ISSN 1361 - 6161

THE UNIVERSITY of MANCHESTER

Computer Science
University of Manchester

Comments on Specifying Reactive Systems

Pierre Collette

Department of Computer Science
University of Manchester
Technical Report Series
UMCS-96-8-2

Comments on Specifying Reactive Systems

Pierre Collette

Department of Computer Science
University of Manchester
Oxford Road, Manchester, UK.
collette@cs.man.ac.uk

1996

Abstract

This report comments on an experiment in using formal methods to specify and develop reactive systems. The goal is to discuss design decisions that impact on the complexity of the development. Two major design decisions are the choice of the starting point of the development and the choice of the interface when decomposing specifications. The language used in this case study is Lamport's Temporal Logic of Actions.

1 Introduction

Typically, a reactive system consists of a controller whose primary role is to maintain some interaction with its environment (generally a set of physical devices). Applying formal methods to the development of reactive systems is not a novel idea. A good starting point to the literature is [1]: an impressive number of formal methods have been applied to the same case study, and then compared.

The purpose of this report is not to apply yet another formal method to the same case study. Indeed, instead of analysing the pros and cons of a particular formalism, this report comments on the methodological aspects of writing and decomposing specifications of reactive systems. The case study can be explained in a few sentences and it will be shown that a formal development can progress without having to raise its complexity to unreasonable levels. In particular, a good choice of the starting point and of the interface in decomposing specifications considerably helps in keeping specifications relatively simple.

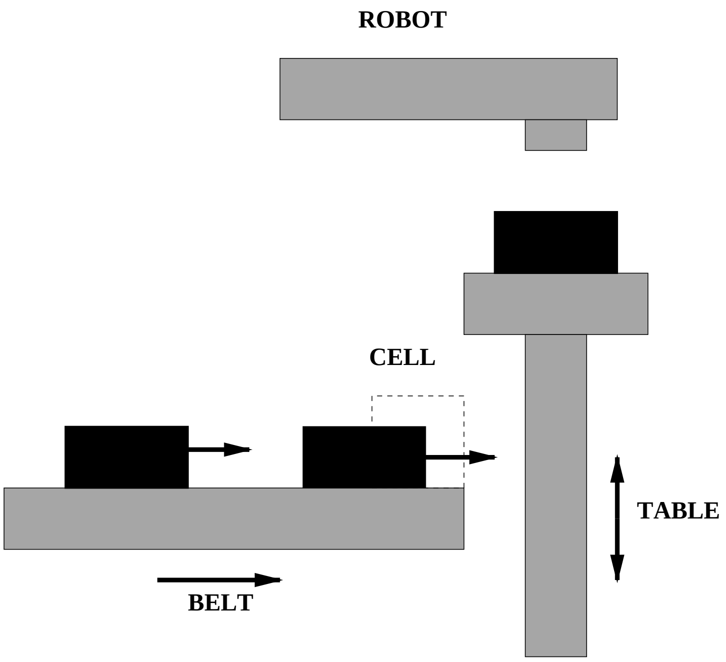
Each specification module is first presented with some explanatory notes; methodological comments and other discussions then follow.

2 Presentation of the case study

2.1 Informal description

The production cell is concerned with moving plates through a series of devices; a complete description can be found in [1]. This experiment (see figure) focuses on the elevating table which moves plates from a belt to a robot. More precisely, plates are delivered by the belt onto the elevating table, then lifted by the table, and finally picked up by a robot arm. To ensure safety, the table must be empty and correctly positioned in front of the belt when a plate is loaded. Similarly, it must be correctly positioned at the high position when a plate is unloaded. It is also required that the table should not move further than necessary. The obvious liveness requirement is that no new plate is delayed forever and that each one is eventually picked up by the robot.

The goal of the design is to write a controller which drives the belt and the table to meet those requirements. Control of the robot is not considered in this case study but of course some signalling between the robot and the table will have to be introduced. The interaction between between the controller and the physical devices consists of:



- An actuator by which the controller can start/stop moving the belt
- An actuator by which the controller can start/stop moving the table (upwards or downards).
- A sensor which gives the position of the table.
- A sensor (photoelectric cell) placed at the end of the belt. This photoelectric cell switches on when a plate intercepts the light ray. Just after the plate has completely passed through it, the cell switches off. It can be assumed that no new plate arrives in front of the photoelectric cell before the previous one has left the belt.

Before proceeding, the reader might try to sketch a program that reads the sensors ands write the actuators and then provide convincing arguments for its correctness.

2.2 The specification language

Lamport's Temporal Logic of Actions (TLA) [2] is a logic with a formal syntax and a formal semantics designed to specify reactive and concurrent systems. TLA^+ is a (draft) specification language equipped with a number of facilities to present TLA formulas in a structured way. All notations regarding TLA and TLA^+ will be explained on first occurence. Evaluating TLA is however out of the scope of this report.

