Modeling Concurrent Systems with Shared Resources

Ángel Herranz[1], Julio Mariño[1], Manuel Carro[1], and Juan José Moreno Navarro[2]

[1] Universidad Politécnica de Madrid
[2] Spanish Ministry of Science and Innovation

Abstract. Testing is the more widely used approach to (partial) system validation in industry. The introduction of concurrency makes exhaustive testing extremely costly or just impossible, requiring shifting to formal verification techniques. We propose a methodology to design and verify a concurrent system that splits the verification problem in two independent tasks: internal verification of shared resources, where some concurrency aspects like mutual exclusion and conditional synchronisation are isolated, and external verification of processes, where synchronisation mechanisms are not relevant. Our method is language independent, non-intrusive for the development process, and improves the portability of the resulting system. We demonstrate it by actually checking several properties of an example application using the TLC model checker.
Keywords: Validation, Verification, Shared resource, Concurrency.

1 Introduction

Concurrency is a key aspect in many software systems and often a reason for their failure as well, as programming concurrent systems is notoriously more difficult and error-prone than programming sequential systems. Almost every aspect the programming gets worse when several processes have to be considered at once: language constructs, library availability (and multi-thread safeness), debugging, runtime exceptions and their semantics, specification and verification, etc.

Although it is common for industrial software to restrict features that can be a potential source of hazards (e.g. by enforcing adherence to certain coding rule sets), many applications show some degree of implicit concurrency that cannot be ignored, or which brings about advantages which make them highly interesting. A recent example is the trend towards multi-task web browsers where every web page, or even every component in a web page, is handed out to a different thread in order to improve security and stability by sandboxing these threads.

Unfortunately, there is still much room for improvement in terms of methods, tools and language support for the development of concurrent software in industrial environments. To make things worse, software developers are in general insufficiently trained. The following paragraphs outline what we think are some of the most salient issues.

Good Language Support. While some languages deploy good support for concurrency, with some of them providing developers with platform-independent constructs (e.g. Ada, Java, C#...), there are still application niches where the use of languages less suited for concurrency, like C or C++, is mandatory. In these cases, concurrency is

only possible through the use of certain mechanisms with not so clear semantics and, sometimes, subject to change. Even when support is good, clear design guidelines / patterns are not in widespread use.

Methodology. There are no standard notations or tools to model concurrent systems which at the same time help developers in clearly documenting this aspect of software in isolation from other requirements — the concurrency equivalent for UML/OCL is not yet here. As a result, concurrency is often poorly documented and the chances that some concurrency requirements are lost are considerable.

Validation and verification. The inherent nondeterminism introduced by concurrency and the execution conditions often specific to the application itself (specially in an industrial environment) make standard testing techniques unreliable, more expensive, or directly inapplicable. This is one of the reasons to emphasise the use of formal techniques for the verification of this kind of software. Unfortunately, support for concurrency is still scarce in existing verification tools, which only deal with language subsets which do not include synchronisation and communication primitives. An ongoing European COST action on verification of O.O. software [2] places concurrency among one of the three major challenges of verification technology for O.O. languages, along with genericity and components.

Portability. Given the dependability often associated with industrial software, the risks (and costs) associated with making an upgrade or porting a running system are huge. Very often, systems of this kind become legacy code fossils that nobody dares to touch. The risk, then, is that any seemingly innocent change in the execution environment (hardware, operating system, running conditions...) may affect its behaviour in unpredictable ways.[3]

There probably no single solution for these problems. But it seems that promoting the separation of concerns when designing, providing developers with (graphical) notations that address concurrency at the same level as other aspects, isolating the code which depends on synchronisation and communication primitives from the rest, and giving hints for the practical verification of concurrent software are steps in the right direction.

Here, we are proposing an approach to the development of concurrent systems based on a sharp distinction between *active* (processes, clients) and *passive* (interactions, resources) entities. This admittedly simplistic view of concurrency will help us, however, in achieving some of the aforementioned goals. For example, synchronisation and communication primitives (often language-dependent) will appear only inside the implementation of the shared resources, not in the process code. This separation will make possible to work on the verification of the processes on relatively standard grounds. On the other hand, the implementation of the shared resources will be ensured to be correct *by construction*, using template-based code generation schemes both for shared

[3] And, in fact, it is the case that whole lines of hardware and tools (e.g., compilers) have been maintained for the sole purpose of keeping this kind of systems running untouched.

memory and message-passing schemes [1], rather than verified *a posteriori*. This enormously simplifies portability. Also, the method is supported by a graphical notation intended to be reminiscent of UML class and collaboration diagrams.

