Example Specifications of Non-functional Properties of a Simple Counter Application

Technical Report COMP-006-2008, Computing Department, Lancaster University

Steffen Zschaler
Computing Department
Lancaster University
Lancaster, United Kingdom
szschaler@acm.org

November 3, 2008

1 Introduction

This document lists the complete TLA$^+$ specifications for the main example from [3]. It is meant to serve as an external appendix to that paper in order to improve understanding and provide additional detail that could not be included in the journal-paper version for space reasons. We begin by giving some more TLA$^+$ background before discussing the individual specification modules one after the other.

2 Background

This section provides additional TLA$^+$ background beyond what has been given in [3]. Of course, we cannot go into all detail of TLA$^+$; for this, the interested reader is referred to [1], which is an excellent textbook on the language and logics. Here, we focus on the things necessary for understanding our specifications.

TLA$^+$ specifications are divided into modules. Each module starts with a line containing the MODULE keyword and the name of the module. Modules may contain arbitrarily many horizontal divider lines. These are only used for visual structuring and have no formal semantics. Modules may extend other modules (using the EXTENDS keyword on the first line), importing all definitions of all extended modules. Modules may also instantiate other modules, by using the INSTANCE keyword and mapping all variables and constants of the instantiated module to variables and constants of the instantiating module. Modules may also contain inner modules. These are only available within their containing module and cannot be instantiated from anywhere else. Inner modules can use all definitions of the outer module directly. The outer module can only make use of the definitions within an inner module by instantiating the inner module. In such an instantiation, only the variables and constants defined in the inner module must be mapped. In the specifications below we will use inner modules to allow us to hide helper variables from users of the outer module. The basic pattern is to define all externally visible variables as variables of the outer module and all helper variables

as variables of the inner module. The inner module is then instantiated in the outer module, using existential quantification to provide values for the helper variables.
Understanding a TLA$^+$ module is best done beginning from the end. Typically, the last formulas in a TLA$^+$ module are the ones that are really of interest. Everything before is often defined to help with the definition of these interesting formulas.
Often, a TLA$^+$ module defines a state machine. Such a definition looks like this:

$$\begin{aligned} Spec \quad \triangleq \quad & \land\ Init \\ & \land\ \Box[Next]_{vars} \end{aligned}$$

$Spec$ is the name of the state-machine specification defined. $Init$ and $Next$ refer to a previously defined predicate and a previously defined action. $Init$ describes the possible initial states of the state machine and $Next$ describes what can happen in a step, using a disjunction of individual actions describing individual step alternatives. $vars$ is a collection of all variables relevant for the state machine. Often, this is given directly as a sequence of the relevant variables, written $\langle a, b, c \rangle$.
In definining measurements and other forms of specifications, we will use a form of specification that is very close to aspect-oriented programming. The final formula looks very similar to the definition of a state machine as described above. However, $Next$ is a *conjunction* of alternatives and each alternative is defined as an implication $A \Rightarrow B$, where A is an action describing an alternative step from some base state-machine specification and B is an action that should be executed whenever A is executed. As has been discussed in [3] and in more detail in [2], this form of specification effectively adds B to the base state machine whenever A holds.

3 Specification of Time

The first module defines the notion of *time.* It has been taken and slightly modified from [1]. The main modification is that we have separated safety and liveness parts of the specification so that we can use the safety part of the definition independently. Time is captured by the new variable now.

—————— MODULE $RealTime$ ——————

This is based on the original $RealTime$ specification from the TLA toolkit, but we removed the lifeness part—– that is, $NZ(v)$—– from $RTnow$.

EXTENDS $Reals$

Variables:

now - the current system time.

VARIABLE now

A helper definition

LOCAL $NowNext(v) \triangleq \land\ now' \in \{r \in Real : r > now\}$
$\qquad\qquad\qquad\qquad \land$ UNCHANGED v

$RTnow(v)$ asserts two things: a) time never runs backward, b) steps changing now do not change any other variable in v, and vice versa

$RTnow(v)$ is a safety property, that is, it allows systems in which time stops altogether. This is useful for certain proofs. If one needs to explicitly exclude this possibility, one conjoins $NZ(v)$, which adds the required fairness constraints.

$RTnow(v) \triangleq \land\ now \in Real$

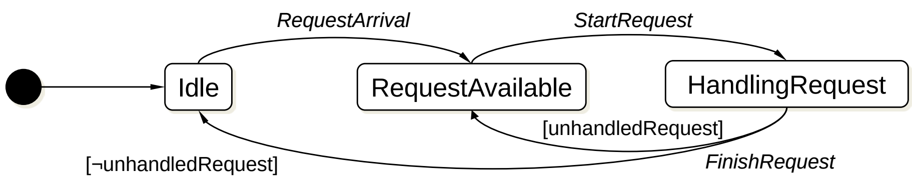

Figure 1: State-machine representation of the service-operation context model

$$\wedge \square[NowNext(v)]_{now}$$

The so called $NonZeno$ condition, which asserts that time will eventually exceed every bound. This liveness constraint is only required under certain circumstances.

$$NZ(v) \triangleq \forall\, r \in Real :$$
$$\quad \text{WF}_{now}(NowNext(v) \wedge (now' > r))$$

4 Context Model Definition

We define two context models: $Service$ defines a service operation and will be the basis for defining the response-time measurement, $Component$ defines a component operation and will be the basis for defining the execution-time measurement. These context models have already been discussed in their state-machine form in the main paper, but here we show the TLA$^+$ specifications.

4.1 A Context Model for Service Operations

Figure 1 gives the state-machine view of this context model again. This has already been shown in [3], but we show it here again to simplify understanding of the formal specification. Notice that the TLA$^+$ specification differs from the graphical rendering in two respects:

1. State information has been divided into two parts: Variable $inState$ captures if the service is currently idle or is handling a request. Additionally, variable $unhandledRequest$ captures if a request is currently waiting to be handled.

2. The specification has been split into an environment specification and a service specification. This has been done to simplify proofs of feasibility further down the line. It can be shown, however, that this form of specification can be transformed to a form that uses only one integrated state machine.

MODULE $Service$

Service Context Model

Variables:

$inState$ – the current state of the service execution machinery.
$unhandledRequest$ – TRUE indicates a fresh request has been placed in the system.

VARIABLES $inState,\ unhandledRequest$

$$vars \triangleq \langle inState,\ unhandledRequest\rangle$$

```
────────────────────────────────────────────────────────────────────────
The environment model

Initially there are no requests.
InitEnv ≜ unhandledRequest = FALSE

The environment sets the unhandledRequest flag at some arbitrary moment to indicate a new request.
RequestArrival ≜ ∧ unhandledRequest = FALSE
                 ∧ unhandledRequest' = TRUE
                 ∧ UNCHANGED inState

Somebody, but not the environment, will collect the request
Also, inState changes independently of the environment
ServAgent ≜ ∨ ∧ unhandledRequest = TRUE
              ∧ unhandledRequest' = FALSE
            ∨ ¬UNCHANGED inState

EnvSpec ≜ ∧ InitEnv
          ∧ □[RequestArrival ∨ ServAgent]_vars

────────────────────────────────────────────────────────────────────────
The actual service.

Initially we start out in the Idle state
InitServ ≜ inState = "Idle"

The transition from idle to handling request is triggered by an incoming request
StartRequest ≜ ∧ inState = "Idle"
               ∧ unhandledRequest = TRUE
               ∧ inState' = "HandlingRequest"
               ∧ unhandledRequest' = FALSE

Request handling can finish any time
FinishRequest ≜ ∧ inState = "HandlingRequest"
                ∧ inState' = "Idle"
                ∧ UNCHANGED unhandledRequest

NextServ ≜ StartRequest ∨ FinishRequest

The environment occasionally provides new requests
EnvAgent ≜ ∧ unhandledRequest = FALSE
           ∧ unhandledRequest' = TRUE

ServiceSpec ≜ ∧ InitServ
              ∧ □[ ∨ NextServ
                   ∨ EnvAgent]_vars

────────────────────────────────────────────────────────────────────────

Service ≜ EnvSpec -+▷ ServiceSpec

────────────────────────────────────────────────────────────────────────
```