2.3 Brief outline of the development

First, the expected behaviour of the production cell is specified in Section 3. That initial specification considers the production cell as a whole and does not distinguish between what can be assumed from the existing environment (physical devices) and what has to be enforced by the controller under development. The next stage of the development is the identification of an interface between controller and environment entities; that interface is specified in Section 4. The individual components of the environment and the controller are then specified in Section 5. That decomposition requires an additional assumption which is specified in Section 6. The correctness of the decomposition is asserted in Section 7.

Proofs. No formal proof has been carried out in this case study. Therefore, all formulas appearing in the **theorems** section of a TLA$^+$ module should be considered as conjectures. This paper is mainly concerned about methodological comments on writing specifications and the cost of carrying out all proofs by hand is therefore not justified.

3 Global specification

3.1 Modules

A TLA$^+$ specification module essentially consists of a set of definitions whose syntax and semantics are explained below. The horizontal bar is purely decorative and has no special meaning. Keywords (e.g. **parameters**, **assumptions**, **theorems**) identify special sections whose scope is given by the indentation. By convention, long conjunctions (disjunctions) are presented in a list format with $\wedge$ ($\vee$) appearing in front of each line.

module *Global*

parameters

End, OnT, Pos, Out : VARIABLE

H, L : CONSTANT

$New \triangleq End = 0 \wedge End' = 1$

$Load \triangleq Pos = L \wedge OnT = empty \wedge OnT' = one \wedge End = 1 \wedge End' = 0$

$Take \triangleq Pos = H \wedge OnT = one \wedge OnT' = empty \wedge Out' = Out + 1$

$Up \triangleq L \leq Pos < Pos' \leq H \wedge OnT = one$

$Down \triangleq H \geq Pos > Pos' \geq L \wedge OnT = empty$

$Init \triangleq End = 0 \wedge Pos = L \wedge OnT = empty \wedge Out = 0$

$$Next \triangleq \begin{array}{l} \wedge \Box[New \vee Load]_{End} \wedge \Box[Take]_{Out} \\ \wedge \Box[Load \vee Take]_{OnT} \wedge \Box[Up \vee Down]_{Pos} \end{array}$$

$Fair \triangleq \mathrm{WF}_{End}(Load) \wedge \mathrm{WF}_{Pos}(Up) \wedge \mathrm{WF}_{Pos}(Down) \wedge \mathrm{WF}_{Out}(Take)$

$Spec \triangleq Init \wedge Next \wedge Fair$

Global specification. The specification of the expected behaviour of the production cell is given by the TLA formula *Global.Spec*, which is simply referred to as *Spec* when the name of the module is clear from the context. The formula *Spec* is the conjunction of three TLA formulas:

- *Init* gives initial values to the system variables.
- *Next* specifies how the state may change.
- *Fair* specifies how the state must change.

States and computations. All formulas in module *Global* are expressed in terms of the variables and constants listed in the parameters section. In this example, the parameters section gives the list of the observable entities of the system:

- *End* (photoelectric cell) detects the presence of a plate at the very end of the feed belt. *End* switches on on arrival of a new plate and switches off exactly when the plate leaves the belt.
- OnT (On Table) is concerned with the presence of plates on the table. The only two expected values are *empty* and *one*; $OnT = empty$ models the situation where no plate is on the table; $OnT = one$ models the situation where there is is exactly one plate on the table and this plate is correctly positioned.
- *Pos* represents the position of the table. Its expected range is between L (Low position) and H (High position).
- *Out* counts the number of plates removed from the table by the robot.

The presentation could be improved if $End = 0$ and $End = 1$ were replaced by $\neg End$ and *End* respectively but TLA variables and constants are untyped and there are therefore no 'boolean' variables. Explicit type assumptions must be stated when proving theorems (see module *ThmGlobal*).

A state is a mapping from variables to values. A computation is an infinite sequence of states $\langle s_0 s_1 s_2 \ldots \rangle$. Temporal formulas (expressed in TLA) are evaluated on computations. For instance, a TLA formula might specify that x is never decreased ($s_k(x)$ is greater or equal to $s_j(x)$ for all $k \geq j$). Specifying a system by a TLA formula φ thus amounts to specifying the set of computations that satisfy φ.

States (hence computations) are global in that they give values to all variables of the universe. However, a TLA formula in a module constrains only the variables listed in the parameter section of that module (or of an imported module); in other words, it constrains only the part of the universe a module is concerned with.

State functions and predicates. Expressions are built from variables, constants, and operators. The only type distinction in TLA is between boolean expressions (predicates) and non-boolean expressions (state functions). Semantically, state functions and predicates map states to non-boolean and boolean values respectively. For instance, the state function $x + 3$ maps s to three plus the value of x in s and the state predicate $x > y$ maps s to true ('holds' at state s) if and only if the value of x in s is strictly greater than the value of y in s. In module *Global*, $Out + 1$ and the tuple (Pos, OnT) are examples of state functions and $End = 1$ is an example of a predicate.