The rest of the paper is organised as follows: Section 2 gives an informal overview of the method and the notation by means of an example. Section 3 presents a translation of shared resources into TLA. This serves two purposes: on one hand, it provides an *interlingua* semantics, and, secondly, the translation is used as the basis for a practical method to check properties of a concurrent system using the TLC model checker (Section 4). Section 5 summarises our results, discusses related approaches, and points out to future improvements.

2 Specifying Shared Resources by Example

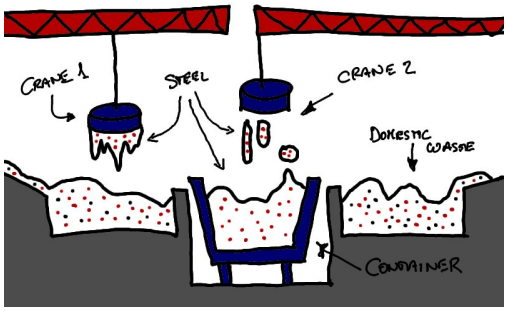

Fig. 1. Recycling plant.

In this section we will introduce an example which will be used throughout the rest of the paper and we will use this example to present our resource-oriented notation [1]. We will present the intended semantics of the specification language without formal apparatus but, hopefully, with clarity enough to justify its translation into TLA+ and to grasp the more relevant points of our notation.

2.1 The Recycling Plant Example

In a recycling plant (Fig. 1), steel is recovered from unsorted domestic waste with automatically controlled cranes equipped with electromagnets: cranes collect steel and deposit it in a container until the container is (nearly) full. The crane controller is accessed using a library with a public API, part of which appears next.

```
package Cranes is
   MAX_CRANES : constant Positive := 5;
   subtype Crane_Id is Positive range 1 .. MAX_CRANES;
   MIN_W_CRANE : constant Positive := 1000;
   MAX_W_CRANE : constant Positive := 1500;
   subtype Weight is Positive range MIN_W_CRANE .. MAX_W_CRANE;
   --  Grab the steel and report its weight
   procedure Collect (N : in Crane_Id; W : out Weight);
   --  Move the crane to a dropping point and
   --  deactivate the electromagnet.
   procedure Drop (I : Crane_Id);
end Cranes;
```

The container is also electronically controlled, and its relevant API is:

```
package Container is
   MAX_W_CONTAINER : constant Natural := 20000;
   --  Replace the current container with an empty one
   procedure Replace;
end Container;
```

Our aim is to specify and verify a concurrent system which controls the cranes and the container so that the cranes simultaneously collect steel and fill in the container without exceeding its maximum capacity, and replace full containers with empty ones, making sure that cranes do not try to deposit debris when containers are being replaced. We assume space enough for several cranes to deposit steel in the container without interacting with each other.

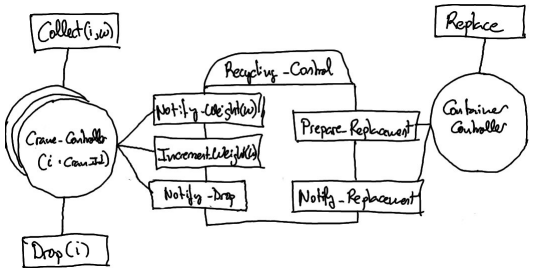

Fig. 2. System Design.

We assume that the system can be expressed as a collection of processes interacting through a shared resource (see Fig. 2), an instance of what we term a **CADT** (**C***oncurrent* **A***bstract* **D***ata* **T***ype*). Using a generalization of data abstractions as the base for concurrency puts the emphasis on the interaction with the environment instead of on internal organization and algorithms, and also separates the functionality and implementation (i.e., message passing vs. shared memory).

Unlike other proposals, our specification language does not capture process behavior, which is instead written directly in a very simple programming language (Section 2.3). In what follows we will give a brief account of the main characteristics of the specification language using the crane example.

2.2 Design of a Resource-Based Solution

We are not aiming at describing the design process itself here, as to some extent, this relies on experience and common sense.[4] We will instead present a finished design and describe how it is assumed to work.

