From CSP to B: Specifications of a Post Office.*

Marielle Doche and Andrew Gravell

Department of Electronics and Computer Science
University of Southampton, Highfield
Southampton SO17 1BJ, United-Kingdom
{mfd, amg}@ecs.soton.ac.uk

Abstract. In this paper, we describe the methodology followed to specify a distributed system. The first step of this approach is to use a model checker to define a specification in CSP, which highlights the structure of the system and how the different components communicate. Then, we translate this specification in B to improve it with more complex data structures, infinite types, and additional operations. Finally, we propose an approach to reuse the results of failures-divergences refinement from our CSP specification, to check data refinement of the improved B specification.
Keywords: Formal specification, CSP, failures-divergences refinement, B, data refinement, distributed system
Category 2

1 Introduction

To improve the specification and the validation of complex systems, lots of recent works integrate two different formal notations. For example, in [But99,MS98,FW99,MC99] the authors combine a behaviour-based notation (CSP) with a model-based one (B, Z or Object-Z).

In this paper, we describe the methodology followed to specify a Post Office case study, which combined two formal notations CSP [Hoa85] and B [Abr96].

A first specification of the Post Office is defined in CSP with the help of the FDR model-checker [For97]. This specification highlights the structure of the system and the links between the components. The model-checker allows us to detect presence of deadlock or livelock in our specification and to check if it satisfies some abstract requirements (by checking for failures-divergences refinement). However, the size of the specifications produced by such an approach is limited: we can use only finite types and some mechanisms cannot be specified and checked.

So we translate our CSP specification in B, using the Csp2B tool [But99]: each CSP process which describes a basic component produces a B abstract machine where the operations are the events involved in the process. Sequential properties of the events introduced in the CSP specification are preserved in the B machine by the introduction of state variables to guard the B operations. The abstract requirements are also described in a B abstract machine, communications between the components are described in a B refinement. We can then complete our specification with more powerful mechanisms and check data refinement with more complex data structures (using Atelier B [Ste96] or the B-Toolkit [B-C99]).

However, to make data refinement proofs easier, we can reuse the results of failures-divergences refinement. Indeed Morgan [Mor90] has defined a correspondence between action system and CSP. Moreover in [WM90] the authors prove that refinement of action system corresponds to the failures-divergences refinement. More recently Butler has extended these results to the refinement

* We acknowledge the support of the EPRSC (GR/M91013) for the ABCD project (http://www.ecs.soton.ac.uk/~phh/abcd/).

of B machines and to the failures-divergences refinement [But97]. We have found that, for this case study, about seventy per cent of proof obligations to check data refinement do not need to be discharged with the B prover because the FDR refinement check allows us to assume them.

Section 2 introduces our case study. Section 3 describes the CSP specification and the first verifications using the FDR model-checker. The translation to a B specification is given in section 4. Section 5 discusses how to facilitate data refinement checking with failures-divergences refinement results. Finally, section 6 concludes and presents on-going work.

2 Case study: a Post Office

The Post Office is a distributed database, which provides pensions to some customers (an original description, and some models in different formalisms are given in [HBC+99]). The system consists of:

- a single *Centre*, where most of the data is held;
- some *Offices*, where copies of relevant parts of the data are held;
- some *Customers*, who have each a home office (by default, the Centre).

A customer requests some data (here the amount of her/his pension represented by a number of tokens) at an office. If this office holds the data, the customer can directly collect them, else the office requests the data to the centre. If the centre holds the data, it sends them to the office, else it requests them from the home office of the customer. Moreover, the centre can distribute some data of a customer to the home office of this customer, and can receive new data (here an allocation of tokens) for a customer.

At the most abstract level, if a customer requests (as many times she/he wants) data at an office, and if there are some data in the database, then she/he collects at least a non-empty part of them.

In [HBC+99], the communications between the centre and the offices are synchronous. Here, we define asynchronous communications by adding a medium, in which messages are queued. This medium is loosely based on the Java Messaging Service [HBS99].

To make a formal finite description, we choose a system with two offices and three customers. Office1, resp. Office2, is the home office of Cust1, resp. Cust2, and the Centre is the home of Cust3. A graphical description is given on figure 1, which illustrates after a request of Cust2 at Office1 the possible events concerning Cust2.

3 CSP-FDR specification.

3.1 Language and methodology

Our first specification is defined in CSP [Hoa85], which is convenient for providing a small behavior-based model of our system that highlights the main components and the communications between them. The Failures-Divergence Refinement (FDR [For97]) model-checker is used to check absence of deadlock and divergence in our specification and to check if it refines abstract requirements.