Initial conditions. State predicates can be lifted to TLA formulas: $\langle s_0 s_1 \ldots \rangle$ satisfies P if $P(s_0)$ is true. For instance, *Init* specifies that there are initially no plates and that the table is initially in the low position.

Actions. An action is a boolean-valued expression containing primed and unprimed variables. Semantically, actions map pairs of states to boolean values, with primed variables referring to the second state. For instance, $x' > y \wedge x = 0$ is an action which maps (s, t) to true if and only if the value of x in t is strictly greater than the value of y in s and the value of x in s is zero. A pair of states that is mapped to true by A is called an A step. Module *Global* identifies the only five actions which are allowed in the system:

- *New*: arrival of a new plate at the end of the feed belt.
- *Load*: delivery of a plate from the feed belt to the table.
- *Take*: removal of a plate from the table.
- *Up*: movement of the table from the belt towards the robot.
- *Down*: movement of the table from the robot towards the belt.

For example, the action *Load* is formally described by a change in *End* from 1 to 0, a change in *OnT* from *empty* to *one*, and the condition that the table is in the low position ($Pos = L$).

Actions in temporal formulas. The most commonly used TLA formula is $\Box[A]_V$ where A is an action and V is a tuple of variables (hence a state function). A computation $\langle s_0 s_1 s_2 \ldots \rangle$ satisfies $\Box[A]_V$ if and only if every step (s_k, s_{k+1}) is an $A \vee V' = V$ step ($k \geq 0$); in other words, every step that modifies a variable in V must be an A step. There are four such formulas in *Next*. The first one asserts that *End* changes by *New* or *Load* actions only; the second one asserts that *Out* changes by *Take* actions only; the third one asserts that *OnT* changes by *Load* or *Take* actions only; the last one asserts that *Pos* changes by *Up* or *Down* actions only.

Fairness. The expected behaviour of a system is not only specified by what may happen (actions) but also by what must happen. This is the role of the formula *Fair*. The TLA formula $\mathrm{WF}_f(A)$ asserts that the action $A \wedge f' \neq f$ must eventually happen if continuously enabled (weak fairness); an action A is enabled at state s if there exists a state t such that (s, t) is an A step. In this case, the specification asserts that the *Load*, *Up*, *Take*, and *Down* actions must eventually happen if continuously enabled.

Assumptions and theorems. Module *ThmGlobal*, which imports module *Global* collects theorems on the global specification; in this simple case, importing just means that module *Global* is copied into module *ThmGlobal*. The first theorem asserts that every plate which arrives at the end of the feed belt is eventually carried away by the robot. This theorem is valid (conjecture) under the assumptions that Out,L,Pos,H are natural numbers and that L is strictly smaller than H. A formal proof of that theorem would be rather complex because the fairness requirements (which entail progress) must be combined with the safety requirements (which limit progress to certain directions). The second theorem shows that concurrency can be represented by conjunction.

module *ThmGlobal*

import
Global

$$Belt \triangleq End = 0 \wedge \Box[New \vee Load]_{End} \wedge \mathrm{WF}_{End}(Load)$$
$$Table \triangleq \wedge\ Pos = L \wedge OnT = empty \wedge \mathrm{WF}_{Pos}(Up) \wedge \mathrm{WF}_{Pos}(Down)$$
$$\qquad \wedge\ \Box[Load \vee Take]_{OnT} \wedge \Box[Up \vee Down]_{Pos}$$
$$Robot \triangleq Out = 0 \wedge \Box[Take]_{Out} \wedge \mathrm{WF}_{Out}(Take)$$

assumptions
$$Out \in Nat \wedge L \in Nat \wedge Pos \in Nat \wedge H \in Nat \wedge L < H$$

theorems
$$Thm_1 \triangleq \forall k \in Nat : Spec \Rightarrow \Box(End = 1 \wedge Out = k \Rightarrow \Diamond(Out = k + 1))$$
$$Thm_2 \triangleq Spec \Leftrightarrow Belt \wedge Table \wedge Robot$$

Sequential ordering. The sequential ordering of actions in not explicit in a TLA$^+$ specification: there is no ';' operator. Some ordering is however imposed by the enabledness condition of actions (e.g. $Pos = L \wedge OnT = empty \wedge End = 1$ for *Load*). In this example, the constraint on *Pos* and *OnT* in *Next* imposes the actions *Load*, *Up*, *Take*, and *Down* to occur in that order. In particular, *Down* cannot occur just after *Up* because *Down* requires $OnT = empty$ whereas *Up* maintains $OnT = one$. When necessary, sequential ordering can be otherwise imposed by introducing auxiliary variables; examples will be given in Section 5.