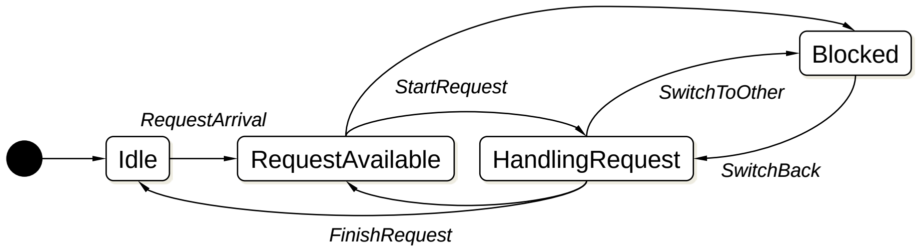

Figure 2: State-machine representation of the component-operation context model

4.2 A Context Model for Component Operations

Figure 2 gives the state-machine view of this context model for comparison with the specification. The same notes as in the previous subsection apply also for this context model specification.

MODULE *Component*

Context Model of a component implementation.

Variables:

unhandledRequest – set to TRUE by the environment to indicate that a new request has arrived and should be handled.

inState – the state in which the component is.

VARIABLE $inState$

VARIABLE $unhandledRequest$

$vars \triangleq \langle inState, unhandledRequest \rangle$

The environment specification.

The environment in particular influences the *unhandledRequest* variable by entering new requests into the system.

Initially there are no requests in the system

$InitEnv \triangleq unhandledRequest = \text{FALSE}$

The environment sets the *unhandledRequest* flag at some arbitrary moment to indicate a new request.

$RequestArrival \triangleq \land unhandledRequest = \text{FALSE}$
$\qquad \land unhandledRequest' = \text{TRUE}$
$\qquad \land \text{UNCHANGED } inState$

Somebody, but not the environment, will collect the request

Also, *inState* changes independently of the environment

$CompAgent \triangleq \lor \land unhandledRequest = \text{TRUE}$
$\qquad \land unhandledRequest' = \text{FALSE}$
$\qquad \lor \neg\text{UNCHANGED } inState$

$EnvSpec \triangleq \land InitEnv$

$$\land \Box[RequestArrival \lor CompAgent]_{vars}$$

The actual component.

It mainly specifies changes to the $inState$ variable, however it communicates with the environment via the $unhandledRequest$ variable.

Initially we start out in the idle state

$$InitComponent \triangleq inState = \text{“Idle”}$$

The transition from idle to handling request is triggered by an incoming request

$$\begin{aligned} StartRequest \triangleq\ & \land inState = \text{“Idle”} \\ & \land unhandledRequest = \text{TRUE} \\ & \land \lor inState' = \text{“HandlingRequest”} \\ & \quad \lor inState' = \text{“Blocked”} \\ & \land unhandledRequest' = \text{FALSE} \end{aligned}$$

Request handling can finish any time

$$\begin{aligned} FinishRequest \triangleq\ & \land inState = \text{“HandlingRequest”} \\ & \land inState' = \text{“Idle”} \\ & \land \text{UNCHANGED } unhandledRequest \end{aligned}$$

Also, the runtime environment may at any time take away the CPU from us and assign it to someone else.

$$\begin{aligned} SwitchToOther \triangleq\ & \land inState = \text{“HandlingRequest”} \\ & \land inState' = \text{“Blocked”} \\ & \land \text{UNCHANGED } unhandledRequest \end{aligned}$$

But, it may also at any time give back the CPU to us

$$\begin{aligned} SwitchBack \triangleq\ & \land inState = \text{“Blocked”} \\ & \land inState' = \text{“HandlingRequest”} \\ & \land \text{UNCHANGED } unhandledRequest \end{aligned}$$

$$\begin{aligned} NextComponent \triangleq\ & \lor StartRequest \lor FinishRequest \\ & \lor SwitchToOther \lor SwitchBack \end{aligned}$$

The environment occasionally provides new requests

$$\begin{aligned} EnvAgent \triangleq\ & \land unhandledRequest = \text{FALSE} \\ & \land unhandledRequest' = \text{TRUE} \end{aligned}$$

$$\begin{aligned} ComponentSpec \triangleq\ & \land InitComponent \\ & \land \Box[\lor NextComponent \\ & \qquad \lor EnvAgent]_{vars} \end{aligned}$$

The complete specification

$$Component \triangleq EnvSpec \overset{+}{\triangleright} ComponentSpec$$

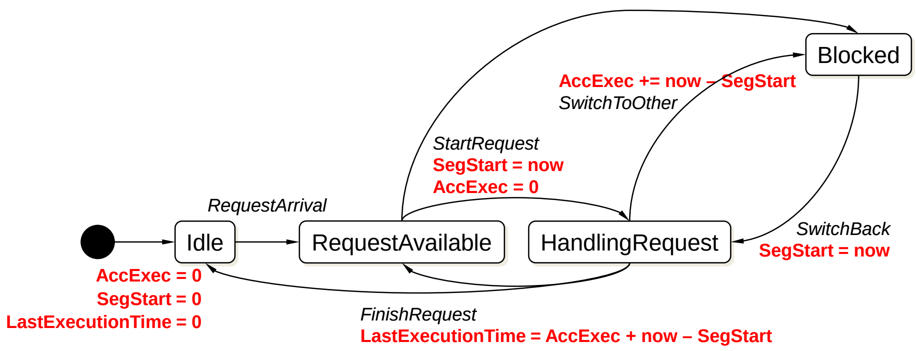

Figure 3: Definition of the execution-time measurement

5 Measurement Definition

Based on the context models defined in the previous section, we can now define measurements. In this section, we will define three different measurements:

1. *Execution time:* This intrinsic measurement is based on the component context model from Sect. 4.2 and represents the execution time of the last invocation of a component operation.

2. *Response time:* This extrinsic measurement is based on the service context model from Sect. 4.1 and represents the response time of the last invocation of a service operation.

3. *Inter-request time:* This extrinsic measurement is based on the service context model from Sect. 4.1 and represents the time between the two last invocations of a service operation.

The following subsections present these measurement specifications in full detail. Additionally, each module will also predefine a parametrised property based on the measurement.

