masticola, S. P. (1992) Safe optimization for hard real-time programming, *Proceedings of IEEE Second Int'l Conf. on Systems Integration*, May.

[50] Mueller, F. and Whalley, D. B. (1994) On debugging real-time applications, *Proceedings of the Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June.

[51] Mueller, F., Whalley, D. B. and Harmon, M. (1994) Predicting instruction cache behaviour, *Proceedings of the Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June.

[52] Nirkhe, V. and Pugh, W. (1992) Partial evaluation of high-level imperative languages, with applications in hard real-time systems, *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, January.

[53] Nirkhe, V. and Pugh, W. (1993) A partial evaluator for the Maruti hard real-time system, *Real-Time Systems*, **5** (1), 13–30.

[54] Park, C. Y. (1993) Predicting program execution times by analyzing static and dynamic program paths, *Real-Time Systems*, **5** (1), 31–62.

[55] Spezialetti, M. and Gupta, R. (1994) Timed perturbation analysis: a static analysis approach for the non-intrusive monitoring of real-time computations, *Proceedings of the Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June.

[56] Tsai, J. J. P., Fang, K.-Y. and Chen, H.-Y. (1990) A noninvasive architecture to monitor real-time operating system, *IEEE Computer*, March.

[57] Vrchoticky, A. (1994) Compilation support for fine-grained execution-time analysis, *Proceedings of the Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June.

[58] Wedde, H. F., Korel, B. and Huizinga, D. M. (1994) Formal timing analysis for distributed real-time programs, *Real-Time Systems*, **7** (1), 57–90.

[59] Wolfe, V. F., Davidson, S. and Lee, I. (1993) RTC: language support for real-time concurrency, *Real-Time Systems*, **5** (1), 63–87.

[60] Younis, M., Marlowe, T. and Stoyenko, A. (1994) Compiler Transformations for Speculative Execution in a Real-Time System, *Proceedings of the 15th Real-Time Systems Symposium*, San Juan, Puerto Rico.

[61] Stoyenko, A. D. and Baker, T. (1994) Real-time schedulability-analyzable mechanisms in Ada9X, *Proceedings of the IEEE*, 95–107, January.

[62] DIN 44300: Informationsverarbeitung, No. 161 (1972) Realzeitbetrieb.

[63] Gerber, R. and Hong, S. (1994) Compiling real-time programs with timing constraints refinement and structural code motion, CS-TR-3323, UMIACS-TR-94-90, Department of Computer Science, University of Maryland.

[64] Chung, T. M. and Dietz, H. G. (1995) Language constructs and transformation for hard real-time systems, *Proceedings of the Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June.

[65] Marlowe, T. J., Stoyenko, A. D., Masticola, S. P. and Welch, L. R. (1994) Schedulability-analyzable exception-handling for fault-tolerant real-time languages, *Real-Time Systems*, **7** (2), 183–212.

[66] Stoyenko, A. D. (1992) Evolution and state-of-the-art of real-time languages, *Journal of Systems and Software*, **18**, 61–84.

[67] Nguyen, C. M. and Howell, S. L. (1992) *Systems Design Factors: The Essential Ingredients of Systems Design*, Informal Report, Version 0.3, Naval Surface Warfare Center.

[68] Amaro, C. C., Harelick, M., Sinha, P., Stoyenko, A. D., Laplante, P. A., Marlowe, T. J., Cheng, B.-C., Jones, N. and Tugcu, T. (1994) Economics of resource allocation, *1994 Complex Systems Engineering and Assessment Technology Workshop*, Beltsville, MD.

[69] Marlowe, T. J., Stoyenko, A. D., Laplante, P. Daita, R. S., Amaro, C. C., Nguyen, C. M. and Howell, S. L. (1994) Multiple-goal objective functions for optimization of task assignment in complex computer systems, *19th IFAC/IFIP Workshop on Real Time Programming*, Isle of Reichenau, Lake Constance, Germany.

[70] Stoyenko, A. D., Welch, L. R. and Cheng, B.-C. (1994) Response time prediction in object-based, parallel embedded systems, *Microprogramming and Microprocessing*, **40** (2&3), 135–150.

[71] Stoyenko, A. D. and Georgiadis, L. (1992) On optimal lateness and tardiness scheduling in real-time systems, *Computing*, **47**, 215–234.