A process is assigned to every crane and to the container. A shared resource will be used as central point for synchronization. The state of the shared resource is rich enough to to determine when a container change is needed and when it has been changed. In particular, it contains the *replacement state*, which can take the following values: *ready* (there is room for more waste), *to_replace* (a crane carries more steel than what the container can hold, and the crane has decided to order an empty container), and *replacing* (the container is being replaced). Even if the *to_replace* state is entered, a

[4] Although there are, of course, guidelines which help in removing from an early stage many clearly wrong designs or which help in moving towards arguably better designs.

crane carrying an amount of steel which still fits into the container can unload it as long as the *replacing* state has not been entered yet.

This approach does not maximize the container load, as the replacement process can start when some crane still carries a load which fits in the container. The alternative solution of storing the weight on every crane in the resource and making a central decision on which crane leaves its load was not chosen since it reduces concurrency.

Updates to the state need to be performed atomically. We define the resource as providing this atomicity for every operation, as well as more complex, data-dependent synchronization operations (Section 2.4).

2.3 Processes

Our starting point is to express the behavior of the system in terms of processes and then decide how they have to synchronize. As this is done exclusively by means of the shared resource, processes drive the design of the shared resource. As an less desirable but unavoidable side effect, this can result in resources with little reusability. We will see later how to detect the lack of certain reusability properties.

A crane process controller follow. Variable `Recycling_C` represents the shared resource and variable `I` (with `I` different for every process) identifies the crane managed by that process. The shared resource is represented by the object `Recycling` and all operations prefixed by it belong to the resource.

```
loop  --  Controller for the I-th crane
   Cranes.Collect (I, W);
   Recycling.Notify_Weight (Recycling_C, W);
   Recycling.Increment_Weight (Recycling_C, W);
   Cranes.Drop (I);
   Recycling.Notify_Drop (Recycling_C);
end loop;
```

`Notify_Weight` decides if the container has to be replaced. `Increment_Weight` suspends if the load cannot be unloaded because there is no space in the container or it is being replaced. Otherwise, it increments the container weight before actually dropping the steel. A counter of the cranes which have committed to unload but have not done it yet is kept to avoid the container to be replaced when some cranes are not yet through.

The container controller **waits** for the container to be replaceable, changes its state to *replacing*, replaces the container, and, atomically, changes the state to *ready* again.

```
loop  -- Container controller main loop
   Recycling.Prepare_Replacement (Recycling_C);
   Container.Replace;
   Recycling.Notify_Replacement (Recycling_C);
end loop;
```

2.4 Anatomy of a Specification

Our specification language is based on first-order logic, which is sufficiently known not to need but a quick brush-up in most cases. Its core ideas are inspired on a simplification of well-known formal methods (notably VDM [5]) with additional constructions to

CADT Recycling_Control
OPERATIONS
ACTION Prepare_Replacement: *Recycling_Control [io]*
ACTION Notify_Replacement: *Recycling_Control [io]*
ACTION Notify_Weight: *Recycling_Control [io]* × *Weight[i]*
ACTION Increment_Weight: *Recycling_Control [io]* × *Weight[i]*
ACTION Notify_Drop: *Recycling_Control [io]*

SEMANTICS
DOMAIN:
TYPE: *Recycling_Control = (weight:* $\mathbb{N}$× *state: State_Ty* × *accessing:* $\mathbb{N}$*)*
State_Ty = ready | to_replace | replacing
Weight = MIN_W_CRANE .. MAX_W_CRANE
INVARIANT: $\forall r \in Recycling\_Control \bullet r.weight \leq MAX\_W\_CONTAINER \wedge r.accessing \leq MAX\_CRANES$

CPRE: $r = (\_, to\_replace, 0)$
Prepare_Replacement(r)
POST: $r^{\text{out}} = (r^{\text{in}}.weight, replacing, 0)$

PRE: $w \leq MAX\_W\_CRANE$
CPRE: $r.weight + w \leq MAX\_W \wedge r.state \neq replacing$
Increment_Weight(r, w)
POST: $r^{\text{in}} = (cw, e, a) \wedge r^{\text{out}} = (cw + w, e, a + 1)$

Fig. 3. Partial CADT for the central crane controller.

address concurrency. Following [11], our specifications are state-based, with the state accessible only through a set of public operations. Fig. 3 shows a partial specification of the resource at hand, which we will explain in this section.

Public Interface: Actions and Their Signatures. The **OPERATIONS** section defines the names and signatures of the public operations and (optionally) tags arguments as input and/or output. The state of the resource itself is currently not directly available to the body of the specification; it must instead be a formal parameter (by convention, the first one) of every operation.[5]

Domain: Types. The *Domain* section contains type definitions for the resource and, optionally, an *invariant* which restricts the values of the resource state to those which are admissible in the problem.