5.1 Execution Time

Based on the `Component` context model, we define the execution-time measurement. This specification uses the $Component$ module (Line 47) and attaches actions measuring the time the component spends actually computing (this is specified on Lines 55–78). The execution time of the last completed invocation of the operation is stored in variable $LastExecutionTime$. In addition to defining the measurement, Line 106 adds a constraint on execution time. This is parametrised by an upper-bound value passed in through the $ExecutionTime$ parameter defined on Line 14.
Figure 3 shows the corresponding state-machine representation. The new variables and actions introduced by the measurement definition are high-lighted in red.

```
┌──────────────── MODULE ExecTimeConstrainedComponent ────────────────┐
Specification of a component which offers one operation the execution time of which can be constrained.

EXTENDS RealTime

Parameters:

ExecutionTime – an upper bound for the execution time of the component's operation.

CONSTANT ExecutionTime
ASSUME
  (ExecutionTime ∈ Real) ∧ (ExecutionTime > 0)

Variables:

inState            – the state in which the component currently is.
unhandledRequest   – TRUE if the environment put another request into the system.
LastExecutionTime  – the execution time of the last service execution.

VARIABLES inState, unhandledRequest
VARIABLE  LastExecutionTime

├──────────────────────────────────────────────────────────────────────┤

  ┌──────────────── MODULE Inner ────────────────┐
Internal module containing the actual specification.

Variables:

AccExec    – The accumulated execution time of the current service execution.
SegStart   – The start time of the current service execution.

VARIABLE AccExec
VARIABLE SegStart

 Based on the component context model
BasicComponent ≜ INSTANCE Component

├──────────────────────────────────────────────────────────────────────┤

  Init ≜ ∧ AccExec = 0
         ∧ SegStart = 0
         ∧ LastExecutionTime = 0

   StartNext reacts to a StartRequest step
  StartNext ≜ BasicComponent!StartRequest ⇒
                   ∧ SegStart′ = now
                   ∧ AccExec′ = 0
                   ∧ UNCHANGED LastExecutionTime

   RespNext reacts to a FinishRequest step
  RespNext ≜ BasicComponent!FinishRequest ⇒
                   ∧ LastExecutionTime′ =
                         AccExec + now − SegStart
                   ∧ UNCHANGED ⟨SegStart, AccExec⟩

   STONext reacts to a SwitchToOther step
  STONext ≜ BasicComponent!SwitchToOther ⇒
                   ∧ AccExec′ =
```

```
70                                    AccExec + now − SegStart
71                                ∧ UNCHANGED ⟨LastExecutionTime,
72                                             SegStart⟩

74    SBNext reacts to a SwitchBack step
75    SBNext ≜ BasicComponent!SwitchBack ⇒
76                          ∧ SegStart′ = now
77                          ∧ UNCHANGED ⟨LastExecutionTime,
78                                       AccExec⟩

80    ExcludeOtherChange ≜
81          (¬ ∨ BasicComponent!StartRequest
82             ∨ BasicComponent!FinishRequest
83             ∨ BasicComponent!SwitchToOther
84             ∨ BasicComponent!SwitchBack)
85          ⇒ UNCHANGED ⟨AccExec, SegStart, LastExecutionTime⟩

88    Next ≜  ∧ StartNext
89            ∧ RespNext
90            ∧ STONext
91            ∧ SBNext
92            ∧ ExcludeOtherChange

94    ctxvars ≜ ⟨inState, unhandledRequest⟩
95    vars ≜ ⟨AccExec, SegStart, LastExecutionTime,
96            inState, unhandledRequest⟩

98    Spec ≜  ∧ Init
99            ∧ □[Next ∧ ¬UNCHANGED ctxvars]_vars

101   Compose the various partial specifications
102   Component ≜
103     ∧ BasicComponent!Component
104     ∧ RTnow(vars)
105     ∧ Spec
106     ∧ □(LastExecutionTime ≤ ExecutionTime)

108  └──────────────────────────────────────────────────────┘

110 ├────────────────────────────────────────────────────────┤

112 _Component(AccExec, SegStart) ≜ INSTANCE Inner
113 Component ≜
114     ∃ ae, ss : _Component(ae, ss)!Component

116 └────────────────────────────────────────────────────────┘
```

5.2 Response Time

Based on the `Service` context model, we define the response-time measurement, in a similar fashion to execution time. Here, too we already added the definition of a constraint on response time on Line 70.

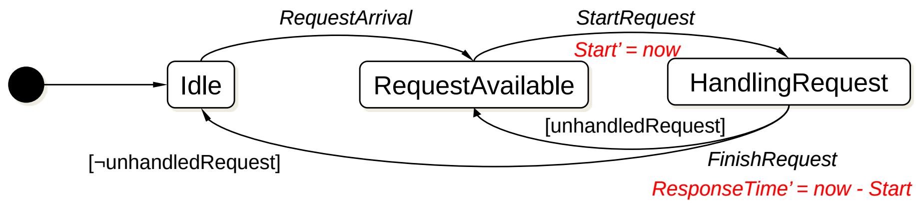

Figure 4: Definition of the response-time measurement

Figure 4 shows the corresponding state-machine representation. The new variables and actions introduced by the measurement definition are high-lighted in red.

```
---------------- MODULE ResponseTimeConstrainedService ----------------
EXTENDS RealTime

  Parameter:
  ResponseTime – Maximum response time a request should exhibit.

CONSTANT ResponseTime
ASSUME (ResponseTime ∈ Real) ∧ (ResponseTime > 0)

  Variables:
  inState           – the current state of the service machinery.
  unhandledRequest  – TRUE indicates the arrival of a new request.
  LastResponseTime  – the response time of the last request serviced.

VARIABLES inState, unhandledRequest
VARIABLE  LastResponseTime

------------------------------------------------------------------------

  ---------------------- MODULE Inner ----------------------
  The actual specification.

  Variables:
  Start – the start of the last request.

VARIABLE Start

   Based on the Service context model
Serv ≜ INSTANCE Service

------------------------------------------------------------------------

  Init ≜ ∧ Start = 0
         ∧ LastResponseTime = 0

   StartNext reacts to a StartRequest step
  StartNext ≜ Serv!StartRequest ⇒
                  ∧ Start′ = now
                  ∧ UNCHANGED LastResponseTime

   RespNext reacts to a FinishRequest step
  RespNext ≜ Serv!FinishRequest ⇒
```

```
51                          ∧ LastResponseTime' = now − Start
52                          ∧ UNCHANGED Start

54  ExcludeOtherChange ≜
55      ¬(Serv!StartRequest ∨ Serv!FinishRequest)
56       ⇒ UNCHANGED ⟨Start, LastResponseTime⟩

58  Next ≜ StartNext ∧ RespNext ∧ ExcludeOtherChange

60  ctxvars ≜ ⟨inState, unhandledRequest⟩
61  vars ≜ ⟨Start, LastResponseTime, inState,
62            unhandledRequest⟩

64  RespSpec ≜ ∧ Init
65             ∧ □[Next ∧ ¬UNCHANGED ctxvars]_vars

67  Service ≜ ∧ Serv!Service
68            ∧ RTnow(vars)
69            ∧ RespSpec
70            ∧ □(LastResponseTime ≤ ResponseTime)

72  ─────────────────────────────────────────────

74  ─────────────────────────────────────────────

76  _Service(Start) ≜ INSTANCE Inner
77  Service ≜ ∃ s : _Service(s)!Service

79  ─────────────────────────────────────────────
```

5.3 Inter-Request Time

The following module describes an additional measurement, that we will use to describe environment behaviour in later modules. It measures the time between individual requests for a service sent by the environment. This is later used to define a constraint on the frequency with which the environment sends request for an operation to a given service (see Line 65). The specification is parametrised: The desired minimum time between requests should be passed to the constant *RequestPeriod*. The actual specification is encapsulated in module *Inner* on Lines 63–65. It makes use of the specification of a service from above.

Figure 5 shows the corresponding state-machine representation. The new variables and actions introduced by the measurement definition are high-lighted in red. There is a subtle difference between the simplified state-machine diagram and the actual specification: Because we have separated environment specification and service specification in the service context model (see Sect. 4.1), the TLA$^+$ specification actually measures all incoming requests including those arriving during request handling.

```
1  ──────────── MODULE MaxRequPeriodEnv ────────────
```

Specification of a system environment which sends service request with a certain minimum time between individual requests.

Note that this is not a specification of what we expect from an environment but actually a description of a behaviour of one specific system environment. It only becomes a specification of an expectation the way it is used in the system specification.

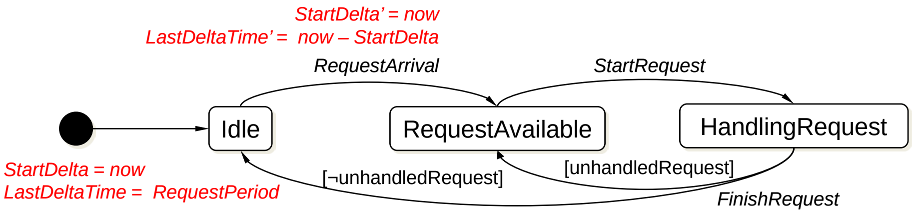

Figure 5: Definition of the inter-request–time measurement

EXTENDS *RealTime*

Parameters:

RequestPeriod – the lower limit for the time between individual requests that should be observed by the environment.