The notation of the FDR tool is very close to CSP with event prefix operator $a \rightarrow P$, input $e?i$ and output $e!off$. We have also the external choice $\Box$ and interleaving $|||$ operators, all of which may be indexed. But in the parallel combination we make explicit the shared events X: $P \ |[\, X \,]|\ Q$. We have also an hiding operator $P \setminus A$, where A is the set of hidden events. $\{|\, X \,|\}$ represents

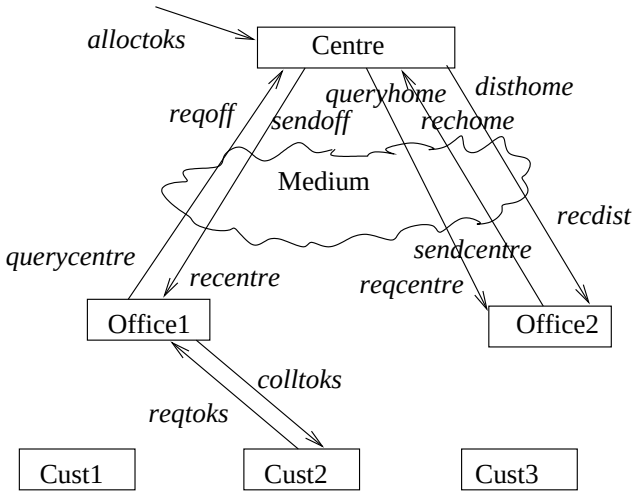

Fig. 1. Extended Post Office

the set of the productions of the values of X, i.e. the set of values which begin by a value of X. We can define some conditional expression *if* b *then* $e1$ *else* $e2$ and guarded events $b \,\&\, e$.

The other elements of the language used in the following specification are the boolean constants *true* and *false*, the boolean operators *not*, $\vee$ (or), $\wedge$ (and), and the comparison operators $==, !=$ (different) and $<$ (for more details on this language see [Ros97,For97]).

To specify our case study we have followed this compositional approach:

1. highlight the different components of the system and the links between them.
2. specify each component independently by a process.
3. specify the whole system in a process with sharing, synchronization,... avoiding where possible interleaving operators, which produce huge automata (state explosion).
4. check if this last process has any deadlocks or livelocks.
5. specify some properties in an abstract process and check that the system refines them.
6. add new constraints or new elements and repeat as necessary.

3.2 Application

Our Post Office is composed by four kinds of components, each of which is specified by a process: a customer `CUST(cu)`, the centre `CENTRE(cu)`, an office `OFF(cu,off)` and the medium `MEDIUM(cu,off)`. To use efficiently the model-checker, we have parameterized each of these processes by a given customer `cu`: they focus on the description of the behavior of a component according to a given customer. Indeed, we can suppose that each customer acts independently from another one, so we can define and check the system from a single customer's point of view, before checking it for many customers. In practice, we keep a small state machine as long as possible. In the same way, we parameterized the office and medium processes by a given office `off`.

Then, we define a process describing the whole system, first for a given customer `ASYNCHCUST(cu)` and then for several `ASYNCH`. We can also define easily a synchronous version (without the medium) in the processes `SYNCHCUST(cu)` and `SYNCH`.

Finally the abstract requirements are described also for a single customer `ABSTCUST(cu)` and for the whole system `SPEC`.

In this CSP-FDR specification, we do not specify the allocation of tokens at the centre (see discussion at the end of this section).

Types and functions. We define 4 types: the tokens, the customers, the homes and the offices. The set of homes consists of the centre `centre` and the set of offices `{office1,office2}`. For each customer, we define a home:

$$
\begin{array}{l}
nametype\ Tokens = \{0..5\} \\
nametype\ Cust = \{cust1,\ cust2,\ cust3\} \\
nametype\ Home = \{centre,\ office1,\ office2\} \\
nametype\ Office = diff\,(Home, \{centre\}) \\
home(cust1) = office1 \quad home(cust2) = office2 \quad home(cust3) = centre
\end{array}
$$

The customers. A customer makes a request at an office and then collects some tokens at this office.

$$
\begin{array}{l}
channel\ reqtoks : Cust.Office \\
channel\ colltoks : Cust.Office.Tokens
\end{array}
$$

$$
CUST(cu) \mathrel{\widehat{=}} reqtoks.cu?off \rightarrow colltoks.cu!off?n : Tokens \rightarrow CUST(cu)
$$

The centre. The centre communicates with the offices: it receives a request from an office `off` about a customer `cu`. If the centre has some tokens for this customer (`ctokens>0`), or if it is his home (`home(cu)==centre`) or if the office `off` is the home office of the customer (`home(cu)==off`) (the centre can not request as home the office from where it receives a request), then it sends the tokens it has to the office `off`, else it sends a request to the home of the customer, waits for the answer of the home and forwards it to the office `off`. `ctokens` represents the amount of tokens available at the centre for the customer `cu`.