In this example, the resource state contains the weight in the container, its replacement state (of enumerated type *State_Ty*), and how many cranes remain to deposit their load. `Weight` represents valid crane loads.

[5] This is not a strong requirement and is kept for compatibility with procedural languages.

Basic types include Booleans, naturals, integers, and reals, and complex types are built on them by means of algebraic types (free types). A series of predefined non-basic types, such as we also provide sequences (indexable flexible-length arrays), sets, and finite mappings, with a complete set of operations on them, are also provided.

Domain: Invariants. The invariant is a formula which constrains the range of values in the resource (maybe relating different state components). This allows restricting the admissible states to those which are legal, and, therefore, it also specifies which states the resource must **not** evolve into. It is defined on the *current* state and has no direct means to refer to past or future states. It is the responsibility of the resource specification to ensure that forbidden states are not reached. In our case, the container cannot carry more weight than the maximum allowed, and the number of cranes waiting to unload cannot exceed the total number of cranes.

Specifying the Effect of Operations. Preconditions and postconditions describe when operations can proceed (i.e., they express synchronization) and how these operations change the resource state. Both are first-order formulas which involve the resource and the arguments of the operation.

Synchronization. The resource semantics assumes mutual exclusion between operations, and ensuring this is left to the final implementation. More involved operation synchronization is taken care of by means of concurrency preconditions (CPRE), which are evaluated against the resource state, and which are aimed at expressing *safety* conditions. A call whose CPRE is evaluated to *false* will block until a change in the resource (done by some other process) makes it *true*. Only one operation among those whose CPRE evaluates to true is allowed to proceed. We do not assume any fixed selection procedure — not even fairness.

Sequential preconditions (PRE) can be added to the operations to express conditions which have to hold for the operations to be safely executed. While a CPRE states synchronization, a PRE deals mainly with data structure coherence.

Updating Resources and Arguments. Changes in the resource and in the operation arguments are specified using a postcondition (POST) for every operation which relates the state of the resource and of the parameters before and after the call. The PRE and CPRE of the operation and the invariant have to hold after a POSTs is *executed*, assuming they held before. Values before and after the operation are decorated with the superscripts "in" and "out", respectively.

In our case study, preparing the replacement keeps the number of cranes accessing the container (to zero, as was necessary to make the CPRE true) and the $weight$ in the container, and sets the container state to $replacing$.

3 Translating Shared Resources into TLA

In this section we present an interlingua-based semantics for our notation. The interlingua is the specification language TLA+ [7, 9], a combination of a linear-time temporal logic (The Temporal Logic of Actions [8]) and Zermelo-Fränkel set theory. We will

present the semantics in an informal way, introducing some general information about the translation process and using the system specified in Section 2 to illustrate it.

3.1 Anatomy of a translation

Roughly speaking, a TLA+ specification is a formula S written as a conjunction of a TLA predicate I that states the initial value of TLA variables x, y, ... representing the state of the system and a next-state relation N (TLA action) that specifies valid value changes of the variables: $S \triangleq I \wedge \Box[N]_{\langle x, y, \ldots \rangle}$.

Resource state Each component of the domain of the specification in our notation will be translated into a TLA variable.

Example: TLA variables that represent the resource domain:

VARIABLES $weight$, $state$, $accessing$

Types and predicates Types are translated into sets and predicates (initial state predicate, invariant, preconditions, and concurrency preconditions) into TLA predicates.

Example: TLA set that represents type $Weight$:

$Weight \triangleq (MIN\_W\_CRANE \,..\, MAX\_W\_CRANE)$

Example: TLA predicate that represents the initial state:

$Init \triangleq \wedge weight = 0 \wedge state =$ "ready" $\wedge accessing = 0$

Postconditions are translated into TLA *actions* (a TLA action is a predicate which relates input and output states). The value of a variable before an action is represented using the variable name, and its value after the action is represented with the same name primed; so we replace x^{in} by x and x^{out} by x'.

Example: Type information, PRE, CPRE, POST of $Increment\_Weight$ in TLA:

$TYPE\_Increment\_Weight(w) \triangleq w \in Weight$

$PRE\_Increment\_Weight(w) \triangleq w \leq MAX\_W\_CRANE$

$CPRE\_Increment\_Weight(w) \triangleq$
$\quad \wedge weight + w \leq MAX\_W\_CONTAINER \wedge state \neq$ "replacing"

$POST\_Increment\_Weight(w) \triangleq$
$\quad weight' = weight + w \wedge$ UNCHANGED $state \wedge accessing' = accessing + 1$

We will not present in detail the translation of types, predicates and expressions, as it is a non difficult compilation exercise made easier by the richness and expressiveness of the mathematical toolkit of TLA+.

