Formal Specification and Analysis of Timing Properties in Software Systems

Musab AlTurki[1], Dinakar Dhurjati[2], Dachuan Yu[2], Ajay Chander[2], and Hiroshi Inamura[2]

[1] University of Illinois at Urbana-Champaign, Urbana IL 61801, USA
alturki@illinois.edu
[2] DOCOMO USA Labs, Palo Alto CA 94304, USA
{Dhurjati,Yu,Chander,Inamura}@docomolabs-usa.com

Abstract. Specifying and analyzing timing properties is a critical but error-prone aspect of developing many modern software systems. In this paper, we propose a new specification language and analysis framework for expressing and analyzing timing behaviors of complex software systems. Our framework has the following significant benefits: a) it is expressive, b) it supports trace analysis and simulation of timing behaviors, c) allows for verification of properties of specification, and d) checks for common usage errors of timing constructs. The language constructs for timing were chosen to be very flexible, suitable for expressing different kinds of timing behaviors, and are inspired from timing constructs used in previous languages like SDL. We define the formal semantics of our language using a real-time rewrite theory. Since real-time rewrite theories are executable in Real-Time Maude, our framework supports trace analysis and simulation of timing behavior for specifications. Furthermore, the timed model checker for Real-Time Maude can be readily used for analyzing and verifying various real-time properties of the specifications. Finally, to prevent misuses of timing constructs that can be made possible due to their flexibility, we develop abstract interpretation based static analysis tools that check for common usage errors. We believe that our framework, with the above benefits, provides a significant step forward in facilitating the use of formal tools for specification and analysis of timing behaviors in software development.

1 Introduction

Due to increasing complexity of modern software systems, the likelihood of making errors during the software design phase has increased exponentially. While some of these errors might be detected during the testing phase, it is much more cost effective to detect and correct these errors early during the design phase. For this reason, formal specification and analysis tools are increasingly being deployed to improve the quality of software design.

Many real-world software systems rely on components that have timing requirements to be met. These may represent maximal timing constraints, such as timeouts, minimal timing constraints, such as delays, or durational constraints,

M. Chechik and M. Wirsing (Eds.): FASE 2009, LNCS 5503, pp. 262–277, 2009.

which combine both maximal and minimal constraints. Consequently, correctness of such software systems depends not only on their functional requirements but also on the non-functional timing requirements. Therefore, to be able to formally reason about such requirements, methods and tools for the specification and analysis of real-time requirements need to be developed.

There have been several attempts at developing formal analysis and verification tools for timing properties in software specifications (see [1] and the references there) but there is a gap between the languages used by these tools and what the current specification languages provide, making it hard to integrate them into current design activities in the software development industry. Most of these tools are based on timed formalisms, such as timed automata [2] and timed Petri Nets [3], that typically sacrifice expressiveness for decidability. While they provide efficient formal analysis and verification tools, such timed formalisms are typically difficult to understand and use by the software specification writer, which further limits their applicability in industry. Furthermore, timing constructs in existing high-level specification languages are either restrictive (e.g. Erlang [4]) or flexible but at the cost of allowing many misuses while not providing effective mechanisms to detect them (e.g. SDL [5]).

In this paper, we propose a simple but powerful specification language for expressing timing properties together with an integrated analysis framework that makes available a suite of formal analysis tools for software designers. The language constructs for timing were chosen to be very flexible, suitable for expressing different kinds of timing behaviors, and are inspired from timing constructs used in previous languages like SDL. Due to this expressiveness, timing constructs used in other high level specification languages like SDL and UML can be easily translated into constructs of our specification language. We define the formal semantics of our language with rewrite rules in a real-time rewrite theory [6]. Since real-time rewrite theories are executable in Real-Time Maude [7] under few reasonable assumptions, our framework automatically supports trace analysis and simulation of timing behavior for specifications. Furthermore, the timed model checker for Real-Time Maude can be readily used for analyzing and verifying various real-time properties of the specifications. Thus the integrated analysis framework facilitates the use of formal specification tools by reducing the gap between the specification language and the language used by the verification tools. Finally, since the timing constructs are intended to be very flexible, there is a possibility of misusing the constructs. To prevent such misuse, we develop abstract interpretation based static analysis tools that check for common usage errors.

The main benefits of our framework can be summarized as follows: (1) It is an expressive framework that is capable of formally capturing software specifications given in various specification languages; (2) it supports trace analysis and simulation of timing behaviors; (3) it allows for verification of complex properties of specifications; and (4) it can automatically check for common usage errors of timing constructs. We believe that our framework, with the above

benefits, provides a significant step forward in facilitating the use of formal tools for specification and analysis of timing behaviors in software development.

The rest of the paper is organized as follows. In Section 2 we present our specification language, $\mathcal{L}$ and its semantics. This is followed in Section 3 with a description of a prototype implementation of the language using Real-Time Maude. In Section 4 we describe how the timing abstractions can be misused, and then in Section 5 we describe our abstract interpretation based solution to detect and prevent such misuses. In Section 6 we compare our approach to related work in the area. Finally, we conclude in Section 7 with a summary of our approach.