Moreover, if the centre has tokens for a given customer (`ctokens>0`), and if it is not his home (`home(cu)!=centre`), then it can distribute the tokens to the home of the customer. The second control is necessary for the asynchronous version (otherwise there is a deadlock), but it seems obviously necessary in the synchronous version.

$$
\begin{array}{l}
channel\ sendoff,\ rechome,\ disthome : Cust.Office.Tokens \\
channel\ reqoff,\ queryhome : Cust.Office
\end{array}
$$

$$
\begin{array}{l}
CENTRE(cu, ctokens) \mathrel{\widehat{=}} \\
\quad reqoff.cu?off : Office \rightarrow \\
\qquad (if\ ctokens > 0 \lor home(cu) == centre \lor home(cu) == off \\
\qquad then\ sendoff.cu!off!ctokens \rightarrow CENTRE(cu, 0) \\
\qquad else\ queryhome.cu?off\,1 : home(cu) \rightarrow rechome.cu.home(cu)?am : Tokens \rightarrow \\
\qquad\quad sendoff.cu!off!am \rightarrow CENTRE(cu, ctokens) \\
\qquad) \\
\quad \Box \\
\quad (ctokens > 0 \land home(cu)! = centre)\ \& \\
\qquad disthome.cu?off\ : home(cu)!ctokens \rightarrow CENTRE(cu, 0)
\end{array}
$$

The offices. An office can receive a request from the centre or from a customer.

When it receives a request from the centre about a customer, it replies with the tokens it has for this customer.

When it receives a request from a customer, if it has tokens for him (or her) (`otokens>0`), then it gives them to him, else it sends a request to the centre, waits for an answer and gives it to the customer. `otokens` represents the amount of tokens available at the office `off` for the customer `cu`.

Moreover, an office can receive a distribution of tokens from the centre (the last line of the process `(#)`, see overleaf). But the centre can send the distribution at any moment, for example after a request of the customer (lines `(+)`).

However, to avoid a deadlock, if the office has already send a query to the centre after the request of a customer, and if it receives a distribution of tokens for this customer, it must wait for the answer to its query before giving the received tokens to the customer (line `($)`).

$$
\begin{array}{l}
\mathit{channel}\ \mathit{sendcentre},\ \mathit{reccentre},\ \mathit{recdist} : \mathit{Cust}.\mathit{Office}.\mathit{Tokens} \\
\mathit{channel}\ \mathit{reqcentre},\ \mathit{querycentre} : \mathit{Cust}.\mathit{Office} \\
\\
\mathit{OFF}(\mathit{cu}, \mathit{off}, \mathit{otokens}) \mathrel{\widehat{=}} \\
\quad \mathit{reqcentre}.\mathit{cu}.\mathit{off} \rightarrow \mathit{sendcentre}.\mathit{cu}.\mathit{off}!\mathit{otokens} \rightarrow \mathit{OFF}(\mathit{cu}, \mathit{off}, 0) \\
\quad \Box \\
\quad \mathit{reqtoks}.\mathit{cu}.\mathit{off} \rightarrow \\
\qquad (\mathit{otokens} > 0 \,\&\, \mathit{colltoks}.\mathit{cu}.\mathit{off}!\mathit{otokens} \rightarrow \mathit{OFF}(\mathit{cu}, \mathit{off}, 0) \\
\qquad \Box \\
\qquad \mathit{otokens} == 0 \,\&\, (\\
\qquad\quad (\mathit{querycentre}.\mathit{cu}.\mathit{off} \rightarrow (\\
\qquad\qquad \mathit{reccentre}.\mathit{cu}.\mathit{off}?\mathit{am} : \mathit{Tokens} \rightarrow \\
\qquad\qquad \mathit{colltoks}.\mathit{cu}.\mathit{off}!\mathit{am} \rightarrow \mathit{OFF}(\mathit{cu}, \mathit{off}, 0) \\
\qquad\qquad \Box \\
\qquad\qquad \mathit{recdist}.\mathit{cu}.\mathit{off}?\mathit{am} : \mathit{Tokens} \rightarrow \qquad -- (+) \\
\qquad\qquad \mathit{reccentre}.\mathit{cu}.\mathit{off}?\mathit{bm} : \mathit{Tokens} \rightarrow \qquad -- (\$) \\
\qquad\qquad \mathit{colltoks}.\mathit{cu}.\mathit{off}!\mathit{am} \rightarrow \mathit{OFF}(\mathit{cu}, \mathit{off}, \mathit{bm}) \qquad -- (+) \\
\qquad\quad)) \\
\qquad\quad \Box \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad -- (+) \\
\qquad\quad (\mathit{off} == \mathit{home}(\mathit{cu}) \,\&\, \mathit{recdist}.\mathit{cu}.\mathit{off}?\mathit{am} : \mathit{Tokens} \rightarrow \qquad -- (+) \\
\qquad\qquad \mathit{colltoks}.\mathit{cu}.\mathit{off}!\mathit{am} \rightarrow \mathit{OFF}(\mathit{cu}, \mathit{off}, 0)) \qquad -- (+) \\
\qquad\quad) \\
\qquad) \\
\quad \Box \\
\quad \mathit{recdist}.\mathit{cu}.\mathit{off}?\mathit{am} : 1..5 - \mathit{otokens} \rightarrow \mathit{OFF}(\mathit{cu}, \mathit{off}, \mathit{otokens} + \mathit{am}) \quad -- (\#)
\end{array}
$$