[72] Cheng, B.-C., Stoyenko, A. D. and Marlowe, T. J. (1994) Least-space-time-first scheduling algorithm: a policy for complex real-time tasks in multiple processor systems, *Proceedings of the 19th IFAC/IFIP Workshop on Real Time Programming*, June.

[73] Hong, S. and Gerber, R. (1993) Compiling real-time-programs into schedulable code, *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, June.

[74] Laplante, P., Stoyenko, A. D. and Marlowe, T. J. (1994) A language framework for real-time image processing, *19th IFAC/IFIP Workshop on Real Time programming*, Isle of Reichenau, Lake Constance, Germany, June.

[75] Davidson, S., Lee, I. and Wolfe, V. (1991) Time atomic commitment, *IEEE Transactions on Computers*, **40** (5), 573–583.

[76] Furht, B. *et al.* (1991) *Real-time UNIX Systems Design and Applications Guide*, Kluwer Academic Publishers, Boston.

[77] Stoyenko, A. D., Marlowe, T. J. and Laplante, P. A. (1995) A Description Language for Engineering of Complex Real-Time Systems, *Journal of Real-Time Systems*, **7** (in press).

APPENDIX A : CRL EXAMPLES

To present the syntax as well as the power of CRL in expressing timing constraints, we show two examples written in CRL. The first one, taken from [75], describes the removal of defective containers out the assemble line in a chemical plant. On detection of a defective container, that container should be removed and discarded, delaying the production process. A robot with two arms is used to pick the defective container out the assembly line. The quality of the containers is monitored by a quality control system which coordinates with the robot to lift the defective container. To allow the container to be reachable by the robot arms, removal should start after at least 5 seconds of detection. The container should be lifted within 10 seconds of being detected; otherwise, the line should stop. The quality control system will set an alarm in case of failure in lifting the container.

To pick up the container, the arms should first move towards the container and then grab it. The arms should not take more than 1 second to move or to grab. The arms should not be moved before 100 ms of grabbing the container to ensure stability.

```
types

% class for handling alarms
classdefinition Alarm
  % Class interface specification
  export Alarm
    methodinterface set
    endmethodinterface set
endclassdefinition Alarm,

% Class for controlling an arm of the robot
classimplementation Arm_Controller
  % Class interface specification
  export Arm_Controller
    methodinterface Prepare_Lift
    endmethodinterface Prepare_Lift
    methodinterface Perform_Lift
    endmethodinterface Perform_Lift
  % Constants declaration action
  consts
    100  target_x,
    100  target_y,
    0    origin_x,
    0    origin_y,
    1    time_allowed
  endconsts

% Move robot arms to a new position
method Prepare_Lift
  % The method timing constraints
  timeconstraint
    nolaterthan (1)
  endtimeconstraint
  % Variables declaration section
  vars
    rational  current_x,
    rational  current_y,
    integer   direction_x,
    integer   direction_y
  endvars
  IOread(ROBOT current_x)
  if current_x)> target_x
    then
      direction_x := -1
    else
      direction_x := 1
  endif
  IOread(ROBOT current_y)
  if current_y> target_y
    then
      direction_y := -1
    else
      direction_y := 1
  endif
  call self.Move_Arm(target_x,direction_x,
                     target_y,direction_t)
  % move the arms towards the container
endmethod Prepare_Lift

% Make robot arms grab the defective container
method Perform Lift
  % The method timing constraints
  timeconstraint
    nolaterthan (1)
  endtimeconstraint
  % remove the defective container
  call self.Grab_Arm() !grab$
  call self.Move_Arm(origin_x,-1,origin_y,-1
  timeconstraint
    nosoonerthanrelative (grab 1 local)
  endtimeconstraint
  % arm motion should not start before
  % 100ms to ensure stability
endmethod Perform_Lift

% Interact with the robot to actually move arms
method Move_Arm
  % Method interface specification
  in rational x,
    integer dir_x,
    rational y,
    integer dir_y
  % the method body
endmethod Move_Arm

% Interact with the robot to actually grab arms
method Grab_Arm
  % the method body
endmethod Grab_Arm

endclassimplementation Arm_Controller,

classimplementation Quality_Monitor
  % Class interface specification
  export threadinterface Monitor_Container
    endthreadinterface Monitor_Container
  import Arm_Controller Alarm
  % defined types
  types
  array 1..2 of Arm_Controller endarray Arms
  endtypes
  % Constants declaration section
consts
```

```
    5     START,
    10    END,
    20    PERIOD,
    0.001 TOLERANCE
  endconsts
  % Variables declaration section
  vars
    Arms arm,
    Alarm alarm
  endvars

% Control the robot arms to remove the defective
% container without disturbing the production line
method Remove_Container
  % Method interface specification
  in integer start_time,
    integer deadline
    out boolean indicator
    vars
      boolean robot_status,
      integer lindex
    endvars
    block
      IOread(ROBOT robot_status)
      if not robot_status
        then
            indicator := 1          % failure
        else
            lindex := 1
            loop nomorethaniterations 2
            call arm(lindex).Prepare_Lift()
            lindex := lindex + 1
        endloop
        lindex := 1
      loop nomorethaniterations 2
        call arm(lindex).Perform_Lift()
        lindex := lindex + 1
      endloop
    endif
    indicator := 0                  % Success
  endblock
  % The method timing constraints
  timeconstraint
    nosoonerthan (start_time)
    nolaterthan (deadline)
  endtimeconstraint
endmethod Remove_Container

% Periodic container monitor to remove defective
% containers without blocking the production line
thread Monitor_Container
  % Period and any activation constraints
  activationdeactivationconstraint
    periodic use (PERIOD)
    firstactive nosoonerthan (START)
  endactivationdeactivationconstraint
  vars
    rational status,
    integer now,
    boolean failure
  endvars
  % Get the status of the current container
  IOread(SENSOR status)
  % if the container is defective - -> remove it
  if status < TOLERANCE
    then
      IOread(TIME now)
      call self.Remove_Container
      (now+5,now+10,failure)
      if failure
        then
          call alarm.set()
      endif
    endif
endthread Monitor_Container

endclassimplementation Quality_Monitor

endtypes
```