Concurrency. This portion of the production cell, which focuses on the table, is mainly sequential; other devices up or down the production line may operate concurrently but those are not specified in module *Global*. The only source of concurrency in this example is the arrival of new plates on the feed belt while the table is busy transferring the previous plate: *Spec* allows the action *New* to occur concurrently with *Take*, *Up*, or *Down*. Concurrency is not explicit: there is no $\|$ operator but, as shown by theorem Thm_2, concurrency is captured by conjunction.

3.2 Comments

Choice of the observables. The choice of the observables is dictated by what must be expressed in the specification. This case study focuses on the coordination of the elevating table (Pos and OnT) with its neighbouring devices: the belt (End) and the robot (Out). Introducing End implies that stopping the belt before a plate falls onto the floor (if the table is up) is part of the problem considered in this case study; indeed, End will change only when the belt is in motion. Of course, one can suppress the observable Out and view the robot as a device which just performs the action $OnT = one \wedge OnT' = empty$ but the presence of Out allows properties such as Thm_1 to be expressed.

Note that OnT is not a 'boolean' variable. That '*empty*' and '*one*' are the only two possible values is a property of *Global.Spec* but the correctness of the controller with respect to *Global.Spec* will have to be proved ! This issue is further discussed in Section 6.

Application-oriented specification. *Spec* does not refer to controller or 'interface' entities: it is entirely expressed in terms of environment entities. Using observables from the real world makes the specification easier to validate by the customer of the development: the customer has a better understanding of the real world he wants to be controlled than of the 'program' that will control it. Referring to environment entities narrows the gap between the informal and the formal specification.

Global specification. *Spec* does not distinguish between what can be assumed from the devices and what must be guaranteed by the controller. For instance, it does not say whether $Pos \leq H$ is ensured by physical safety equipment that prevent the table for going higher than position H or by the controller which has to send a signal to stop a motor in due time; the specification only says that moving up ($Pos' > Pos$) only happens when $Pos \geq L$ and $Pos' \leq H$. Who is responsible for an action to occur or not is of no concern; only the expected overall behaviour is specified. This strategy explains why the specification *Spec* is quite simple (to someone accustomed to TLA notations, of course).

Joint actions. The specification *Spec* reflects the observation that the processing of each plate in the cell is a sequence of joint actions. For instance, the delivery of a plate onto the table is a joint action by the belt, the table, and their controller(s). The presence of joint actions and the resulting simplicity of *Spec* is another consequence of the strategy of specifying the system globally. Any attempt to split joint actions into a combination of actions from the individual physical devices and actions from the controller(s) would result in a much more sophisticated specification. As a matter of fact, this would amount to solving the most important part of the problem (i.e. the synchronisation part) at the very first specification level. This does not fit with the philosophy of specifying the problem rather than the solution first.

No unexpected behaviours. Unexpected behaviours are of no concern in this specification: what happens in unwanted cases is not specified. For instance, *Spec* says nothing about the case where a plate is delivered from the feed belt when the table is not ready (e.g. is moving up or has another plate on it). Suppose the development ends up in a detailed specification *Detail* from which an implementation can be easily derived. Then, the correctness of the development, i.e. $Detail \Rightarrow Spec$, ensures that only the actions *New*, *Load*, *Take*, *Up*, and *Down* will ever occur and there is therefore no reason to worry about actions that will not happen. The burden of a formal development should not be in specifying all possible cases but in developing a detailed specification which entails the specified expected behaviour.

4 Logical interface

4.1 Modules

As discussed further below, the logical decomposition of a specification does not necessarily correspond to the physical decomposition of a system into its components. In the specification *Composition.Spec*, which refines *Global.Spec*, the system is specified by a synchroniser (module *Sync*) that communicates with monitored devices (modules *MBelt*, *MTable*, and *MRobot*).

A monitored device can be viewed as a physical device (e.g. table) equipped with some piece of hardware/software that provides services (e.g. move the table to position H) to the synchroniser. Requests and acknowledgments are modelled by flags. For instance, the synchroniser requests the table to be moved up to position H by setting the flag FU; FU is reset by the monitored belt when the table has reached position H. In this case, the flags and their associated services are:

- FB (Flag to Belt): deliver a new plate on the table.
- FU (Flag to go Up): move the table up to the high position.
- FD (Flag to go Down): move the table down to the low position
- FR (Flag to Robot): take a plate from the table.

The first service is provided by the monitored belt; the next two services are provided by the monitored table; the last service is provided by the monitored robot.