The medium. The medium manages an asynchronous communication between the centre and the offices.

A first solution was to define a process for each kind of message exchanged, and to compose them by interleaving. However, we have some problems proving refinement with such a mechanism: some messages can overtake others and tokens can be missed. So we define, for a given customer, a FIFO mechanism for the medium to order the messages on a queue between a sender and a receiver, for example between the centre and office1.

We need first to give them a common "shape" (multiplexing) (see figure 2): this is described by the process `MULTI(cu,off)`. Then at the end of the medium the messages should be analyzed and sent in the appropriate shape (demultiplexing). This is described in the process `DEMULTI(cu,off)`. Each queue is described in the process `Transmit(s, r, c)`, the provider itself in `PROVIDER(cu)`.

First of all we define a new type for the messages. FDR allows us to build a complex type with a label and sometimes an amount of tokens. Then we define functions: a message contains its type and eventually the amount of tokens it is carrying.

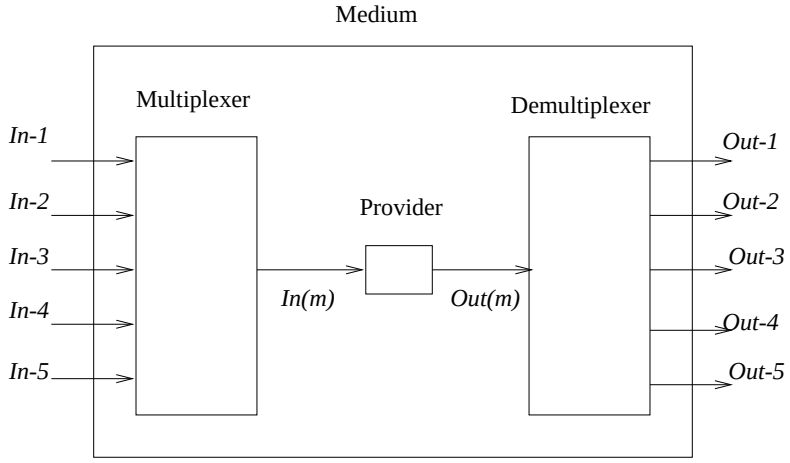

Fig. 2. Multiplexing-Demultiplexing mechanism.

$$
\begin{array}{l}
datatypeMessage = hc.Tokens \mid ro \mid co.Tokens \mid rc \mid di.Tokens \\
mtype(hc.am) = hc\ mval(hc.am) = am \ -- \ hc: \ home\ sends\ tokens\ to\ centre \\
mtype(ro) = ro \qquad\qquad\qquad\qquad\quad -- \ ro: \ office\ requests\ tokens\ to\ centre \\
mtype(co.am) = co\ \ mval(co.am) = am \ -- \ co: \ centre\ sends\ tokens\ to\ office \\
mtype(rc) = rc \qquad\qquad\qquad\qquad\quad -- \ rc: \ centre\ requests\ tokens\ to\ home \\
mtype(di.am) = di\ \ mval(di.am) = am \ -- \ di: \ centre\ distributes\ tokens\ to\ home
\end{array}
$$