CONSTANT *RequestPeriod*

ASSUME $(RequestPeriod \in Real) \land (RequestPeriod > 0)$

Variables:

LastDeltaTime – The amount of time between the last two requests.
inState – Current state of the service invoked.
unhandledRequest – TRUE signals that a new request has been put into the system.

VARIABLES *LastDeltaTime*

VARIABLES *inState*, *unhandledRequest*

MODULE *Inner*

The actual specification.

Variables:

StartDelta – Start time of the last request.

VARIABLE *StartDelta*

$TheService \triangleq$ INSTANCE *Service*

$vars \triangleq \langle inState, unhandledRequest, LastDeltaTime, StartDelta \rangle$

$$Init \triangleq \land\ LastDeltaTime = RequestPeriod$$
$$\qquad \land\ StartDelta = now$$

$$NewRequest \triangleq TheService!RequestArrival$$
$$\qquad \Rightarrow \land\ LastDeltaTime'$$
$$\qquad\qquad = now - StartDelta$$
$$\qquad\quad \land\ StartDelta' = now$$

$$ReqPeriod \triangleq \land\ Init$$
$$\qquad \land\ \Box[NewRequest]_{vars}$$

$$Service \triangleq \land\ TheService!Service$$

$\land\ ReqPeriod$
$\land\ \Box(LastDeltaTime \geq RequestPeriod)$

$\_Environment(StartDelta) \triangleq$ INSTANCE $Inner$
$Environment \triangleq \exists\, sd : \_Environment(sd)!Service$

---

6 Resource specification

The following three specifications deal with the resource CPU. Each of the modules specifies one of the layers of a resource specification (see [3, p. 13]):

1. The *resource-service layer* models the service provided by the resource. Here, the corresponding module models the essential service provided by a CPU: to be available to tasks for a certain time and to be assigned from one task to another, eventually serving all tasks.

2. The *resource-measurement layer* provides measurement definitions that allow quantitative statements to be made about a resource.

3. The *resource-property layer* defines constraints over the measurements defined in the resource-measurement layer. Here, we define a RMS-scheduled CPU and its schedulability criterion.

6.1 Resource-Service Layer

The first module defines what a CPU is: It is a resource that is allocated to tasks one at a time in some fashion. Constant $TaskCount$ is used to identify the number of tasks to be scheduled, variable $AssignedTo$ indicates the task to which the resource has currently been assigned.

MODULE $CPUScheduler$

A CPU Scheduler allocates the resource CPU to various tasks. We model this through a variable $AssignedTo$ holding in each state the number of the task which has currently been allocated the resource.

EXTENDS $Naturals$

Parameters:

$TaskCount$ – the number of tasks which need to share the resource.

CONSTANT $TaskCount$
ASSUME $(TaskCount \in Nat) \land (TaskCount > 0)$

Variables:

$AssignedTo$ – holds the number of the task currently assigned the resource

VARIABLE $AssignedTo$

$AssignedToType \triangleq \{1 \,..\, TaskCount\}$

```
Initially, an arbitrary task has been assigned the CPU.
Init                ≜ AssignedTo ∈ AssignedToType

The Switch action reassigns the resource from from to to.
Switch(from, to)    ≜ ∧ AssignedTo = from
                      ∧ AssignedTo' = to

The CPU can be switched from any task to any other task.
Next                ≜ ∃ i ∈ AssignedToType :
                        ∃ j ∈ AssignedToType :
                          Switch(i, j)

CPUScheduler        ≜ ∧ Init
                      ∧ □[Next]_AssignedTo
```

6.2 Resource-Measurement Layer

The next specification adds some history-determined variables (quite similar to measurements) that allow to determine for what amount of time each task has been allocated the resource. It is based on the previous specificatio, which it imports on Line 38. In addition to the $TaskCount$ parameter, it introduces the parameter $Periods$ storing the requested period length per task, so that times can be determined per period. Lines 175–177, finally, provide a boolean measurement formalising the situation where all tasks get a sufficiently large share of the resource. To this end, an additional parameter $Wcets$ is introduced. This parameter captures the requested amount of time per period for each task.

The newly defined variables are all array variables. We, therefore, need to use TLA$^+$'s syntax for array definition and update:

- $[k \in K \mapsto e(k)]$ represents an array that is defined for all $k \in K$. The value associated to a specific k is defined by $e(k)$.
- $A[k]$ represents the value associated with k in array A.
- $[A \text{ EXCEPT } ![k] = e]$ represents an array that is identical to array A except that value k is mapped to the result of expression e. e may use the special identifier @, which stands for $A[k]$.

```
1  ---------------- MODULE TimedCPUScheduler ----------------
   A CPU scheduler for which the time each task is assigned can be measured.

   The corresponding formulae are derived by conjoining history variables to the CPU scheduler specification.
8  EXTENDS RealTime

   Parameters:

   TaskCount – the number of tasks which need to share the resource.
   Periods – the periods of each task This is an array with one entry per task.
   Wcets   – the worst case execution times of the tasks to be scheduled. This is an array with one entry per
             task.
19 CONSTANT TaskCount
20 ASSUME (TaskCount ∈ Nat) ∧ (TaskCount > 0)
```

```
CONSTANT Periods
ASSUME Periods ∈ [{1 .. TaskCount} → Real]

CONSTANT Wcets
ASSUME Wcets ∈ [{1 .. TaskCount} → Real]

Variables:

MinExecTime – records for each task the minimum amount of execution time per period it has been allo-
              cated over all periods so far.
AssignedTo – holds the number of the task currently assigned the resource

VARIABLE MinExecTime
VARIABLE AssignedTo

CPUSched ≜ INSTANCE CPUScheduler
----------------------------------------------------------------------------

  ---------------------------- MODULE Inner ----------------------------
Inner module with the actual specification. This is done so that we can hide some of the helper variables.

Variables:

ExecTimeStart – Records for each task the time when it last started executing
LastExecTime – Records the last accumulated execution time for each task.
LastPeriodStart – Records for each task when it last started a period.

VARIABLES ExecTimeStart, LastExecTime
VARIABLE LastPeriodStart

----------------------------------------------------------------------------

   A little helper function
  Min(a, b) ≜ CASE a ≤ b → a
                 □ a > b → b
  ------------------------------------------------------------------------
  Init ≜ ∧ ExecTimeStart =
               [k ∈ CPUSched!AssignedToType ↦ 0]
         ∧ LastExecTime =
               [k ∈ CPUSched!AssignedToType ↦ 0]
         ∧ IF (TaskCount > 1) THEN
                 We start out with infinity, so that any real
                 execution time will definitely be smaller
                 MinExecTime =
                    [k ∈ CPUSched!AssignedToType
                      ↦ Infinity]
             ELSE
                 We need to handle this case specially for
                 technical reasons
                 MinExecTime =
                    [k ∈ CPUSched!AssignedToType
                      ↦ Periods[k]]
         ∧ LastPeriodStart =
               [k ∈ CPUSched!AssignedToType ↦ 0]
```

Next we define what happens when a $CPUSched!Switch$ occurs

$$OnSwitch(from,\ to) \triangleq$$

Cumulate the time the CPU was allocated to Task from

$$\begin{aligned}
\land\ & LastExecTime' = [LastExecTime \ \text{EXCEPT} \\
& \qquad ![from] = @ + now \\
& \qquad\qquad - ExecTimeStart[from]]
\end{aligned}$$

Remember when Task to received the CPU

$$\begin{aligned}
\land\ & ExecTimeStart' = [ExecTimeStart \ \text{EXCEPT} \\
& \qquad ![to] = now] \\
\land\ & \text{UNCHANGED } \langle MinExecTime,\ LastPeriodStart \rangle
\end{aligned}$$