The specification of the synchroniser is combined (conjunction) with the specification of the monitored devices in module *Interaction*; all included modules will be given in the subsequent sections. The specification *Composition.Spec* that refines *Global.Spec* and from which subsequent development should continue is obtained from *Interaction.Spec* by abstracting from the communication mechanism: the variables FB, FU, FD, and FR are existentially quantified (hiding: see comments below).

module *Flag*

$IsFlag(b) \triangleq b \in \{0, 1\}$
$Reset(b) \triangleq b = 1 \wedge b' = 0$
$Set(b) \triangleq b = 0 \wedge b' = 1$

module *Interaction*

parameters
End, OnT, Pos, Out : VARIABLE
H, L : CONSTANT
FB, FU, FD, FR : VARIABLE

import
Flag

include
Sync **as** S, *AddHypo* **as** A
MBelt **as** B, *MRobot* **as** R, *MTable* **as** T

$Init \triangleq FB = 0 \wedge FU = 0 \wedge FD = 0 \wedge FR = 0$
$Spec \triangleq Init \wedge A.Spec \wedge B.Spec \wedge T.Spec \wedge R.Spec \wedge S.Spec$

module *Composition*

parameters
End, OnT, Pos, Out : VARIABLE
H, L : CONSTANT

include
Interaction **as** $IT(FB, FU, FD, FR)$

$Spec \triangleq \exists FB, FU, FD, FR : IT(FB, FU, FD, FR).Spec$

Definition parameters. In addition to the variables and constants listed in the parameters section, a definition in a module may also refer to definition parameters. A typical example is b in module *Flag*.

Included modules. Suppose that a module *InMod* has parameters a and b, and defines $\mathit{Form}(x) \triangleq \varphi(x, a, b)$. Then, the effect of

$$\textbf{include}\ \mathit{InMod}\ \textbf{as}\ \mathit{In}(a)$$

in module M (of which b is a parameter) is to add the definition

$$\mathit{In}(a).\mathit{Form}(x) \triangleq \varphi(x, a, b)$$

therefore turning a into a definition parameter. In this example, *B.Spec* in the definition of *Interaction.Spec* refers to the formula *Spec* of module *MBelt*. Because module parameters cannot be quantified (they are the visible entities), the include mechanism in module *Composition* turns the communication variables into definition parameters, thus ensuring that the specification is over the *same* visible variables as in module *Global*.

Existential quantification. Existential quantification in TLA obeys the same laws as existential quantification in predicate logic. Intuitively, a computation σ satisfies the TLA formula $\exists\, x : \varphi$ if there exists a computation σ' that satisfies φ such that σ' can be obtained from σ by changing the assignments to x only[1].

4.2 Comments

Choice of the interface. The choice of an appropriate interface is always critical because a poor choice leads to unnecessarily complex specifications. Consider an implementation in which a centralised controller interacts with the physical devices by sensors and actuators and suppose that the controller consists of a main routine plus one interface routine for each device. The structure of the specification above then captures the following decomposition:

Synchroniser	=	Main Routine
Monitored Device	=	Physical Device + Interface Routine

Another way to structure a specification would be to follow the 'natural' division defined by the sensors and the actuators:

Controller	=	Main Routine + Interface Routines
External World	=	Physical Devices

[1]The exact definition allows σ and σ' to differ by their stuttering steps.

This however would make the specifications much more complicated than necessary at this stage because the interface between the controller and the external world (sensors and actuators) is a very low-level one. The major concern at this stage of the development, which is the synchronisation of the services, would then be buried in the details of specifying how and when each actuator should change in response to every change in a sensor. In particular, all real-time requirements associated with a low-level interface would have to be included (see also the discussion in Section 5). Of course, the low-level synchronisation problems must eventually be solved but this is better done locally, in the subsequent development of each monitored device.

Logical view of communications. The existential quantification in *Composition* means that the introduction of the variables FB, FU, FD, and FU is not a design decision but only a way to represent the synchronisation at the specification level. These variables might not exist in the actual implementation. Suppose for instance that each monitored device is composed of the actual physical device plus some hardware to implement the services. The communications between the monitored devices and the synchroniser might then consist of hardware interrupts: the meaning of $FX = 1$ is 'an interrupt from the synchroniser has occurred' and the meaning of $FX = 0$ is 'an interrupt from the monitored device has occurred'. The existential quantification means that an implementation is correct if *there exists* some way of retrieving those variables so that *Composition.Spec* holds.

Logical versus physical interface. The existential quantification also means that the structure of the implementation might differ from the structure of the specification. Of course, this specification suggests an implementation where the services are implemented by either hardware components or interface routines. But the synchronisation and the implementation of the services might also be entangled in a single centralised routine which at each cycle reads the sensors, performs some computation, and then writes the actuators. That implementation would probably be more difficult to verify against *Composition.Spec* but the specification only says that it should be possible to retrieve the variables FB, FU, FD, and FR from the implementation; this can be done by augmenting the centralised routine with assignments to these variables.

5 Specification of the components

5.1 Modules

The specification modules for the monitored devices and for synchroniser are now presented. Those modules are imported by module *Interaction*.