2 The Specification Language $\mathcal{L}$

In this section, we introduce a high-level specification language $\mathcal{L}$ that is well-suited for describing a spectrum of behaviors of various software systems, including their timed behaviors. $\mathcal{L}$ is a simple, concurrent specification language that is aimed to serve as a formal programming model for various user-level specification languages, such as SDL and UML. The language is intended to provide a unified, well-established specification framework for the analysis and verification of such higher-level specifications. Beside providing a core language with formal semantics for specification creation, management and analysis, the simplicity of $\mathcal{L}$ directly translates into a simple formal model that can easily be analyzed and manipulated.

While the language supports several imperative features for describing sequential computations, concurrency in $\mathcal{L}$ is modeled by asynchronously communicating processes that can be dynamically created or destroyed. A process maintains a thread of sequential computation representing a simple component in software. A process may create another process with a specified computational behavior, or may destroy itself. Processes communicate by exchanging messages asynchronously, and use timers as the basic timing abstraction to account for timing behaviors. The syntax and semantics of $\mathcal{L}$ are described next.

2.1 Syntax and Examples

The syntax of $\mathcal{L}$ is shown in Figure 1. A constant expression in $\mathcal{L}$ can be either an integer value, a boolean value, a literal string, or a variable name. Complex expressions can be constructed using standard arithmetic, relational, and boolean composition operators.

Unlike expressions which evaluate to some constant value, commands do not produce values, but are there for their side effects. A command in $\mathcal{L}$ can be an assignment statement, a scoped declaration of a variable using a **let** statement, a conditional statement, or a while loop statement. The language also has a few process-level commands, which include creating a new process, destroying the current process, sending a message to a process, and receiving a message. The body of the receive statement may consist of a list of exclusive case

$$x, y \in \text{Variable} \qquad n \in \text{Integer} \qquad r \in \text{String}$$

$$\begin{aligned}
e \in \text{Expression} &::= x \mid r \mid a \mid b \\
a \in \text{Arithmetic Expression} &::= n \mid a \circ a \\
b \in \text{Boolean Expression} &::= \mathit{true} \mid \mathit{false} \mid a \bullet_{rel} a \mid b \bullet_{bool} b \mid \neg b \\
c \in \text{Command} &::= x := e \mid \textbf{let } x = e \textbf{ in } c \mid \textbf{if } b \textbf{ then } c \textbf{ else } c \mid \textbf{while } b\ c \\
&\quad \mid \textbf{new } x = y \textbf{ in } c \mid \textbf{destroy} \\
&\quad \mid \textbf{send } e \textbf{ to } e \mid \textbf{receive } x \textbf{ in } \{l\ ;\ \textbf{default} : c\} \\
&\quad \mid \textbf{set } x \textbf{ to } e \mid \textbf{release } x \mid \{\} \mid \{c\} \mid c\ ;\ c \\
l \in \text{CaseList} &::= \epsilon \mid \textbf{case } e : \ c\ ;\ l \\
m \in \text{Module} &::= \textbf{module } x \textbf{ is } c
\end{aligned}$$

Fig. 1. The abstract syntax of $\mathcal{L}$

statements followed by a default statement. Timers are managed using two constructs: **set**, for starting a timer, and **release**, for dropping a timer. The expiration of a timer in a process triggers a signal that can be checked by a **receive** command. Furthermore, commands can be grouped into command blocks, and sequenced using the semicolon as a sequencing operator. Finally, a specification in $\mathcal{L}$ may use an optional list of module declarations, serving as templates for new processes.

A variable x is bound in c in the commands **let** $x = e$ **in** c and **new** $x = y$ **in** c, and is bound in l and c of the **receive** command. Variables used in **set** and **release**, called *timer variables*, are globally scoped variables and are assumed to be distinct in a given specification for it to be meaningful. A variable is said to be free if it is not bound.

For compactness, we use **if** b **then** c as syntactic sugar for a conditional with an empty *else* branch, and **receive** x **in** c to denote a receive statement with no *case* branches, i.e. **receive** x **in** $\{\epsilon$; **default** : $c\}$. We shall also use **let** $x_1 = e_1, x_2 = e_2$ **in** c as a shorthand for two nested **let** commands.

Example. (Client) The specification shown in Figure 2 defines a client process that interacts with the user and a server, and timeouts responses from the server, for which it maintains two timers t_1 and t_2. Upon receiving a "resend" request from the user, the process forwards a request to the server and sets a timeout of 60 time units for the first response from the server using the timer variable t_1. There are three possibilities at this point; (1) a timeout occurs, which is indicated by the incoming signal t_1, and at which case the process restarts and waits for another request from the user; (2) another request from the user is received, at which case the client resets the timer t_1, and sends a request to the server; (3) a response from the server is received, at which case t_1 is dropped and a similar process is initiated for subsequent responses from the server using another timer t_2. For simplicity, the example does not specify how the client actually processes incoming responses from the server.