Operations Every operation is translated into a TLA action which is the conjunction of the predicates collecting type information, precondition, concurrency precondition, and postcondition of the operation. This action will represent an atomic, valid transition of the system.

Example: TLA action that represents operation $Increment\_Weight$:

$$
\begin{array}{l}
Increment\_Weight(w) \triangleq \\
\quad \wedge\ TYPE\_Increment\_Weight(w) \wedge PRE\_Increment\_Weight(w) \\
\quad \wedge\ CPRE\_Increment\_Weight(w) \wedge POST\_Increment\_Weight(w)
\end{array}
$$

Putting It All Together The TLA formula that gives semantics to the resource specification as a dynamic system is given by the definition of the *next-step* relation as the disjunction of all actions resulting from the translation of operations. Informally, every transition (step) the resource may experience is triggered by the execution of some of its operations. We are deliberately ignoring the restrictions that the processes impose on the possible operation interleavings.

Example: A TLA action that represent the execution of any operation:

$$
\begin{array}{l}
Next \triangleq \\
\quad Prepare\_Replacement \vee\ Notify\_Replacement \\
\quad \vee\ \exists\, w \in Weight : Notify\_Weight(w) \vee \exists\, w \in Weight : Increment\_Weight(w) \\
\quad \vee\ Notify\_Drop
\end{array}
$$

The formula $Spec$ which specifies that the system starts in a valid state and every transition it takes is one of these defined by the $Next$ formula, maybe leaving variables $weight$, $state$, or $accessing$ unchanged, is then

$$Spec \triangleq Init \wedge \Box[Next]_{\langle weight,\, state,\, accessing \rangle}$$

3.2 Translation explained

Let us summarise some of the most relevant points in the previous translation:

- The TLA specification syntactically reflects most of the components of our notation.
- Type information has been explicitly introduced in (the untyped) TLA by representing types as sets. Variable typing is therefore translated into set membership and type declarations have been translated as guarded TLA formulae — for example, $Weight$, which is used in the definition of predicate $TYPE\_Increment\_Weight$, itself part of the action $Increment\_Weight$, and the in the bounded existential quantification in the definition of $Next$.
- The invariant will be also eventually translated into the TLA resulting specification. It will be used during the checking stage (Section 4).
- Output parameters of operations which are private to processes are represented by new TLA variables visible by all operations but conceptually not part of the resource. They are however necessary to faithfully represent the operation behaviour and, since all operations are continuously available, the interleavings this specification can represent are a superset of the ones the processes can perform.

4 Verifying System Properties Using TLC

In this section we will see how some execution properties of our example can be studied thanks to the translation of shared resources into TLA and, eventually, the use of the TLC model checker. Being able to use such a tool does not guarantee, in general, the correctness of the system, but it helps to find possible inconsistencies or holes in the specifications.

Some of the properties to check are generic (i.e., the invariant always holds) and some of them depend on the system at hand. We will use in fact two variants of the specification. The first one is what we described in Section 3.1, which leaves complete freedom to the interleaving of the operations, and is adequate to verify safety properties which are connected with resource reuse. The second one includes the necessary machinery to enact interleaving constraints which model the behaviour of the process.

4.1 Checking the resource integrity

With no information about the context in which the specified resource will be used, only the integrity of the invariant and type information of variables can be checked.

Example: Checking the invariant

The invariant has been translated into the following TLA specification:

$$Types \triangleq weight \in Nat \land state \in State\_Ty \land accessing \in Nat$$

$$Invariant \triangleq \\ \land weight \leq MAX\_W\_CONTAINER \land accessing \leq MAX\_CRANES$$

The input to the model checker TLC (Fig. 4) consists of the TLA specification plus the definition of values for constants and the properties (invariants in this case) to be checked. The model checker found two violations of the invariant. The first one is a type error:

```
Error: Invariant Types is violated. The behavior up to this point is:
STATE 1: <Initial predicate>
/\ state = "ready" /\ weight = 0 /\ accessing = 0
STATE 2: <Action Notify_Drop>
/\ state = "ready" /\ weight = 0 /\ accessing = -1
```

```
CONSTANTS
  MAX_CRANES = 5
  MIN_W_CRANE = 1000
  MAX_W_CRANE = 1010
  MAX_W_CONTAINER = 20000
SPECIFICATION Spec
INVARIANT Types
INVARIANT Invariant
```

Fig. 4. TLC definition of the system to check.