Monitored table. *Up* and *Down* are the only two actions that can modify the position of the table. Those actions are enabled only when the synchroniser has requested the table to be moved up ($FU = 1$) or down ($FD = 1$) respectively. The monitored table signals the end of a movement by resetting the corresponding flag (actions *AckUp* and *AckDw*).

module *MTable*

parameters
  Pos, FU, FD : VARIABLE

import
  Flag

$$Init \triangleq Pos = L$$
$$Up \triangleq FU = 1 \land Pos < Pos' \leq H$$
$$Down \triangleq FD = 1 \land Pos > Pos' \geq L$$
$$AckUp \triangleq Pos = H \land Reset(FU)$$
$$AckDw \triangleq Pos = L \land Reset(FD)$$

$$\begin{aligned} Spec \triangleq\ & \land\ Init \land \Box[Up \lor Down]_{Pos} \\ & \land\ \Box[Reset(FU) \Rightarrow AckUp]_{FU} \land \Box[Reset(FD) \Rightarrow AckDw]_{FD} \\ & \land\ \mathrm{WF}_{Pos}(Up \lor Down) \land \mathrm{WF}_{FU}(AckUp) \land \mathrm{WF}_{FD}(AckDw) \end{aligned}$$

Monitored belt. The only two actions that can modify *End* are *New* (arrival of a new plate) and *Load* (transfer of a new plate from the belt to the table). The *Load* action only occurs if a new plate has been requested by the synchroniser ($FB = 1$) and the transfer is then confirmed by resetting FB.

The use of an existential quantifier shows that x is an auxiliary variable that must be definable from the implementation of the monitored belt. Its only purpose in the specification *MBelt.Spec* is to impose the actions *New*, *Load*, and *Ack* to occur in that sequential order (the ordering between *New* and *Load* is imposed by *End*). This guarantees that only one plate is delivered by the belt before FB is reset.

Monitored robot. The only action that can modify *Out* is *Take*. This action occurs only when the removal of a plate is requested by the synchroniser ($FR = 1$). As for the monitored belt, the existentially quantified variable imposes an ordering between the actions: *Ack* must occur after *Take*.

Synchroniser The synchroniser initiates a phase by setting the appropriate flag. It waits for the previous phase to be completed before launching the next one. The order between phases is specified by the auxiliary variable p.

module *MBelt*

parameters

End, FB : VARIABLE

import

Flag

$$Init(x) \triangleq End = 0 \land x = 0$$
$$New(x) \triangleq x = 0 \land End = 0 \land End' = 1$$
$$Load(x) \triangleq FB = 1 \land End = 1 \land End' = 0 \land Set(x)$$
$$Ack(x) \triangleq Reset(FB) \land Reset(x)$$

$$\begin{aligned} LocSpec(x) \triangleq\ & \land\ Init(x) \land \Box[New(x) \lor Load(x)]_{End} \land \Box[Load(x) \lor Ack(x)]_x \\ & \land\ \Box[Reset(FB) \Rightarrow Ack(x)]_{FB} \\ & \land\ \mathrm{WF}_x(Load(x) \lor Ack(x)) \end{aligned}$$
$$Spec \triangleq \exists\, x : LocSpec(x)$$

module *MRobot*

parameters

Out, FR : VARIABLE

import

Flag

$$Init(y) \triangleq Out = 0 \land y = 0$$
$$Take(y) \triangleq FR = 1 \land Out' = Out + 1 \land Set(y)$$
$$Ack(y) \triangleq Reset(FR) \land Reset(y) \land Out' = Out$$

$$\begin{aligned} LocSpec(y) \triangleq\ & \land\ Init(y) \land \Box[Take(y) \lor Ack(y)]_{y, Out} \\ & \land\ \Box[Reset(FR) \Rightarrow Ack(y)]_{FR} \\ & \land\ \mathrm{WF}_y(Take(y) \lor Ack(y)) \end{aligned}$$
$$Spec \triangleq \exists\, y : LocSpec(y)$$

5.2 Comments

Component specification. Unlike the global specification in module *Global*, each of these specifications concerns one component only. This means that only the actions performed by that component must be constrained. Consider *MTable.Spec*: this TLA formula only constrains state transitions where *Pos* has been changed or one of the two flags has been reset. This formula does not constrain flag setting. Since no other variable

module *Sync*

parameters
FB, FU, FD, FR : VARIABLE

$$Init(p) \triangleq p = 0$$
$$Load(p) \triangleq p = 0 \land FD = 0 \land Set(FB) \land p' = 1$$
$$Up(p) \triangleq p = 1 \land FB = 0 \land Set(FU) \land p' = 2$$
$$Take(p) \triangleq p = 2 \land FU = 0 \land Set(FR) \land p' = 3$$
$$Down(p) \triangleq p = 3 \land FR = 0 \land Set(FD) \land p' = 0$$

$$Next(p) \triangleq Load(p) \lor Up(p) \lor Take(p) \lor Down(p)$$
$$\begin{aligned} LocSpec(p) \triangleq\ & \land\ Init(p) \land \Box[Next(p)]_p \land \mathrm{WF}_p(Next(p)) \\ & \land\ \Box[Set(FB) \Rightarrow Load(p)]_{FB} \land \Box[Set(FU) \Rightarrow Up(p)]_{FU} \\ & \land\ \Box[Set(FR) \Rightarrow Take(p)]_{FR} \land \Box[Set(FD) \Rightarrow Down(p)]_{FD} \end{aligned}$$
$$Spec \triangleq \boldsymbol{\exists}\, p : LocSpec(p)$$

is constrained by *Table.Spec*, actions from other components on other variables may occur concurrently (e.g. *MBelt.New*).