We will refer to the Client example above in the rest of the paper to illustrate various aspects of the specification framework. Below, we give a more precise description of the semantics of $\mathcal{L}$.

```
module client is
    let a = true, b = true in
    receive o in                                  — server id
    while (true)
        receive p in  {                           — resend request from user
            a := true;  b := true;
            while (a) {
                set t1 to 60;
                send "request" to o;
                receive m in {
                    case t1 :    a := false;
                    case "resend" :    release t1;
                    default :  {
                        release t1;
                        while (b) {
                            set t2 to 120;
                            receive m in {
                                case t2 : {    b := false;  a := false    };
                                case "resend" : {    release t2;  b := false    };
                                default :  {    release t2    } } } } } } }
```

Fig. 2. A specification in $\mathcal{L}$ for a client process with timeouts

2.2 Formal Semantics

We give the formal semantics of $\mathcal{L}$ as an object-oriented real-time rewrite theory $\mathcal{R}_{\mathcal{L}}$. The semantics is distributed and concurrent in that a state for a specification in $\mathcal{L}$ consists of one or more process objects that are executed concurrently, and which may interact with each other as time elapses.

Real-Time Rewrite Theories. A rewrite theory, a unit of specification in rewriting logic [8], gives a formal description of a concurrent system including its static state structure and dynamic behavior. A rewrite theory is a tuple $\mathcal{R} = (\Sigma, E, R)$, with

- (Σ, E) a membership equational logic [9] theory with signature Σ and a set of universally quantified equations and/or memberships E. The signature Σ declares the sorts and operators to be used in the system specification, while equations and memberships E algebraically specify the properties satisfied by these operators.
- R a set of universally quantified, possibly conditional, rewrite rules specifying the computational behavior of the system. A rewrite rule has the form:

$$r : t_1 \longrightarrow t_2 \text{ if } C \tag{1}$$

where r is a label, t_1 and t_2 are terms over Σ, and C is a conjunction of equational or rewrite conditions. A rewrite rule gives a general pattern for a possible change in the state of a concurrent system (See [10] for a detailed account of generalized rewrite theories).

A real-time rewrite theory [6] extends a regular rewrite theory with support for modeling temporal behaviors of systems. In particular, in a real-time rewrite theory $\mathcal{R}^\tau = (\Sigma^\tau, E^\tau, R^\tau)$: (i) the equational theory (Σ^τ, E^τ) contains a sort for *Time* representing the time domain, which can be either dense or discrete, and declares a system-wide operator that encapsulates the whole system being modeled into a special sort *GlobalSystem* for managing time elapse, and (ii) the set of rewrite rules R^τ is the disjoint union of two sets R_I and R_T, where R_I consists of *instantaneous* rewrite rules having the form (1) above and representing instantaneous transitions in the system, and R_T consists of *tick* rewrite rules modeling system transitions that take non-zero amount of time to complete. A tick rewrite rule has the following form

$$r : \{t_1\} \xrightarrow{\tau} \{t_2\} \text{ if } C$$

where τ is a term of sort *Time* representing the duration of time required to complete the transition specified by the the rule. The global operator $\{\_\}$ encapsulates the whole system into the sort *GlobalSystem* to ensure the correct propagation of the effects of time elapse to every part of the system.

Semantic Infrastructure. We fix a sort V of values in $\mathcal{L}$. Lists of values can be constructed as fully associative lists of comma-separated values. An environment σ is a mapping from variable names to values, specified in $\mathcal{R}_\mathcal{L}$ as an associative list of entries of the form $[x, v]$ with identity *nil*.

A state in the system is represented by a *configuration* consisting of a multi-set of objects. The fundamental class of objects within a configuration is the *Process* class. In addition to the process object identifier, a process object contains the following fields: a name, an environment, a command, a field for the timer set of the process, and a queue of incoming messages:

$$\langle id : Process \mid name : x, env : \sigma, cmd : c, tmr : T, msg : M \rangle$$

The queue of messages M is simply a list of values, and T is a set of timer records of the form $\{x, v_t\}$, with v_t a time value. A timer record in T represents an *active* timer, which is a timer that has been started but is not yet expired or handled.