The $OSNext$ action binds $OnSwitch$ to corresponding $Switch$ actions

$$\begin{aligned}
OSNext \triangleq\ & \forall i \in CPUSched!AssignedToType : \\
& \quad \forall j \in CPUSched!AssignedToType : \\
& \qquad CPUSched!Switch(i,\ j) \\
& \qquad \Rightarrow OnSwitch(i,\ j)
\end{aligned}$$

The $ExecTime$ action determines the accumulated execution time for task i in the next state, but at most to the end of its current period. A helper action used by action $PeriodEnd$ below.

$$\begin{aligned}
ExecTime(i) \triangleq\ & LastExecTime[i] + \\
& \quad \text{IF } (AssignedTo = i) \text{ THEN} \\
& \qquad Min(now', \\
& \qquad\qquad LastPeriodStart[i] + \\
& \qquad\qquad\quad Periods[i]) - \\
& \qquad ExecTimeStart[i] \\
& \quad \text{ELSE } 0
\end{aligned}$$

The $PeriodEnd$ action reacts to the end of a period for task i

$$PeriodEnd(i) \triangleq$$

$\land$ A period is going to end

$$(now' - LastPeriodStart[i]) \geq Periods[i]$$

$\land$ The following is the measurement we are really interested in

$$\begin{aligned}
& MinExecTime' = [MinExecTime \ \text{EXCEPT} \\
& \qquad ![i] = Min(@,\ ExecTime(i))]
\end{aligned}$$

$\land$ But we also need to perform some cleanup to prepare for the next period

$$\begin{aligned}
& LastPeriodStart' = [LastPeriodStart \ \text{EXCEPT} \\
& \qquad ![i] = @ + Periods[i]] \\
\land\ & ExecTimeStart' = [ExecTimeStart \ \text{EXCEPT} \\
& \qquad ![i] = LastPeriodStart'[i]] \\
\land\ & LastExecTime' = [LastExecTime \ \text{EXCEPT } ![i] = 0]
\end{aligned}$$

$CheckPeriods$ catches all period ends of all tasks

$$\begin{aligned}
& CheckPeriods \triangleq \\
& \quad \text{IF } (TaskCount > 1) \text{ THEN} \\
& \qquad \forall k \in CPUSched!AssignedToType : PeriodEnd(k) \\
& \quad \text{ELSE}
\end{aligned}$$

If there's only one process it will be allowed to run for the whole period

```
      MinExecTime'[1] = Periods[1]

  Next ≜ OSNext

  vars ≜ ⟨AssignedTo, ExecTimeStart, LastExecTime⟩

  timeVars ≜ ⟨LastPeriodStart, MinExecTime, now⟩

  TimingSpecification ≜ ∧ RTnow(vars)
                        ∧ Init
                        ∧ □[Next]_vars
                        ∧ □[CheckPeriods]_timeVars

  TimedCPUScheduler ≜ ∧ CPUSched!CPUScheduler
                      ∧ TimingSpecification

 └──────────────────────────────────────────────┘

├──────────────────────────────────────────────┤

_TimedCPUScheduler(ExecTimeStart, LastExecTime,
                   LastPeriodStart)
    ≜ INSTANCE Inner
TimedCPUScheduler
    ≜ ∃ ets, let, lps :
        _TimedCPUScheduler(ets, let, lps)
        !TimedCPUScheduler

ExecutionTimesOk ≜
    ∀ k ∈ CPUSched!AssignedToType :
      (MinExecTime[k] ≥ Wcets[k])

└──────────────────────────────────────────────┘
```

6.3 Resource-Property Layer

Finally, *RMSScheduler* below defines an actual CPU which is scheduled using RMS. The main contribution of this specification is the schedulability criterion defined on Lines 59–64. This is the standard RMS schedulability criterion.

```
1  ┌──────────── MODULE RMSScheduler ────────────┐
   A CPU Scheduler using RMS.
5  EXTENDS Reals

   Parameters:

   TaskCount – the number of tasks to be scheduled on the CPU.
   Periods – the periods to be scheduled for these tasks. This is an array with one entry per task.
   Wcets   – the worst case execution times of the tasks to be scheduled. This is an array with one entry per
             task.
16 CONSTANT TaskCount
17 ASSUME (TaskCount ∈ Nat) ∧ (TaskCount > 0)

19 CONSTANT Periods
20 ASSUME Periods ∈ [{1 .. TaskCount} → Real]
```

```
CONSTANT Wcets
ASSUME Wcets ∈ [{1 .. TaskCount} → Real]

  Variables:
  MinExecTime – records for each task the minimum amount of execution time it has been allocated over
                all periods so far.
  AssignedTo – holds the number of the task currently assigned the resource
  now        – the current time.

VARIABLE MinExecTime
VARIABLE AssignedTo
VARIABLE now

TimedCPUSched ≜ INSTANCE TimedCPUScheduler

├──────────────────────────────────────────────────────────────┤
  A few helpers

  bth root of a
sqrt(b, a) ≜ a^(1/b)

  Sum of all the elements in an array (function)
  Copied from Bags.tla
Sum(f) ≜
  LET DSum[S ∈ SUBSET DOMAIN f] ≜
    LET elt ≜ CHOOSE e ∈ S : TRUE
    IN   IF S = {}
           THEN 0
           ELSE f[elt] + DSum[S \ {elt}]
  IN    DSum[DOMAIN f]

├──────────────────────────────────────────────────────────────┤
  Schedulable is TRUE if the given task load can be scheduled using RMS.
Schedulable ≜
  LET usage ≜ [k ∈ {1 .. TaskCount}
                  ↦ (Wcets[k]/Periods[k])]
  IN
    Sum(usage) ≤ (TaskCount ∗ (sqrt(TaskCount, 2)
                    − 1))

  The actual specification: A TimedCPUScheduler which will meet all deadlines provided the RMS
  schedulability is met by the tasks to be scheduled.
RMSScheduler ≜
   ∧ TimedCPUSched!TimedCPUScheduler
   ∧ □Schedulable
     -+▷ □TimedCPUSched!ExecutionTimesOk

└──────────────────────────────────────────────────────────────┘
```

7 Container Strategy Specification

Resource allocations, intrinsic component properties, and extrinsic service properties must be related by a container strategy. The following module defines such a container strategy. It is structured into four major parts:

1. Import of measurements and abstract resource specifications required. This is on Lines 49–176.
2. Definition of container expectations. This is on Lines 179–198.
3. Definition of services guaranteed by the container. This is on Lines 201–215.
4. The actual container strategy specification. This is on Lines 217–218.