The second one is the violation of the property $weight \leq MAX\_W\_CONTAINER$. We have modified the specification by introducing a CPRE in the operations $Increment\_Weight$ and $Notify\_Drop$ ($accessing < MAX\_W\_CRANE$ and $accessing > 0$, respectively). After this, it seems that our resource specification has reached a better degree of integrity. The model checker did not find more errors during the checking of the specified resource invariant. Note that, since we did not impose any restriction to the interleavings of the operations due to the way processes are defined, the properties we are checking here will be valid in any context of the resource, guaranteeing the reusability of the shared resource.

4.2 *Cooking* the processes

Studying properties which take wholly into account how processes and resources are defined as needs some additional *cooking* in order to encode their behaviour in TLA. We have followed this systematic method:

1. Introducing per-process program counters and local variables to represent the internal state of every process.
2. Establishing relevant program points for every type of process.
3. Introducing next-step relations in the specification for every process.
4. Mixing next-step relations of processes with next-step relations of the resource.
5. Writing the whole system specification with the initial state, the disjunction of all *cooked* TLA actions, and weak fairness conditions on every TLA action.

Example: *Cooking* `Crane_Controllers`

1. Variables $Crane\_Controller\_PC$ and $Crane\_Controller\_w$ represent the internal state of the every crane controller:

   VARIABLES $Crane\_Controller\_PC$, $Crane\_Controller\_w$

2. `Crane_Controller` program points:

   $Crane\_Controller\_Points \triangleq$
   {"toNotify_Weight", "toIncrement_Weight", "dropping", "toNotify_Drop"}

   The following fragment captures the type of program counters, local variables and initial state of the processes:

   $Processes\_Types \triangleq$
   $\land\ Crane\_Controller\_PC \in [Crane\_Id \to Crane\_Controller\_Points]$
   $\land\ Crane\_Controller\_w \in [Crane\_Id \to Weight]$

   $Processes\_Init \triangleq$
   $\land\ Crane\_Controller\_PC = [i \in Crane\_Id \mapsto$ "toNotify_Weight"$]$
   $\land\ Crane\_Controller\_w = [i \in Crane\_Id \mapsto$ CHOOSE $w \in Weight$: TRUE$]$

3. The next-step relations for the process `Crane_Controller` are the invocation of shared resource operation and the invocation of the `Cranes` API. Actually, just one new next-step relation is relevant: the transition from the invocation of `Increment_Weight` to the invocation of `Cranes.Drop`:

   $Dropping \triangleq$
   $\exists\, i \in Crane\_Id$:
   $\land\ Crane\_Controller\_PC[i] =$ "dropping"
   $\land\ Crane\_Controller\_PC' =$
   $[Crane\_Controller\_PC$ EXCEPT $![i] =$ "toNotify_Drop"$]$
   $\land$ UNCHANGED $Container\_Controller\_PC$
   $\land$ UNCHANGED $Crane\_Controller\_w$
   $\land$ UNCHANGED $weight \land$ UNCHANGED $state \land$ UNCHANGED $accessing$

4. We extend the next-step relations of the resource with the valid state changes in the processes:

$Cooked\_Notify\_Weight(w) \triangleq$
$\exists\, i \in Crane\_Id :$
$\land\ Crane\_Controller\_PC[i] =$ "toNotify_Weight"
$\land\ Crane\_Controller\_w[i] = w$
$\land\ Notify\_Weight(w)$
$\land\ Crane\_Controller\_PC' =$
$[Crane\_Controller\_PC$ EXCEPT $![i] =$ "toIncrement_Weight"$]$
$\land$ UNCHANGED $Container\_Controller\_PC$
$\land$ UNCHANGED $Crane\_Controller\_w$

$Cooked\_Increment\_Weight(w) \triangleq$
$\exists\, i \in Crane\_Id :$
$\land\ Crane\_Controller\_PC[i] =$ "toIncrement_Weight"
$\land\ Crane\_Controller\_w[i] = w$
$\land\ Increment\_Weight(w)$
$\land\ Crane\_Controller\_PC' =$
$[Crane\_Controller\_PC$ EXCEPT $![i] =$ "dropping"$]$
$\land$ UNCHANGED $Container\_Controller\_PC$
$\land$ UNCHANGED $Crane\_Controller\_w$