Instantaneous Transition Rules. In $\mathcal{R}_\mathcal{L}$, instantaneous transitions of $\mathcal{L}$ are modeled by regular rewrite rules, which specify the behavior of a process within a configuration based on the next command to be executed by that process. The command field *cmd* of a process serves as a continuation that defines what action to be taken next. For example, the rule labeled set below specifies the semantics for setting a timer:

$$[\mathsf{set}] : \langle id : Process \mid env : \sigma, cmd : \mathbf{set}\ x\ \mathbf{to}\ a\ ;\ c', tmr : T \rangle \\ \longrightarrow \langle id : Process \mid env : \sigma, cmd : c', tmr : \{x, a \downarrow_\sigma\}, T \rangle$$

where $e \downarrow_\sigma$ denotes the evaluation of e using the environment σ, while expiration of a timer is captured by the timeout rule below:

$$[\mathsf{timeout}] : \langle id : Process \mid env : \sigma, cmd : \mathbf{receive}\ x\ \mathbf{in}\ C\ ;\ c', tmr : \{y, 0\}, T\rangle \\ \longrightarrow \langle id : Process \mid env : \sigma[x, y], cmd : cases(x, C);\ pop;\ c', tmr : T\rangle$$

with $cases(x, C)$ and pop as auxiliary continuation items for processing the body of a receive statement. We note that the rules are given in the object-oriented specification style, in which attributes within a process object that do not play a role in the rule need not be mentioned. We assume that message exchanges are instantaneous (take no time to complete) and are therefore modeled by instantaneous rewrite rules.

2.3 Timed Semantics

Assuming R is a time value and C is a configuration, the *tick* rule in $\mathcal{L}$ that models time elapse and its effects is as follows:

$$[\mathsf{tick}] : \{C\} \xrightarrow{R} \{\delta(C, R)\} \quad \text{if} \quad R \leq mte(C) \wedge inactive(C)$$

There are several important observations to be made here:

- The function δ equationally propagates the effect of a time tick to all objects within the configuration C, which comprises decreasing all timer values within all process objects by the amount R of the tick.
- The function *mte* equationally defines the *maximum time elapse* until the next event of interest. This is a standard technique in RTM to specify upper bounds on how much a clock is allowed to advance before the next event in the configuration. In this case, the *mte* of a configuration of processes is determined by the timer with the minimum time value among all sets of timers in all processes:

$$mte(T, \{x, v_t\}) = min(mte(T), v_t), \qquad mte(\phi) = \infty$$

- The predicate *inactive* distinguishes states in which instantaneous (untimed) transitions are enabled (also called *active states*) from those in which the only possible transition is a tick transition advancing time (*inactive states*). The predicate is used to restrict applications of the tick rule to inactive states so that instantaneous transitions have precedence over time tick transitions. This is to maintain the expected semantics of timers and to prune uninteresting behaviors in which a configuration might appear to be progressing while it is not (for example, advancing time without doing anything else). This semantics enforces the fact that when a timer in a process expires, its signal cannot be ignored and must be handled, either by releasing the timer or by consuming its signal. For this semantics to be fully meaningful, however, configurations may only assume *non-Zeno* behaviors (which are behaviors in which time will always eventually have a chance to advance), which is a common assumption for real-time specifications with logical time.

3 Analysis of $\mathcal{L}$ Specifications in Real-Time Maude

Real-Time Maude (RTM) [7], which is based on Maude [11], provides a highly efficient implementation of real-time rewrite theories. We have developed a prototype in RTM for $\mathcal{L}$ that corresponds to the specification $\mathcal{R}_{\mathcal{L}}$ described above. As an immediate consequence of specifying the formal semantics of $\mathcal{L}$ in RTM, we obtain a simulator and several formal analysis tools essentially for free. Among the analysis tools provided are: (1) the timed fair rewrite `tfrew`, which simulates one possible behavior (a sequence of rewrite steps) of a specification up to a given time bound; (2) the `tsearch` command, which performs timed breadth-first search on the reachable state space from given an initial state, while looking for a state matching a given term and satisfying a given semantic condition; and (3) the timed model-checking command `mc T |=t F in time <= R`, which checks for satisfiability of the linear temporal logic (LTL) formula `F` along paths starting from the initial state `T` within the time bound `R`.

The prototype is specified as a *real-time object-oriented module* $\mathcal{M}_{\mathcal{L}}$ in RTM, which is declared using the syntax `tomod` *Name* ... `tomend`. To simplify analysis, we assume a discrete time domain, implemented using the domain of natural numbers extended with infinity, which can be specified by letting the module $\mathcal{M}_{\mathcal{L}}$ extend RTM's predefined module `NAT-TIME-DOMAIN-WITH-INF`.

We consider the CLIENT specifications given in Figure 2 above to illustrate the use of such formal tools. We use `client` to denote its specification in $\mathcal{M}_{\mathcal{L}}$. Since `client` is a template for a reactive process that communicates with a user and a backend server, we assume an initial configuration `system` in which a user object and a server object are defined in order to be able to perform analysis on `client`. The initial configuration contains a server process object that upon receiving a request sends out five responses, five time units apart, and a user process object that sends two "resend" requests, the first at time 1, and the other at time 20. To simplify the presentation of the analysis, another object, called the *Observer* object, is used to record traces of events of interest along with their time stamps.