The container strategy is parametrised by the response time it should provide and the worst-case execution time it can expect. To ensure that only sensible parameter values are provided, a sanity check is performed on Line 180.
We require container strategies to be *functionality preserving;* that is, the functionality offered by the service should be the same as the functionality provided by the underlying component. This if formally expressed on Line 194. Notice, that the predicates used to express component and service behaviour are left open as parameters to the specification by defining them as abstrract predicates on Lines 102–110 and 168–176. This way, the specification will be applicable to arbitrary concrete components and services. All that needs to be done is to associate these abstract predicates with concrete predicates when instantiating the container-strategy module.

```
1  ---------------------- MODULE SimpleContainer ----------------------
   A container specification for a very simple container. This container manages just one component instance
   and tries to achieve a certain response time with it.
7  EXTENDS RealTime

   Parameters:

   ResponseTime – the response time the container should achieve.
   ExecutionTime – the execution time of the component available.

15 CONSTANT ResponseTime
16 ASSUME (ResponseTime ∈ Real) ∧ (ResponseTime > 0)

18 CONSTANT ExecutionTime
19 ASSUME
20   (ExecutionTime ∈ Real) ∧ (ExecutionTime > 0)

   Variables:

   TaskCount  – the number of tasks the container would want to execute on the CPU.
   Periods    – the periods the container associates with these tasks.
   Wcets      – the worst case execution times the container associates with these tasks.

31 VARIABLES TaskCount, Periods, Wcets

33 --------------------------------------------------------------------

   Specification of required CPU scheduling behaviour. Note that this does not make any statement about the
   actual scheduling regime, but only states what tasks need to be scheduled.

   Variables:

   CPUMinExecTime – records for each task the minimum amount of execution time it has been allocated
                    over all periods so far.
   CPUAssignedTo – holds the number of the task currently assigned the resource.
```

```
VARIABLES CPUMinExecTime, CPUAssignedTo

_SomeCPUScheduler(TaskCountConstraint,
                  PeriodsConstraint,
                  WcetsConstraint)
    ≜ INSTANCE TimedCPUScheduler
        WITH MinExecTime ← CPUMinExecTime,
             AssignedTo ← CPUAssignedTo,
             TaskCount ← TaskCountConstraint,
             Periods ← PeriodsConstraint,
             Wcets ← WcetsConstraint
CPUCanSchedule(TaskCountConstraint,
               PeriodsConstraint,
               WcetsConstraint)
    ≜ ∧ _SomeCPUScheduler(TaskCount,
                          Periods,
                          Wcets)
        !TimedCPUScheduler
      ∧ □_SomeCPUScheduler(TaskCount,
                           Periods,
                           Wcets)
        !ExecutionTimesOk
```

Specification of required component behaviour.

Variables:

CmpInState – the state in which the component currently is.
CmpUnhandledRequest – TRUE if the environment put another request into the system.
CmpLastExecutionTime – the execution time of the last service execution.

```
VARIABLES CmpInState, CmpUnhandledRequest
VARIABLE  CmpLastExecutionTime

_Component(ExecutionTimeConstraint)
    ≜ INSTANCE ExecTimeConstrainedComponent
        WITH
        ExecutionTime ← ExecutionTimeConstraint,
        inState ← CmpInState,
        unhandledRequest ← CmpUnhandledRequest,
        LastExecutionTime ← CmpLastExecutionTime
ComponentMaxExecTime(ExecutionTimeConstraint)
    ≜ _Component(ExecutionTimeConstraint)
        !Component
```

This predicate represents the functionality of the component.

```
CONSTANT CompFun
ASSUME CompFun ∈ BOOLEAN
```

This predicate represents the mapping between functionality and context model of the component.

```
CONSTANT CompModelMapping
ASSUME CompModelMapping ∈ BOOLEAN
```

```
----------------------------------------------------------------------
```

Specification of required request interarrival time.

Variables:

EnvLastDeltaTime – The amount of time between the last two requests.
EnvInState – Current state of the service invoked.
EnvUnhandledRequest – TRUE signals that a new request has been put into the system.

```
VARIABLES EnvLastDeltaTime, EnvInState
VARIABLE  EnvUnhandledRequest

_MinInterrequestTime(RequestPeriodConstraint)
     ≜ INSTANCE MaxRequPeriodEnv
        WITH
       RequestPeriod  ← RequestPeriodConstraint,
       LastDeltaTime ← EnvLastDeltaTime,
       inState ← EnvInState,
       unhandledRequest ← EnvUnhandledRequest
MinInterrequestTime(RequestPeriodConstraint)
     ≜ _MinInterrequestTime(RequestPeriodConstraint)
       !Environment

----------------------------------------------------------------------
```

Specification of guaranteed service behaviour.

Variables:

ServLastResponseTime – the response time of the last request serviced.
ServInState – the current state of the service machinery.
ServUnhandledRequest – TRUE indicates the arrival of a new request.

```
VARIABLES ServLastResponseTime, ServInState
VARIABLE  ServUnhandledRequest

_ServiceResponseTime(ResponseTimeConstraint)
     ≜ INSTANCE ResponseTimeConstrainedService
        WITH
       ResponseTime ← ResponseTimeConstraint,
       LastResponseTime ← ServLastResponseTime,
       inState ← ServInState,
       unhandledRequest ← ServUnhandledRequest
ServiceResponseTime(ResponseTimeConstraint)
     ≜ _ServiceResponseTime(ResponseTimeConstraint)
       !Service
```

This predicate represents the functionality of the service.

```
CONSTANT ServFun
ASSUME ServFun ∈ BOOLEAN
```

This predicate represents the mapping between functionality and context model of the service.

```
CONSTANT ServModelMapping
ASSUME ServModelMapping ∈ BOOLEAN

----------------------------------------------------------------------
ContainerPreCond ≜
```

$\wedge\ ExecutionTime \leq ResponseTime$

$\wedge$ The CPU must be able to schedule exactly one task with a period equal to the requested response time and a wcet equal to the specified execution time of the available component.

$\wedge\ CPUCanSchedule(1, [n \in \{1\} \mapsto ResponseTime], [n \in \{1\} \mapsto ExecutionTime])$

$\wedge$ A component with the required execution time is available.

$\wedge\ ComponentMaxExecTime(ExecutionTime)$

$\wedge\ CompFun$

$\wedge\ CompModelMapping$

$\wedge$ The component functionality implements the service functionality.

$CompFun \Rightarrow ServFun$

$\wedge$ Requests arrive with a constant period, the length of which is somehow related to the period length requested from the CPU.

$\wedge\ MinInterrequestTime(ResponseTime)$

$ContainerPostCond \triangleq$

$\wedge$ The promised response time can be guaranteed

$\wedge\ ServiceResponseTime(ResponseTime)$

$\wedge\ ServFun$

$\wedge\ ServModelMapping$

$\wedge$ The container will allocate exactly one task for the component.

$\Box \wedge TaskCount = 1$

$\wedge\ Periods = [n \in \{1\} \mapsto ResponseTime]$

$\wedge\ Wcets = [n \in \{1\} \mapsto ExecutionTime]$

$\wedge$ State that the container will hand requests directly to the component, without buffering them in any way. If the container provides buffering, this would need to go here

$\Box(CmpUnhandledRequest = EnvUnhandledRequest)$

$Container \triangleq$

$ContainerPreCond \overset{+}{\rhd} ContainerPostCond$

8 The Counter Application

So far, we have been discussing the non-functional properties in the abstract. In the following specifications, we define a sample Counter application and provide model mappings to apply our measurements to this application.

8.1 Application Model

The next two modules define the Counter application itself. We begin with the definition of its interface. Notice that this is just a helper module that we will later use to hide the actual implementation of the Counter application. The interface module uses abstract actions to define the interactions with the environment that can be observed of a Counter application without defining how they are realised. An abstract action is defined by a *Boolean* constant, possibly with open parameter slots (indicated by _). More details on abstract actions can be found in [1].

```
──────────────── MODULE CounterInterface ────────────────

  A global representation of the counter's state. We do not say anything about what this state looks like.
VARIABLE counterState

  A DoInc (counterState, counterState') step represents an incoming request to increment the internal
  counter of the component
CONSTANT DoInc(_, _)

  A GetData (counterState, counterState') step represents an incoming request for the current value.
CONSTANT GetData(_, _)

  A SendData (value, counterState, counterState') step represents a response to a GetData step.
CONSTANT SendData(_, _, _)

CONSTANT InitialCounterStates

ASSUME ∀ v, csOld, csNew :
        ∧ DoInc    (csOld, csNew)     ∈ BOOLEAN
        ∧ GetData  (csOld, csNew)     ∈ BOOLEAN
        ∧ SendData(v, csOld, csNew)   ∈ BOOLEAN

──────────────────────────────────────────────────────────
```