$Cooked\_Notify\_Drop \triangleq$
$\exists\, i \in Crane\_Id : \exists\, w \in Weight :$
$\land\ Crane\_Controller\_PC[i] =$ "toNotify_Drop"
$\land\ Notify\_Drop$
$\land\ Crane\_Controller\_PC' =$
$[Crane\_Controller\_PC$ EXCEPT $![i] =$ "toNotify_Weight"$]$
$\land$ UNCHANGED $Container\_Controller\_PC$
$\land\ Crane\_Controller\_w' = [Crane\_Controller\_w$ EXCEPT $![i] = w]$

5. Putting it all together:

$Cooked\_Next \triangleq$
$\lor\ Cooked\_Prepare\_Replacement \lor Cooked\_Notify\_Replacement$
$\lor\ \exists\, w \in Weight : Cooked\_Notify\_Weight(w)$
$\lor\ \exists\, w \in Weight : Cooked\_Increment\_Weight(w)$
$\lor\ Cooked\_Notify\_Drop$
$\lor\ Dropping \lor Replacing$

$Cooked\_Spec \triangleq\ Cooked\_Init \land \Box[Cooked\_Next]_{Cooked\_State}$

4.3 Checking system properties

With this *cooked* specification we can check the following system properties:

1. Absence of deadlock (this is automatically provided by TLC).

2. That no cranes drop any material while the container is being replaced:
   $No\_Dropping\_While\_Replacing \triangleq$
   $\Box(\neg(\wedge \exists\, i \in Crane\_Id : Crane\_Controller\_PC[i] =$ "dropping"
   $\wedge\ Container\_Controller\_PC =$ "replacing"))
3. Component $state$ in the resource has a cyclic behaviour ($ready \rightarrow to\_replace \rightarrow replacing \rightarrow ready \ldots$):
   $State\_Is\_Cyclic \triangleq$
   $\Box[\ \vee\ state =$ "ready" $\wedge\ state' =$ "to_replace"
   $\vee\ state =$ "to_replace" $\wedge\ state' =$ "replacing"
   $\vee\ state =$ "replacing" $\wedge\ state =$ "ready"$]_{state}$

TLC then detects the violation of one of the properties:

```
Error: Action property State_Is_Cyclic is violated. The behavior up to this point is:
...
STATE 76: <Action Notify_Weight>
/\ state = "to_replace" /\ weight = 19000 /\ accessing = 1
/\ Crane_Controller_PC = <<"toNotify_Weight", "toIncrement_Weight", "dropping">>
/\ Crane_Controller_w = <<1000, 1001, 1000>>
/\ Container_Controller_PC = "toPrepare_Replacement"
STATE 77: <Action Notify_Weight>
/\ state = "ready" /\ weight = 19000 /\ accessing = 1
/\ Crane_Controller_PC = <<"toIncrement_Weight", "toIncrement_Weight", "dropping">>
/\ Crane_Controller_w = <<1000, 1001, 1000>>
/\ Container_Controller_PC = "toPrepare_Replacement"
27332O states generated, 77163 distinct states found, 3008 states left on queue.
The depth of the complete state graph search is 77.
```

4.4 Analysis of the Error Detected

During the design process, the specifier decided to write the following specification for $Notify\_Weight$ (already translated into TLA):
$Notify\_Weight(w) \triangleq$
$CPRE$
$\wedge\ state \neq$ "replacing"
$POST$
$\wedge$ UNCHANGED $weight \wedge$ UNCHANGED $accessing$
$\wedge\ weight + w > MAX\_W\_CONTAINER \Rightarrow state' =$ "to_replace"
$\wedge\ weight + w \leq MAX\_W\_CONTAINER \Rightarrow state' =$ "ready"
The motivation to write such specification is that just because one crane asks for a replacement ($state' = to\_replace$ if the weight of its load exceeds the limit) no other crane with a valid load weight should wait to drop (resulting in changing the value of $state$ to $ready$ and avoiding the container to be replaced).
Is the formalised system a right system? Is $State\_Is\_Cyclic$ a desirable property of the system to be built? At least, the violation of the property revealed a liveness issue in the formalisation: if a crane controller i asks for a replacement ($state' = to\_replace$) and another crane controller j invalidates that request ($state' = ready$), then the crane i will have to wait (conditional synchronisation of operation $Increment\_Weight$) until another crane k (probably j) reactivates a new request.

5 Conclusion

We have presented a method for structuring and analysing concurrent software that allows developers to focus on concurrency in isolation from other aspects. Although conceptually simple, it enables the use of formal methods in order to verify nontrivial properties of realistic systems.