3.1 Simulation and Prototyping

A sample run of `system` for a duration of 200 time units can be obtained by issuing the following command (where some of the output is omitted for brevity):

```
 Maude> (tfrew
system in time <= 200 .) Result ClockedSystem : {...
 < oo : Obser | out :([6 : "t1: first response received"]
[11 : "t2: response received"] [16 : "t2: response received"]
[21 : "t2: resend before it expires"] [21 : "t1: first response received"]
[26 : "t2: response received"] [31 : "t2: response received"]
[36 : "t2: response received"] [41 : "t2: response received"]
[161 : "t2: expired"]) >
 < p(1): Process | name: 'client, cmd: receive p in ... , tmr: empty >
 < p(2): Process | name: 'server, cmd: receive m in ... , tmr: empty >
 < p(3): Process | name: 'user, cmd: {}, ... , tmr: empty >} in time 200
```

The result above shows that after 200 clock ticks, the system reaches a quiescent state where no more message exchanges exist or are scheduled, and no timers are yet to be set or processed. As can be seen from the recorded trace, a "resend" request from the user was received at time 21 while the client was processing the third response from the server, immediately after which the client resent the request and restarted processing. Since the server sends only five responses to a given request, we see the timeout at time 161 after the fifth response had been received at time 41.

Furthermore, using timed search, one can verify, starting from `system`, the property that the system will in fact never be in a quiescent state before that.

```
Maude> (tsearch system =>+ { CF:Configuration } such that
          inactive({CF:Configuration}) and noAliveTimer(CF:Configuration)
          in time <= 160 .)
rewrites: 217595 in 720ms cpu (720ms real) (301842 rewrites/second)
No solution
```

The arrow `=>+` means states reachable by one or more rewrites from the given state. The semantic condition `inactive(CF) and noAliveTimer(CF)` captures exactly what it means for a state to be quiescent.

3.2 Model Checking Analysis

RTM also provides powerful time-bounded model-checking tools for verifying general LTL formulas, representing both liveness and safety properties, which can be immediately applied to specifications in $\mathcal{L}$. The LTL formulas are based on a set of atomic propositions that capture state properties of interest and a labeling function that assigns to each state in the system a subset of atomic propositions that are true in that state. Given a module `M` for some specification in $\mathcal{L}$, this is done in RTM by defining a module `M'` that imports the module `M` and the internal module `TIMED-MODEL-CHECKER` and specifies equationally the meanings of these propositions and the labeling function. For our running example, `client`, we would perform model checking against a module extension of the form:

```
(tomod MODEL-CHECK-CLIENT is
  including TIMED-MODEL-CHECKER .
  protecting CLIENT .
  ...
 endtom)
```

where `including` and `protecting` represent module extension modes (see [11]). The internal module `TIMED-MODEL-CHECKER` declares sorts for states `State`, atomic propositions `Prop`, logical formulas `Formula` to which the various LTL operators belong, and the logical time-bounded satisfaction operator `|=t`, among several other things. Thus, within the module above, one can declare the following two propositions (the keywords `ops`, `var`, and `eq` introduce, respectively, operator declarations, variable declarations, and equations in Maude):

```
ops first-response timeout : -> Prop .
var CF : Configuration . var O1 O2 : Output . var R : Time .
eq {CF < oo : Obser | out :(O1 [R : "t1: first response received"]
      O2) > } |= first-response = true .
eq {CF < oo : Obser | out :(O1 [R : "t2: expired"] O2) > } |=
      timeout = true .
```

The first proposition `first-response` is true in a state in which the client has already received its first response from the server, while the other proposition `timeout` is true in a state where the second timer has expired. States in which a proposition does not hold need not be specified.

Using these proposition, we can verify a fairly complex property about the system modeled by `client`: it is always the case that within the first 200 time units and after receiving the first response from the server, the second timer will eventually expire. This property holds since the server will cease to send out responses after the first response, causing the client to eventually timeout. This can be checked automatically using the model-checking command:

```
Maude> (mc system |=t [] (first-response -> <> timeout) in time <= 200 .)
rewrites: 164943 in 689ms cpu (693ms real) (239084 rewrites/second)
Result Bool : true
```

where `[]` denotes "always", `->` "implication", and `<>` the "eventually" operator. However, the property does not hold if we restrict traces to 100 time units. The corresponding model-checking command presents a counter example trace to that effect:

```
Maude> (mc system |=t [] (first-response -> <> timeout) in time <= 100 .)
rewrites: 35567 in 4332ms cpu (4345ms real) (8209 rewrites/second)
Result ModelCheckResult :
  counterexample({{< od : Decls | dcl :(( module 'client is let a = true
  ...
  [self,vpid(3)],msg : nil,name : 'user,tmr : empty >} in time 41,'tick})
```

4 Proper Use of Timing Abstractions

In order to be able to model a wide range of software systems with real-time components, the timing abstractions of $\mathcal{L}$ are designed so that they are expressive and flexible. However, such flexibility might enable unintended or undesirable usage patterns of these abstractions. We overview in this section possible usage problems with timers and discuss automatic means to detect them.