The next module defines the actual Counter implementing this interface. This is the application model of our example. It is a normal TLA$^+$ specification. However, note how it binds the Counter interface from the previous module by referencing the abstract actions on Lines 14, 20, and 33.

```
1  ──────────────── MODULE CounterApp ────────────────
3  EXTENDS CounterInterface, Naturals
5   Internal variables:
6  VARIABLE internalCounter
7  VARIABLE doHandle
9  Init ≜  ∧ internalCounter = 0
10         ∧ doHandle = 0
11         ∧ counterState ∈ InitialCounterStates
14 IncrementReq ≜  ∧ DoInc(counterState, counterState')
15                 ∧ doHandle = 0
```

```
16                       ∧ internalCounter'
17                            = internalCounter + 1
18                       ∧ UNCHANGED doHandle

20 ReceiveGetData ≜ ∧ GetData(counterState,
21                              counterState')
22                  ∧ doHandle = 0
23                  ∧ doHandle' = 1
24                  ∧ UNCHANGED internalCounter

26 HandleGetData ≜ ∧ doHandle = 1
27                 ∧ doHandle' = 2
28                 ∧ UNCHANGED ⟨internalCounter,
29                              counterState⟩

31 ReplyStep ≜ ∧ doHandle = 2
32             ∧ doHandle' = 0
33             ∧ SendData(internalCounter,
34                        counterState,
35                        counterState')
36             ∧ UNCHANGED internalCounter

38 Next ≜ ∨ IncrementReq
39        ∨ ReceiveGetData ∨ HandleGetData
40        ∨ ReplyStep

42 vars ≜ ⟨counterState, internalCounter, doHandle⟩

44 Spec ≜ ∧ Init
45        ∧ [Next]_vars

47 ──────────────────────────────────────────────
```

8.2 Model Mappings

The following two specifications define the model mappings for execution time of the `Counter` component and for response time of the `Counter` service, resp. Both specifications work in a similar manner: They extend the *CounterApp* specification, so that all specifications and variables from that specification are directly available. Then, they import the measurement specification. Eventually, they define the *Model-Mapping* formula, the actual model-mapping relation ϕ_{App}^{Ctx} by relating states of the Counter application to states of the context model. Lines 55–57 in Module *CounterAppExecTime* and Lines 49–51 in Module *CounterAppResponseTime* finally encode the model mapping as given by Equation (2) in the main paper:

$$\Pi_{Ctx}^{App} \triangleq \Pi_{App} \wedge \Pi_{Ctx} \wedge \Box \left(\langle v_{App}, v_{Ctx} \rangle \in \phi_{App}^{Ctx} \right)$$

```
1 ──────────── MODULE CounterAppExecTime ────────────
  A module defining execution time of the GetData() operation.
5 EXTENDS CounterApp, Realtime
```

Variables:

inState – the state in which the component currently is.
unhandledRequest – TRUE if the environment put another request into the system.
LastExecutionTime – the execution time of the last service execution.

```
VARIABLE inState

VARIABLE unhandledRequest
VARIABLE ExecutionTime

ExecTimeSpec(ExecutionTimeConstr)
    ≜ INSTANCE ExecTimeConstrainedComponent
        WITH LastExecutionTime ← ExecutionTime,
             ExecutionTime ← ExecutionTimeConstr

CompSpec ≜ ExecTimeSpec(20)!Component
```

Definition of the context-model–application-model mapping

Note how this maps the *GetData*/*SendData* operation, but not *DoInc*.

```
ModelMapping ≜
  ∧ doHandle = 0 ⇒
        ∧ inState = "Idle"
        ∧ unhandledRequest = FALSE
  ∧ doHandle = 1 ⇒
        ∧ inState = "Idle"
        ∧ unhandledRequest = TRUE
  ∧ doHandle = 2 ⇒
        ∧ inState ∈ {"HandlingRequest",
                      "Blocked"}
        ∧ unhandledRequest = FALSE
      Dummy mapping for completeness' sake
  ∧ (doHandle ∉ {0, 1, 2}) ⇒
        ∧ inState = "Idle"
        ∧ unhandledRequest = FALSE
```

Final model of the counter component.

```
CounterComponent ≜ ∧ Spec
                   ∧ CompSpec
                   ∧ □ModelMapping
```

```
──────────── MODULE CounterAppResponseTime ────────────
A module defining response time of the GetData() operation.

EXTENDS CounterApp, Realtime

Variables:

ResponseTime       – the response time of the last request serviced.
inState            – the current state of the service machinery.
unhandledRequest   – TRUE indicates the arrival of a new request.

VARIABLES ResponseTime, inState, unhandledRequest

ResponseTimeSpec(ResponseTimeConstr)
    ≜ INSTANCE ResponseTimeConstrainedService
        WITH LastResponseTime ← ResponseTime,
             ResponseTime ← ResponseTimeConstr

ServSpec ≜ ResponseTimeSpec(50)!Service

────────────────────────────────────────────────────────

Definition of the context-model–application-model mapping

Note how this maps the GetData/SendData operation, but not DoInc.

ModelMapping ≜
  ∧ doHandle = 0 ⇒
        ∧ inState = "Idle"
        ∧ unhandledRequest = FALSE
  ∧ doHandle = 1 ⇒
        ∧ inState = "Idle"
        ∧ unhandledRequest = TRUE
  ∧ doHandle = 2 ⇒
        ∧ inState = "HandlingRequest"
        ∧ unhandledRequest = FALSE
  ∧ (doHandle ∉ {0, 1, 2}) ⇒
        ∧ inState = "Idle"
        ∧ unhandledRequest = FALSE

────────────────────────────────────────────────────────

Final model of the counter service .

CounterService ≜ ∧ Spec
                 ∧ ServSpec
                 ∧ □ModelMapping

────────────────────────────────────────────────────────
```

9 System Specification

Finally, we are ready to pull everything together. This we do in the system specification. The important bit is on Lines 237–264, where the system specification is composed from the individual elementary specifications. Everything before that is mainly of technical relevance, importing the previous specifications.

The actual connections between the component, the resource, the container, and the service are expressed by means of shared flexible variables. This can be seen in two ways in the specification: 1) on Lines 239–245 we explicitly pass parameters that perform part of the connection between container and resource and between component and resource; 2) on Lines 246–264 we use explicit constraints to relate other variables, relating the rest of the system parts to each other.

The complete system composition is then defined by formula *System* on Lines 237–264. Formula *ExternalService* on Lines 269–270 defines the service we expect the system to provide. Notice that this is conditional based on environment behaviour. Lines 278 and 279, finally, define what it means for the system to be feasible. This is the property we need to prove to show that we have indeed specified a feasible system. As explained in the main paper, we can make use of Abadi/Lamport's composition theorem for this proof.

```
1  ---------------- MODULE SystemSpecification ----------------
   A sample system specification.
   The system contains one counter with an execution time of 20 milliseconds, a RMS scheduled CPU, and a
   simple container.
8  EXTENDS Reals, CounterInterface

   Parameters:
   RequestPeriod – Part of an environment assertion: The environment promises to send requests with a
                   minimum distance of RequestPeriod milliseconds.
17 CONSTANT RequestPeriod
18 ASSUME (RequestPeriod ∈ Real) ∧ (RequestPeriod > 0)

   Variables:
   now – the current time.
25 VARIABLE now

27 ------------------------------------------------------------
   The counter component. The only intrinsic property offered by this component is its execution time, which
   is always less than 20ms.

   Variables:
   MyCompExec               – The last execution time of a service request handled by MyComponent.
   MyCompInState            – The current state of component MyComponent
   MyCompUnhandledRequest – Set to TRUE to send a request to MyComponent.
41 VARIABLES MyCompExec, MyCompInState
42 VARIABLE  MyCompUnhandledRequest
43 VARIABLES MyInternalCounter, MyDoHandle

45 _MyComponent ≜ INSTANCE CounterAppExecTime
46   WITH
47     ExecutionTime ← MyCompExec,
48     inState ← MyCompInState,
49     unhandledRequest ← MyCompUnhandledRequest,
50     internalCounter ← MyInternalCounter,
51     doHandle ← MyDoHandle

53  The actual component specification.
54 MyComponent ≜ _MyComponent!CounterComponent
```

```
CompMap       ≜ □_MyComponent!ModelMapping

_MyCompFunc ≜ INSTANCE CounterApp WITH
  internalCounter ← MyInternalCounter,
  doHandle ← MyDoHandle
MyCompFunc ≜ _MyCompFunc!Spec
```

A CPU. The parameters of the specification can be used to indicate the number of tasks to be scheduled, their respective periods as well as their respective worst case execution times.