The second example treats an aircraft navigation control system, similar to that discussed in [63].

The route of the aircraft will represented by a set of goal coordinates (stored in the GOAL array). We assume that that set of coordinates will be provided by another module and passed as a parameter to the navigation control thread. The algorithm can be summarized in three steps. In the first, the process samples the aircraft's current coordinates, direction (heading), roll, and its ground speed. Second, it consults the GOAL array for the next coordinate to target and calculate the relative attitude and the new direction angle. Finally, it adjusts throttle and roll to move to a new coordinate point. For simplicity, we consider a two-dimensional abstraction of navigation control problem. Assume the following timing constraints imposed by the problem:

(i) Control update should be done every 20 ms.
(ii) All measurements updates should be done within the first 5 ms in each period.
(iii) All throttle and flap changes must be made within 3 ms of the actual ground speed reading.

The CRL code is shown below. One important observation is the use of labels to express timing constraints relative to some other point, as with label read_stat, referenced when writing the timing constraints imposed on the execution of the block in the thread control. Another observation is the flexibility of expressing timing constraints on an statement or a group of statements (block) in addition to on methods or threads.

```
types

record
  vars
    rational x,
    rational y,
    boolean passed
  endvars

endrecord POINT,

array 1 . . 100 of POINT endarray GOAL,

% Definition of the velocity class
classdefinition velocity
```

```
  export velocity
    methodinterface get
    endmethodinterface get
endclassdefinition velocity,

classimplementation navigation
  % Class interface specification
  export threadinterface control
      in GOAL goal
    endthreadinterface control
  import velocity
  % Constants declaration section
  consts
    100      NCOORD,
    400      VHIGH,
    0.001    EPS
  endconsts
  % Variables declaration section
  vars
    rational x,             % Current x-coordinate
    rational y,             % Current y-coordinate
    rational theta,         % direction angle
    rational speed,         % Current velocity
    rational roll,          % Current roll
    rational throttle,      % The aircraft throttle
    velocity vel            % Velocity monitor
  endvars

% Update the current status by reading sensors
method update_status
  % The method has a deadline of 5 timeconstraint
    nolaterthan (5)
  endtimeconstraint
  IOread(GPS x)             % Read coordinates
  IOread(GPS y)
  IOread (NAV theta)        % Read the current angle
  call vel.get(speed)       % Read the current speed
endmethod update_status

% Calculate the relative attitude and
% the new angle adjustment
method compRelAtt
  % Method interface specitcation
  in rational theta,
    rational x,
    rational y,
    rational gx,
    rational gy
  out rational rtheta
  % The body should be here
endmethod compRelAtt

% Calculate delta theta (angle deviation)
% if the aircraft velocity reaches maximum
method safeDtheta
  % Method interface specification
  in rational rtheta,
    rational roll,
  out rational dtheta
  % The body should be here
endmethod safeDtheta

% Compute the new flap of the aircraft based on
% roll, velocity and required angle deviation
method compFlapw
  % Method interface specitcation
  in rational roll
    rational speed,
    rational dtheta
  out rational wflap
  % The body should be here
endmethodcomp Flapw

% Compute the new throttle of the aircraft based
% on roll, velocity and required angle deviation
method compThrottle
  % Method interface specification
  in rational roll,
    rational speed,
    rational dtheta
  out rational throttle
  % The body should be here
endmethod compThrottle

% The periodic navigation control
thread control
  % Period and any activation constraints
  activationdeactivationconstraint
    periodic use (2)