We consider again our working example specification Client shown in Figure 2. There are several possible misuses of timer-related constructs in Client which would render the specification erroneous or unnecessarily complex. For instance, by dropping any one of the release commands in this specification or by dropping any one of the receive case branch statements, we introduce possible execution paths along which a timer is set but never released or processed. Moreover, by adding any new release command or case statement to this specification,

we essentially introduce dead code that is either superfluous or even unreachable along any execution path in the specification. We note that such problems become more pronounced as specifications get larger and more complex.

The fundamental reason behind such potential problems is flexibility. Indeed, timers of a process are globally scoped within that process. Furthermore, the *set* and *release* statements are not tightly coupled together, which implies that complex timer patterns are possible. Finally, the unified treatment of timer signals and incoming messages in **receive** statements might also add to the conceptual complexity of properly using timers. It is worth mentioning that most of these characteristics are shared with timer-based specification languages such as SDL, making these languages, too, vulnerable to mishandled timers.

The problem, which we call *Mishandled Timers*, identifies usage patterns of timers that could potentially cause semantic or structural problems with specifications in $\mathcal{L}$. It consists of three sub-problems:

1. *Unhandled timers:* a timer is not properly handled in a specification if there exists a possible execution path along which a timer is set but then neither dropped nor its signal is ever consumed.
2. *Extra release commands:* a **release** command is extra if it attempts to drop a timer that is always properly handled along all execution paths to it.
3. *Unreachable case branches:* a **receive** case branch is unreachable if the timer whose signal is being checked is always properly handled along all execution paths to that case branch.

The significance of such analysis revolves around both specification correctness and optimization. Unhandled timers immediately indicate a problem in the specification, since the meaning of an unhandled timer is not clear. Both extra release commands and unreachable case branches might also be the result of an accidentally missed **set** command and can therefore change the intended semantics. In the case that no **set** command was missed, such superfluous statements can as well be eliminated to optimize the specification.

Fortunately, the mishandled timers problem can be formulated as a data-flow analysis problem, and can therefore be checked automatically using standard static analysis means. Instead of defining a static checker that is specific to the mishandled timers problem, we develop a general static analysis framework to be integrated with the specification language $\mathcal{L}$ so that different other static analyses can be easily specified and used. We describe below the static analysis framework and its instantiation to the mishandled timers problem.

5 Static Analysis of Specifications in $\mathcal{L}$

The formal analysis tools and techniques provided by RTM and described above are very useful for analyzing specifications in $\mathcal{L}$ and verifying properties about them. However, due to the dynamic nature of the analysis, such properties are necessarily specific to the specification in hand, and an initial state must be constructed for them to be carried out. For example, for Client, the property

that the system will never be in a quiescent state before 160 time units have passed applies only to this specification and was verified against one possible initial state defined by `system`.

Another class of formal verification techniques with which generic properties can be automatically verified can be obtained through static analysis. Static analysis is an automated formal analysis technique that is based on the static structure of specifications rather than their dynamic behavior. The analysis allows for the verification of a different class of properties dealing with the proper use of constructs in $\mathcal{L}$. These properties are generic in the sense that they are not tied to any particular specification and do not depend on any given initial state. As a result, a library of static analysis properties can be built and reused to check specifications in $\mathcal{L}$ for common bugs or to perform common optimizations, which considerably increases the usefulness and effectiveness of $\mathcal{L}$ and its associated tools as a software specification framework.

5.1 A Generic Abstract Interpretation Framework

The approach to static analysis we use is based on the well-studied framework of Abstract Interpretation [12], which enables building safe approximations of a given concrete semantics, so that if a property holds in the abstract semantics, it also holds in the concrete semantics. Specifically, we use control flow graphs (CFGs) to build such abstract interpretations. A CFG for a specification S consists of a set of nodes, representing commands (or basic blocks) in S, and a set of directed edges, representing possible immediate flows between commands.

We have specified our abstraction framework for $\mathcal{L}$ as an equational theory and implemented it in Maude as a functional module. The module defines an operator `cfg`, which, given a specification in $\mathcal{L}$, builds a flattened graph as a set of nodes and directed edges grouped together using the associative and commutative empty juxtaposition operator. A node in a CFG is a pair `< I : B >`, consisting of an identifier `I` and a statement `B` corresponding to the command represented by that node, while a directed edge is a triple `[ I1 : S : I2 ]`, consisting of identifiers `I1` and `I2` for the source and target nodes, respectively, and an abstract state `S` on that edge, which is used for analysis. The CFG construction process is defined inductively over the structure of commands in $\mathcal{L}$, and computation of fixed points is specified by straight-forward equations that are mostly facilitated by Maude's efficient associative-commutative matching algorithms on the flattened graph. For instance, the following equation specifies the effect of the assignment command (`ceq` introduces a conditional equation):

```
ceq [I1 : S : I] < I : x := e > [I : S'  : I2]
 =  [I1 : S : I] < I : x := e > [I : S'' : I2]
    if S'' := assign(S, x, e) /\ S' < S'' .
```

where `assign(S, x, e)` is the transfer function for assignment and `<` is the strict partial ordering relation on abstract states. The particular definitions of transfer functions, abstract states, and the ordering relation are dependent on the specific property to be analyzed and are therefore left unspecified in the

abstraction framework. Below, we give an instantiation of it for the analysis of the mishandled timers problem.