The action *Table.Up* should be contrasted with the action *Global.Up* which is enabled only when $OnT = one$. Indeed, the table moves up whenever asked for, no matter whether there is something on it. Similarly, the action *Belt.Load* should be contrasted with the action *Global.Load* which is enabled only when the table is ready to accept a new plate ($Pos = L$ and $OnT = empty$). Indeed, a plate will leave the belt when it arrives at the end of it, no matter whether the table is ready to accept it or not.

Real-time requirements. Time critical properties can be found in about every single reactive system. This example is not an exception: if the motor driving the elevating table is not switched off δ seconds after the position $H - \epsilon$ has been detected by a sensor, then the table will crash with the robot. Should this problem be considered ? Certainly. Should it be formally specified ? Perhaps. Should it be formally specified at this stage of the development ? No. There is no need to pay for the extra complexity of using a real-time language at this stage. That real-time requirement should appear only when/if the specification of the monitored table is decomposed into a specification of the elevating table and a specification of the interface routine (or hardware) that implements the services.

The relative simplicity of *MTable.Spec* due to the absence of real-time requirements is another consequence of putting the interface at the right place when dividing the system.

Naming phases. Intuitively, the four phases of the synchroniser correspond to the table being stopped at the low position, moving up, being stopped at the high position, and moving down. The controller variable p is thus an estimation of the position of the table. There is however a difference between the position of a device and an estimation of that position by the controller. For example, it can be proved that $Pos = H$ whenever $p = 3$ but the converse does not hold because $Pos = H \wedge p = 2$ holds between the actions *MTable.AckUp* (termination of a movement) and *Sync.Take*. The correctness of the development (*Composition.Spec* implies *Global.Spec*) shows that the synchroniser does not need a more accurate estimation: this is a proved fact and not a general assumption about the formal method in use.

6 An additional assumption

6.1 Module

module *AddHypo*

parameters

Out, End, Pos, OnT : VARIABLE

$$
\begin{aligned}
Arrive &\triangleq End = 1 \wedge End' = 0 \\
Leave &\triangleq Out' = Out + 1 \\
Spec &\triangleq \wedge\ OnT = empty \wedge \Box[Arrive \vee Leave]_{OnT} \\
&\quad\ \wedge\ \Box[Arrive \wedge OnT = empty \wedge Pos = L \Rightarrow OnT' = one]_{End} \\
&\quad\ \wedge\ \Box[Leave \wedge OnT = one \wedge Pos = H \Rightarrow OnT' = empty]_{Out}
\end{aligned}
$$

Modules *MBelt*, *MTable*, *MRobot*, and *MSync* fail to capture an essential property of the devices. Without the module *AddHypo* which is included in module *Interaction*, the development of *Global.Spec* into *Composition.Spec* would be incorrect because none of the modules specified so far is concerned with the variable OnT.

The first conjunct of *AddHypo.Spec* asserts that there are initially no plates on the table and that OnT can only change when the table is loaded or unloaded: there is no external agent picking up or removing plates from the table. The second and third conjuncts of *Spec* then force OnT to take the correct value if the table is ready when a plate is delivered by the belt or taken away by the robot.

6.2 Comment

The formula *Spec* is an assumption on OnT only. It asserts that OnT changes only when a plate is delivered by the belt or taken away by the robot but those actions are specified in other modules (*MBelt* and *MRobot* respectively).

This assumption has to be validated against the physical environment. This can be done independently (without concern about other modules) because of the use of $\Rightarrow$ in the second and third conjuncts of *Spec*: one only has to validate the right-hand side assuming the left-hand side, given the following meaning for OnT:

- $OnT = one$: there is exactly one plate on the table and this plate is correctly positioned.

- $OnT = empty$: there is no plate on the table.

No 'error-case' has to be considered. *Spec* says nothing about changes to OnT if e.g. a plate is delivered from the belt when the table is in the high position or if there is already one plate on the table. Those situations might be represented by $OnT = error$ but there is no need to introduce error values hence extra complexity because $OnT \in \{empty, one\}$ can be *proved* to be an invariant of *Composition.Spec*. $OnT \in \{empty, one\}$ is not a type assumption but a proved fact, established from the specification of the synchronizer, the specification of the monitored device, and an additional assumption on the environment.