We sketched a semantics for our shared resource specifications based on a translation of CADTs into Temporal Logic of Actions. This translation has two practical advantages that can be relevant for a wider industrial adoption of similar approaches. On one hand, we think that a CADT is much easier to write and understand than the corresponding TLA specification; although it forces a more rigid architecture, in many cases it is arguably more advantageous to have a series of tools / mechanisms which are easy to apply to different scenarios instead of a generic, more complex, one. Our CADT is an example of such a perhaps more specific approach.

On the other hand, the availability of tools such as TLC gives system designers the ability to pinpoint mistakes / misconceptions at an early stage. The automatic translation from a more "focused" formalism to a general, powerful tool helps their adoption. Additionally, from a formal standpoint, providing an interlingua semantics can be easier to follow than introducing a new calculus, specially considering a formalism in evolution.

The process structure presented here is a very simple one. More sophisticated schemes have been proposed in the literature, all of them with the common goals of providing language and platform independence when designing and reasoning about concurrent applications. Models such as *Creol* [4], that proposes a formal model of distributed concurrent objects based on asynchronous message passing, or *CoBoxes* [10] which unifies active objects with structured heaps, are relatively recent proposals that elaborate on previous ones such as the *actors* model.

In the service-oriented computing paradigm, which is attracting much attention in the last years and which is inherently concurrent, languages like BPEL [6] have been used to implement business processes. While BPEL is very powerful and can express complex service networks, their full verification is challenging, partly because of its complex semantics.

Although it can be argued that these alternatives allow to express more complex process structures, the absence of a clear separation between active and passive entities does not favour a simple analysis of the behaviour of systems. On the other hand, it is usually a requirement of many industrial software system to avoid complexity as much as possible. In other words, some models can be *far more expressive* than needed or wanted in practise.

Several relatives to our resources can also be found in the literature. It is worth mentioning the concurrency features in VDM++ [3], similar in spirit, although our CADT's relax some of their restrictions as can be seen in [1]. Moreover, the CADT formalism permits some extensions for improving the expressiveness of CPREs that still allow semi-automatic code generation. We have not used these extensions in this paper for the sake of simplicity.

One of the shortcomings of our method, in its current state, is that system properties, unlike shared resources, must be specified directly in TLA. One possible way to over-

come this would be to enrich CADTs with trace-dependent conditions which could, in turn, simplify the way in which liveness properties are specified.
Our original CADT notation included a construct similar to the *history counters* in VDM [3]: number of times each operation has been requested, activated and completed with information about process identifiers and actual parameters. Formulae on history counters can be very expressive (path expressions or UML protocol state machines can be easily encoded with them) and protocol order between operations or certain liveness conditions can be formalised.
As some programming languages come equipped with constructs that allow to check the lock state of processes at run time, automatic code generation from these specifications seems feasible. These extensions, and a more formal specification of the translation into TLA, are subject of future work.

References

1. Manuel Carro, Julio Mariño, Ángel Herranz, and Juan José Moreno-Navarro. Teaching how to derive correct concurrent programs (from state-based specifications and code patterns). In C.N. Dean and R.T. Boute, editors, *Teaching Formal Methods, CoLogNET/FME Symposium, TFM 2004, Ghent, Belgium*, volume 3294 of *LNCS*, pages 85–106. Springer, 2004. ISBN 3-540-23611-2.
2. COST Action IC 0701 on Verification of Object Oriented Software. http://www.cost-ic0701.org.
3. John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. Springer, 2004.
4. Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 2006.
5. Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1995.
6. Diane Jordan, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Canyang Kevin Liu, Rania Khalaf, Dieter König, Mike Marin, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. Web Services Business Process Execution Language Version 2.0. Technical report, IBM, Microsoft, BEA, Intalio, Individual, Adobe Systems, Systinet, Active Endpoints, JBoss, Sterling Commerce, SAP, Deloitte, TIBCO Software, webMethods, Oracle, 2007.
7. Leslie Lamport. The TLA home page. http://www.research.microsoft.com/users/lamport/tla/tla.html.
8. Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.
9. Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson Education, Inc, 2002.
10. Jan Schäfer and Arnd Poetzsch-Heffter. Coboxes: Unifying active objects and structured heaps. In G. Barthe and F. de Boer, editors, *FMOODS 2008*, volume 5051 of *LNCS*, pages 201–219, 2008.
11. Axel van Lamsweerde. Formal Specification: a Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 147–159. ACM Press, 2000.