5.2 Mishandled Timers

We formulate the mishandled timers problem as a data-flow analysis problem, and then use the abstract interpretation framework described above to automatically check for it. The analysis computes, at each point in a specification, the set of timers that may have not been properly handled on some path to that point in the specification.

By computing such intermediate states, we can build decision procedures to detect misuses of timers as follows. We first define the abstract domain to be a simple lattice $\{\top, \bot\}$ with the usual ordering. A timer variable is mapped to $\top$ in an abstract state if it references a timer that may not have been handled in that state. Otherwise, it is mapped to $\bot$. The abstract state is a valuation from timer variable names to values in the lattice. Both the lattice ordering and the join operation are extended in the usual way to abstract states.

Then, we define, for each command in $\mathcal{L}$, the transfer function that specifies the effect of that command on the abstract state. Most of these functions are fairly trivial to define for this problem since most functions are the identity function on states, except for the commands **set** and **release**, for which the transfer functions respectively map variable names to $\top$ and to $\bot$. Furthermore, the transfer function for the conditional command is defined to reflect the possible change in state in the true and false branches of several other commands, such as receive case statements. Finally, we define the following operators that will automatically check the three problems : (1) `utimers` for unhandled timers, (2) `ers` for extra release commands, and (3) `ecs` for unreachable case branches.

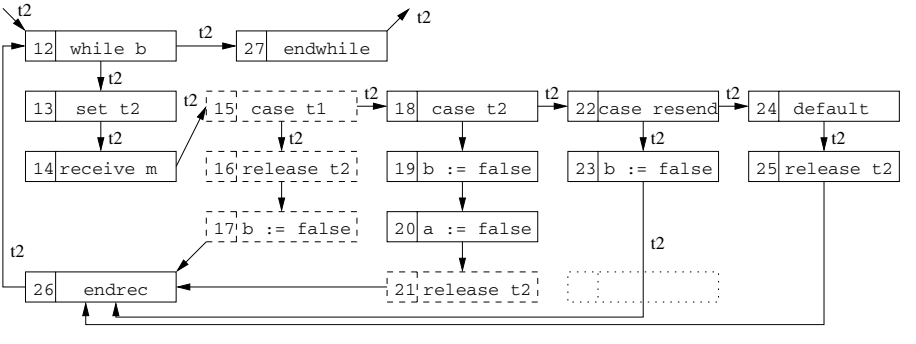

Fig. 3. A partial CFG computed by the mishandled timers analysis for the innermost while loop of a modified (buggy) version of the `client` example; dashed nodes are the ones added, while the dotted node is a `release t2` node that was removed . Only timer variables mapped to $\top$ in abstract states are shown on edges.

To illustrate the use of these operators, we apply them to an instrumented version of the `client` specification, named `BuggyClient`, to which we introduced some instances of the mishandled timers problem. The CFG for the innermost **while** (b) in the modified specification along with internal results of the analysis algorithm are shown in Figure 3. As is clear from the figure, timer `t2` is not properly handled, which can be automatically realized by the command[1]

```
Maude> red utimers(cfg(BuggyClient)) .
rewrites: 47660 in 58ms cpu (59ms real) (807919 rewrites/second)
result State: [t2,top]
```

and which is resulting from a missing `release` statement within the case branch labeled `22`. Moreover, the release command labeled `21` is extraneous, which can be checked by issuing the command:

```
Maude> red ers(cfg(BuggyClient)) .
rewrites: 47675 in 59ms cpu (59ms real) (794702 rewrites/second)
result Node: < 21 : release t2 >
```

Similarly, by executing the command `red ecs(cfg(BuggyClient))`, we can verify unreachability of the case branch labeled `15` in the figure.

6 Related Work

Real-time languages, for which a large body of research exists, differ widely in terms of the timing abstractions they support and their semantics depending mainly on their targeted application domains. The closest languages to our design of timing abstractions are SDL [5,13], a high-level specification language, and Erlang [4], which is a programming language based on the Actors model [14] for distributed, soft real-time systems. Both languages are based on a concurrent process model, and they both use timers and check for timer signals as incoming messages. However, our design has a stricter timers semantics than that of SDL and is much more expressive than Erlang's (some nested timing patterns, which can be expressed in $\mathcal{L}$, are not expressible in Erlang). There has also been some attempts at improving the timing abstractions in SDL for specification writers, such as the work in [15] on extending timers with annotations and supporting transitions with urgencies. Many other timed high-level languages exist [16,17,18].