    firstactive nosoonerthan (5)
  endactivationdeactivationconstraint
  % Interface specification
  in GOAL goal
  inout integer index
  % Declaration section
  vars
    rational gx,
    rational gy,
    rational rtheta,
    rational abs_rtheta,
    rational wflap
  endvars
  % read the current measurements
  call self-update_status() !read_stat$
  block
    % Get the next target coordinates
    if goal(index).passed
      then
        gx := goal(index).x
        gy := goal(index).y
        index := index+1-((index+1)/NCOORD)*NCOORD
    endif
    % Using relative attitude w.r.t target to
    % compute angular adjustment
    call self.compRelAtt(theta,x,y,gx,gy,rtheta)
    call rtheta.abs(abs_rtheta)
    if abs_rtheta < EPS
      then
        dtheta := 0
      elseif speed < VHIGH
        then
          dtheta := rtheta
        else
          call self-safeDtheta(rtheta,roll,dtheta)
    endif
    % Adjust flap and throttle for heading
    call self.compFlap2(roll,speed,dtheta,wflap)
    call self.compcompThrottle
    (roll,speed,dtheta,throttle)
    IOwrite(THROT throttle)
    IOwrite(FLAP wflap)
  endblock
  timeconstraint
    nolaterthanrelative (read_stat 3 local)
    % to generate the output fast enough
  endtimeconstraint
endthread control

endclassimplementation navigation

endtypes
```

The above examples demonstrate some of the distinct features of CRL. In addition to providing the ability of associating timing constraints to processes (thread), every simple or compound statement in CRL can have associated absolute or relative timing constraints. In the assemble line example, the `Perform_lift` method has absolute timing constraints, but also a relative constraint with label *grab*. In the navigation control example, relative timing constraints are imposed on the execution of the block in the thread `control` relative to an earlier execution point by reference to the label `read_stat`. Such flexibility is lacking in most current real-time languages and constraint tools. While only `for` loops are supported by Real-Time Euclid, `while` loops with variable iterations are permitted in CRL; however, there must be a compile-time provable or user-asserted upper bound on iteration to assure that it can be analyzed. In the assembly line example, maximumm iterations of the `while` loop is 2.

APPENDIX B: ADDITIONAL CRL BNF

In this appendix, we list partial BNF for many of the CRL elements, which were either omitted for brevity or presented without their BNF descriptions.

B1. Programs, declarations, classes, objects, visibility

```
<program>    ::= <decls>
<decls>      ::= [<type_decls>][<const_decls>]
                 [<var_decls>]
<type_decl>  ::= types <type_decl> {.<type_decl>}*
                 endtypes
<type_decl>  ::= <type><type_name>
<type>       ::= <scalar_type>|<range_type>
                 |<array_type>|<record_type>
                 | <class_type>|<type_name>
<scalar_type>::= integer|rational|boolean|character
<array_type> ::= array <range_type> of <type> endarray
<range_type> ::= <manifest_integer>..
                 <manifest_integer>
<record_type> ::= record<var_decls>endrecord
<const_decls> ::=consts<const_decl>{,<const_decl>}*
                  endconsts
<const_decl> ::= <scalar><const_name>
                 |<type_name><manifest_rhs>
                 <const_name)
<var_decls> ::= vars<var_decl>{,<var_decl>}*endvars
<var_decl> ::= <scalar_type>[<manifest_rhs>]
                 <var_name> | <type_name>
                 [<manifest_rhs>]<var_name>
                 |<object_decl>
<object_decl> ::= <class_type_name><object_name>
                  <constructor_param list>
<constructor_param_list> ::= [in <parameters>]
<class_type> ::= <class_definition>
                 |<class_implementation>
<class_definition> ::= classdefinition <name>
                 <class_interface>
                 endclassdefinition <name>
<class_implementation>::=classimplementation <name>
      [<class_interface>]<decls>[<constructor>]
      [<destructor>]{<method>}*
      endclassimplementation <name>
```