Variables:

MYCPU_MinExecTime – records for each task the minimum amount of execution time it has been allocated over all periods so far.
MYCPU_AssignedTo – holds the number of the task currently assigned the resource

```
VARIABLE MYCPU_MinExecTime
VARIABLE MYCPU_AssignedTo

_MyCPU(TaskCount, Periods, Wcets)
      ≜   INSTANCE RMSScheduler WITH
            MinExecTime ← MYCPU_MinExecTime,
            AssignedTo ← MYCPU_AssignedTo
MyCPU(TaskCount, Periods, Wcets)
      ≜ _MyCPU(TaskCount, Periods, Wcets)
          !RMSScheduler
```

Environment specification.

Variables:

EnvLastDeltaTime – The amount of time between the last two requests.
EnvInState – Current state of the service invoked.
EnvUnhandledRequest – TRUE signals that a new request has been put into the system.

```
VARIABLES EnvLastDeltaTime, EnvInState
VARIABLE EnvUnhandledRequest

_Environment(RequestPeriodConstraint)
      ≜ INSTANCE MaxRequPeriodEnv WITH
          RequestPeriod  ← RequestPeriodConstraint,
          LastDeltaTime ← EnvLastDeltaTime,
          inState ← EnvInState,
          unhandledRequest ← EnvUnhandledRequest
Environment(RequestPeriodConstraint)
      ≜ _Environment(RequestPeriodConstraint)
          !Environment
```

The service the system is to perform.

Variables:

ServResponseTime – the response time of the last request serviced.
ServInState – the current state of the service machinery.
ServUnhandledRequest – TRUE indicates the arrival of a new request.

```
VARIABLES ServResponseTime, ServInState
VARIABLE  ServUnhandledRequest
VARIABLES ServInternalCounter, ServDoHandle

_Service
    ≜ INSTANCE CounterAppResponseTime WITH
        ResponseTime ← ServResponseTime,
        inState ← ServInState,
        unhandledRequest ← ServUnhandledRequest,
        internalCounter ← ServInternalCounter,
        doHandle ← ServDoHandle
Service ≜ _Service!CounterService
ServMap    ≜ □_Service!ModelMapping

_MyServFunc
    ≜ INSTANCE CounterApp WITH
        internalCounter ← ServInternalCounter,
        doHandle ← ServDoHandle
MyServFunc ≜ _MyServFunc!Spec
```

Container specification.

Variables:

SCCPUMinExecTime – records for each task the minimum amount of execution time it has been allocated over all periods so far.
SCCPUAssignedTo – holds the number of the task currently assigned the resource.

SCCmpInState – the state in which the component currently is.
SCCmpUnhandledRequest – TRUE if the environment put another request into the system.
SCCmpLastExecutionTime – the execution time of the last service execution.

SCEnvLastDeltaTime – The amount of time between the last two requests.
SCEnvInState – Current state of the service invoked.
SCEnvUnhandledRequest – TRUE signals that a new request has been put into the system.

SCServLastResponseTime – the response time of the last request serviced.
SCServInState – the current state of the service machinery.
SCServUnhandledRequest – TRUE indicates the arrival of a new request.

```
VARIABLES SCCPUMinExecTime, SCCPUAssignedTo
VARIABLES SCCmpInState, SCCmpUnhandledRequest
VARIABLES SCCmpLastExecutionTime, SCEnvInState,
VARIABLE  SCEnvLastDeltaTime
VARIABLE  SCEnvUnhandledRequest
VARIABLES SCServLastResponseTime, SCServInState,
VARIABLE  SCServUnhandledRequest

_MyContainer(ExecutionTimeConstr,
             ResponseTimeConstr,
             TaskCount, Periods,
```

```
                 Wcets) ≜
  INSTANCE SimpleContainer
   WITH
  ExecutionTime ← ExecutionTimeConstr,
  ResponseTime ← ResponseTimeConstr,
  CPUMinExecTime ← SCCPUMinExecTime,
  CPUAssignedTo ← SCCPUAssignedTo,
  CmpInState ← SCCmpInState,
  CmpUnhandledRequest ← SCCmpUnhandledRequest,
  CmpLastExecutionTime ← SCCmpLastExecutionTime,
  EnvLastDeltaTime ← SCEnvLastDeltaTime,
  EnvInState ← SCEnvInState,
  EnvUnhandledRequest ← SCEnvUnhandledRequest,
  ServLastResponseTime ← SCServLastResponseTime,
  ServInState ← SCServInState,
  ServUnhandledRequest ← SCServUnhandledRequest,
  CompFun ← MyCompFunc,
  CompModelMapping ← CompMap,
  ServFun ← MyServFunc,
  ServModelMapping ← ServMap

MyContainer(ExecutionTimeConstr,
            ResponseTimeConstr,
            TaskCount, Periods,
            Wcets)
    ≜ _MyContainer(ExecutionTimeConstr,
                   ResponseTimeConstr,
                   TaskCount, Periods,
                   Wcets)!Container
```

The complete system.

Variables:

TaskCount – the number of tasks to be scheduled on the *CPU* as determined by the container.
Periods – the periods to be scheduled for those tasks as determined by container.
Wcets – the worst case execution times to be considered when scheduling. As determined by the container.

```
VARIABLES CPUTaskCount, CPUPeriods, CPUWcets
VARIABLES SCTaskCount, SCPeriods, SCWcets

System ≜
  ∧ MyComponent
  ∧ MyCPU(CPUTaskCount,
          CPUPeriods,
          CPUWcets)
  ∧ MyContainer(20, 50,
                SCTaskCount,
                SCPeriods,
                SCWcets)
```

```
∧ □ ∧ ServResponseTime =
        SCServLastResponseTime
      ∧ ServInState = SCServInState
      ∧ ServUnhandledRequest =
        SCServUnhandledRequest
∧ □ ∧ MYCPU_MinExecTime =
        SCCPUMinExecTime
      ∧ MYCPU_AssignedTo = SCCPUAssignedTo
      ∧ CPUTaskCount = SCTaskCount
      ∧ CPUPeriods = SCPeriods
      ∧ CPUWcets = SCWcets
∧ □ ∧ SCCmpLastExecutionTime = MyCompExec
      ∧ SCCmpInState = MyCompInState
      ∧ SCCmpUnhandledRequest =
        MyCompUnhandledRequest
∧ □ ∧ SCEnvLastDeltaTime = EnvLastDeltaTime
      ∧ SCEnvInState = EnvInState
      ∧ SCEnvUnhandledRequest =
        EnvUnhandledRequest
```

The external behaviour we require of the system.

```
ExternalService
  ≜ Environment(RequestPeriod) -+▷ Service
```

This is the property we need to prove to ensure that we have a feasible system.

```
IsFeasible
  ≜ System ⇒ ExternalService
```

References

[1] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[2] Steffen Zschaler. *A Semantic Framework for Non-functional Specifications of Component-Based Systems*. PhD thesis, Technische Universität Dresden, Germany, April 2007.

[3] Steffen Zschaler. Formal specification of non-functional properties of component-based software systems: A semantic framework and some applications thereof. *Software and Systems Modelling (SoSyM)*, 2008. To appear.