Real-time rewrite theories and their implementations in Real-Time Maude have been used in the specification and analysis of various protocols and algorithms [19,20,21,22]. Our application is fundamentally different though as it applies these methods to a specification language rather than a protocol or an algorithm, which has subtle consequences in terms of design and analysis.

Finally, the technique of abstract interpretation [12] has been successfully applied over the years to static analysis (see [23] for a recent survey, and [24] on its use for data-flow analysis). In particular, the technique has been applied to validation of timing requirements [25] and for more efficient model checking [26].

[1] The Maude command `red` or `reduce` evaluates the given expression according to the equations and memberships of the current module.

7 Conclusion

In this paper, we presented a new simple specification language with formal semantics that can be used to specify and analyze timing behaviors of software systems. Our specification language is flexible and supports, through translation, the timing models of various other high level specification languages like SDL and UML. Our formal semantics is defined as a real-time rewrite theory. This automatically gives us the ability to perform simulation and trace analysis using the RTM tool. Furthermore, we take advantage of the timed model checker provided with RTM, to provide an integrated analysis framework for software designers. Finally we show how to use traditional abstract interpretation based approaches to detect common misuses of timing constructs, thus automatically preventing some of the common errors that a software designer can make when using the flexible timing constructs. Together, we believe that our approach provides a significant step forward in facilitating the use of formal tools for specification and analysis of timing behaviors in software development.

Acknowledgements. Many thanks to José Meseguer for his comments.

References

1. Wang, F.: Formal verification of timed systems: A survey and perspective. Proceedings of the IEEE 92(8), 1283–1305 (2004)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
3. Merlin, P., Farber, D.: Recoverability of communication protocols - implications of a theoretical study. IEEE Tran. on Comm. 24(9), 1036–1043 (1976)
4. Barklund, J., Virding, R.: Specification of the standard Erlang programming language, Draft version 0.7 (June 1999)
5. ITU-T: Recommendation Z.100(08/02), languages and general software aspects for telecom. systems - specification and description language (SDL) (August 2002)
6. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. Theoretical Computer Science 285, 359–405 (2002)
7. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
8. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. 96(1), 73–155 (1992)
9. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
10. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theor. Comput. Sci. 360(1-3), 386–414 (2006)
11. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977, pp. 238–252. ACM, New York (1977)

13. ITU-T: Recommendation Annex F1(11/00), languages and general software aspects for telecom. systems - SDL formal semantics definition (November 2000)
14. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge (1986)
15. Bozga, M., Graf, S., Mounier, L., Ober, I., Roux, J.L., Vincent, D.: Timed extensions for SDL. In: Reed, R., Reed, J. (eds.) SDL 2001. LNCS, vol. 2078, pp. 223–240. Springer, Heidelberg (2001)
16. Tardieu, O.: A deterministic logical semantics for Pure Esterel. ACM Trans. Program. 29(2), 8 (2007)
17. Taft, S.T., Duff, R.A., Brukardt, R.L., Ploedereder, E., Leroy, P.: Ada 2005 Reference Manual. LNCS, vol. 4348. Springer, Heidelberg (2006)
18. Bollella, G., Gosling, J.: The real-time specification for Java. Computer 33(6), 47–54 (2000)
19. Ölveczky, P.C., Meseguer, J., Talcott, C.L.: Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. Formal Methods in System Design 29(3), 253–293 (2006)
20. Ölveczky, P.C., Prabhakar, P., Liu, X.: Formal modeling and analysis of real-time resource-sharing protocols in Real-Time Maude. In: 22nd Int'l Parallel and Distributed Processing Symp. (IPDPS 2008). IEEE Computer Society Press, Los Alamitos (2008)
21. Ölveczky, P.C., Caccamo, M.: Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 357–372. Springer, Heidelberg (2006)
22. Ölveczky, P.C., Grimeland, M.: Formal analysis of time-dependent cryptographic protocols in Real-Time Maude. In: 21st International Parallel and Distributed Processing Symposium (IPDPS 2007). IEEE Computer Society Press, Los Alamitos (2007)
23. Cousot, P.: Abstract interpretation and application to static analysis (invited tutorial). In: First IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering, TASE 2007, Shanghai, China (June 2007)
24. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, 2nd printing edn. Springer, Heidelberg (2005)
25. Wilhelm, R., Wachter, B.: Abstract interpretation with applications to timing validation: Invited tutorial. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 22–36. Springer, Heidelberg (2008)
26. Ioustinova, N., Sidorova, N.: A transformation of SDL specifications - a step towards the verification. In: Bjørner, D., Broy, M., Zamulin, A.V. (eds.) PSI 2001. LNCS, vol. 2244, pp. 64–78. Springer, Heidelberg (2001)