7 Verification

The correctness of the refined specification *Composition.Spec* with respect to the initial specification *Global.Spec* is asserted in Theorem *CompoCorrect* of module *Verif*. That theorem is a consequence of Theorem *LocalCorrect* and of the laws for existential quantification. Correctness proofs are usually carried out by first finding an invariant of the refined system, given by the predicate *Inv* in this case.

8 Conclusion

Clearly, the comments on this development only represent the author's opinions and are therefore subjective. The size of the case study is also a limiting factor: there are only a few realistically sized applications of formal methods and many more are needed before guidelines can be issued.

One should however keep in mind that this case study looks smaller and simpler than it really is, precisely because of the development strategy. Consider again the problem of loading a plate from the belt onto the table. A direct implementation of a controller from the informal specification is not trivial. Indeed, one has to read the position of the table and the status of the photoelectric cell (at a sufficiently high frequency) and then send signals to start/stop the table and belt motors, taking into account that a device does not stop instantaneously. This is a typical synchronisation problem involving real-time aspects.

Two major decisions helped in mastering the complexity of the development. First, the development did not start by specifying a program for driving the table and belt motors. Instead, the system has been considered globally and the initial specification

module *Verif*

parameters

End, OnT, Pos, Out : VARIABLE
FB, FU, FD, FR : VARIABLE
H, L : CONSTANT

include

Interaction **as** I, *Global* **as** G,
MBelt **as** M, *MRobot* **as** R, *MTable* **as** T, *Sync* **as** S, *AddHypo* **as**A

$$
\begin{aligned}
LocSpec(x, y, p) \;\triangleq\; & \wedge\ I.Init \wedge A.Spec \wedge T.Spec \\
& \wedge\ B.LocSpec(x) \wedge R.LocSpec(y) \wedge S.LocSpec(p)
\end{aligned}
$$

$$
\begin{aligned}
Inv(x, y, p) \;\triangleq\; & \wedge\ \forall b \in \{FB, FU, FD, FR, x, y\} : b \in \{0, 1\} \\
& \wedge\ OnT \in \{empty, one\} \wedge End \in \{0, 1\} \\
& \wedge\ p \in \{0, 1, 2, 3\} \wedge L \leq Pos \leq H \wedge Out \geq 0 \\
& \wedge\ (FB = 1 \vee x = 1 \Rightarrow p = 1) \wedge (FR = 1 \vee y = 1 \Rightarrow p = 3) \\
& \wedge\ (FU = 1 \Rightarrow p = 2) \wedge (FD = 1 \Rightarrow p = 0) \\
& \wedge\ (p = 1 \Rightarrow Pos = L) \wedge (p = 2 \wedge FU = 0 \Rightarrow Pos = H) \\
& \wedge\ (p = 3 \Rightarrow Pos = H) \wedge (p = 0 \wedge FD = 0 \Rightarrow Pos = L) \\
& \wedge\ (p = 2 \Rightarrow OnT = one) \wedge (p = 0 \Rightarrow OnT = empty) \\
& \wedge\ p = 1 \Rightarrow \vee\ x = 0 \wedge FB = 1 \wedge OnT = empty \\
& \qquad\qquad\quad \vee\ x = 1 \wedge FB = 1 \wedge OnT = one \\
& \qquad\qquad\quad \vee\ x = 0 \wedge FB = 0 \wedge OnT = one \\
& \wedge\ p = 3 \Rightarrow \vee\ y = 0 \wedge FR = 1 \wedge OnT = one \\
& \qquad\qquad\quad \vee\ y = 1 \wedge FR = 1 \wedge OnT = empty \\
& \qquad\qquad\quad \vee\ y = 0 \wedge FR = 0 \wedge OnT = empty
\end{aligned}
$$

assumptions

$Out \in Nat \wedge L \in Nat \wedge Pos \in Nat \wedge H \in Nat \wedge L < H$

theorems

$IsInv \;\triangleq\; LocSpec(x, y, p) \Rightarrow \Box Inv(x, y, p)$
$LocalCorrect \;\triangleq\; LocSpec(x, y, p) \Rightarrow G.Spec$
$ELaw \;\triangleq\; I.Spec \Leftrightarrow \exists\, x, y, p :: LocSpec(x, y, z)$
$CompoCorrect \;\triangleq\; C.Spec \Rightarrow G.Spec$

just describes the expected behaviour, in terms of environment entities. The second major design decision is the choice of the interface in decomposing the specification. By introducing monitored devices, the real time aspects of the problem have been brushed away to focus on the synchronisation of the services. Concerns about the frequency of reading sensors and delays in response to changes in actuators are better dealt with locally. No implementations of the monitored devices have been given in this report, but there are techniques in control engineering to produce software/hardware which ensure that a device is moved to the requested position.

References

[1] C.Lewerentz and T.Lindner, eds., *Formal Development of Reactive Systems*, Lecture Notes in Computer Science, vol 891, 1995.

[2] L.Lamport, The Temporal Logic of Actions, *ACM Transactions on Programming Languages and Systems*, vol 17, 1995, pp 507–534.