Naturally, class definitions and implementations must correspond (through <name>) to each other, as in other class-based languages.

```
<class_interface> ::= [export<export_list>][import
           {<class_or_type_name>|
           <class_name>.<method_or_type_name>}+]
<export_list> ::= <type_list>{<method_interface>}*
           |{<method_interface>}+
<method_interface> ::= <regular_method_interface>
           |<thread_interface>
<type_list> ::= <type_name>{,<type_name>}*
<regular_method_interface>::= methodinterface<name>
           [<time_constraint>]<param_list>
           endmethodinterface <name>
<param_list> ::= [in <parameters>][out<parameters>]
           [inout<parameters>}
<thread_interface> ::= threadinterface <name>
           <in_param_list>endthreadinterface<name>
<in_param_list> ::= [in <parameters>]
<parameters>::=<parameter_decl>{,<parameter_decl>}*
<parameter_decl> ::= <type><parameter_name>
```

B2. Types, variables, scalars and names

A manifest integer is:

```
<manifest_integer> ::= <integer>|<const_name>
```

where the latter possibility is restricted to integer constants naturally.

Various names as used throughout have the same form:

```
<object_name>,<event_name>,<method_or_type_name>,
<variable>,<type_name>,<parameter_name>,
<const_name>, <label> ::= <name>
<name>::= <alphabetic_char>{<alphanumeric_char>}*
```

There are (at this time) two distinct name spaces, one for labels, and one for all other names.

Miscellaneous definitions are as follows:

```
<scalar> ::= <integer>|<rational>
           |<boolean>|<character>
<manifest_rhs> ::= <rhs>
/*a static semantic error if <rhs> is not a compile-
time constant*/
<rhs> ::= <expr>
<var_ref> ::= <var_name>{.<var_name>}*
<var_name> ::= <name>[{(<index_into_array>)}*]
<index_into_array> ::= <manifest_rhs>
/*static semantic analysis must show it is an
integer constant*/
```

```
<integer> ::= [+|-]<non_zero_digit>{<digit>}*
<non_zero_digit> ::= 1|2|3|4|5|6|7|8|9
<digit> ::= 0|<non_zero_digit>
<rational> ::= [+|-]0[{<digit>}*e[+|-]{<digit>}*]
<boolean> ::= true|false
<alphabetic_char> ::= A|B|C|D|E|F|G|H|I|J|K|L|M
                      |N|O|P|Q|R|S|T|V|W|X|Y|Z
                      |a|b|c|d|e|f|g|h|i|j|k|l|m
                      |n|o|p|q|r|s|t|u|v|w|x|y|z
<alphanumeric_char> ::= <alphabetic_char>|<digit>
<character> ::= <alphanumeric_char>
  |‘|~|!|@|#|$|%|^|&|*|(|)|-|_|-|=|+|||
  |[|{|]|}|;|:|’|’’|,|<|.|>|/|?
```

Currently, all record and method references are fully qualified.

B3. Expressions

The syntax of expressions is given below. The grammar should be disambiguated by the usual rules of associativity and precedence.

```
<boolean_expr> ::= <boolean_expr> xor <boolean_expr>
            |<boolean_expr> or <boolean_expr>
            |<boolean_expr> and <boolean_expr>
            |not <boolean_expr>
            |<expr> <rel_op> <expr>
            |<var_ref>|<boolean>
/*a static semantic error for <var_ref> not to have
type boolean*/
<expr> ::= <expr><op2><expr>
            |<op1><expr>
            |<expr><opr>
            |<var_ref>|<scalar>|<const_ref>
<length_assertion> ::= assert length <manifest_rhs>
            |?assert length
```