The multiplexer waits a sending event from the offices or the centre and defines an input event for the provider, with the sender and the receiver of the message and the message itself.

$$
\begin{array}{l}
MULTI(cu, off) \mathrel{\widehat{=}} \\
\quad sendcentre.cu.off?am \rightarrow in.cu.off.centre!(hc.am) \rightarrow MULTI(cu, off) \\
\quad \Box\ querycentre.cu.off \rightarrow in.cu.off.centre!(ro) \rightarrow MULTI(cu, off) \\
\quad \Box\ sendoff.cu.off?am \rightarrow in.cu.centre.off!(co.am) \rightarrow MULTI(cu, off) \\
\quad \Box\ queryhome.cu.off \rightarrow in.cu.centre.off!(rc) \rightarrow MULTI(cu, off) \\
\quad \Box\ disthome.cu.off?am \rightarrow in.cu.centre.off!(di.am) \rightarrow MULTI(cu, off)
\end{array}
$$

At the end of the medium, the demultiplexer checks the message and provides the appropriate event to the receiver.

$$
\begin{array}{l}
DEMULTI(cu, off) \mathrel{\widehat{=}} off! = centre \wedge (\\
\quad (out.cu.off.centre?mes \rightarrow \\
\qquad (mtype(mes) == hc \,\&\, rechome.cu.off!mval(mes) \rightarrow DEMULTI(cu, off) \\
\qquad \Box\ mtype(mes) == ro \,\&\, reqoff.cu.off \rightarrow DEMULTI(cu, off)) \\
\quad) \\
\Box \\
\quad (out.cu.centre.off?mes \rightarrow \\
\qquad (mtype(mes) == co \,\&\, reccentre.cu.off!mval(mes) \rightarrow DEMULTI(cu, off) \\
\qquad \Box\ mtype(mes) == di \,\&\, recdist.cu.off!mval(mes) \rightarrow DEMULTI(cu, off) \\
\qquad \Box\ mtype(mes) == rc \,\&\, reqcentre.cu.off \rightarrow DEMULTI(cu, off)) \\
\quad) \\
)
\end{array}
$$

There are different ways to define a queue between a sender s and a receiver r. The most efficient for the FDR model-checker, consists of building a sequence of n buffers of one place [For97]. Thus,

we build only an automata for an one-place buffer, which is reused n times (this is expressed by $within\,[out \leftrightarrow in]\, i :< 1..5 >\bullet\, COPY$).

$$channel\ in,\ out : Cust.Home.Home.Message$$

$$--s : the\ sender\ of\ the\ message,\ r : the\ receiver$$
$$Transmit(s, r, c) \mathrel{\widehat=}$$
$$\quad let$$
$$\quad COPY \mathrel{\widehat=} in.cu.s.r?m \rightarrow out.cu.s.r!m \rightarrow COPY$$
$$\quad within\,[out \leftrightarrow in]\, i :< 1..5 >\bullet\, COPY$$

In the process `PROVIDER(cu)` we initialize the system of connections between the centre and the offices.

$$PROVIDER(cu) \mathrel{\widehat=} Transmit(centre, office1, cu) \;|||\; Transmit(centre, office2, cu)$$
$$\quad |||\; Transmit(office1, centre, cu) \;|||\; Transmit(office2, centre, cu)$$

So we can now build the process `ORD_MEDIUM(cu,off)` which describes the ordered medium.

$$ORD\_MEDIUM(cu, off) \mathrel{\widehat=} (MULTI(cu, off) \,|[\ \{|\ in.cu\ |\}\ ]|$$
$$\quad (PROVIDER(cu) \,|[\ \{|\ out.cu\ |\}\ ]|\ DEMULTI(cu, off)) \setminus \{|\ out.cu, in.cu\ |\}$$

Whole system. To build the whole system, we define three processes.

`ASYNHCOM(cu, ctokens)` describes for the customer `cu` the asynchronous transactions between the centre and the offices via the medium, `ctokens` representing the amount of tokens for this customer at the centre; initially there are 0 tokens at each office.

$$ASYNCHCOM(cu, ctokens) \mathrel{\widehat=}$$
$$\quad (CENTRE(cu, ctokens)$$
$$\qquad |[\ \{|\ rechome.cu, reqoff.cu, disthome.cu, sendoff.cu, queryhome.cu\ |\}\ ]|$$
$$\quad (|||_{off:Office}(ORD\_MEDIUM(cu, off)$$
$$\qquad |[\ \{|\ sendcentre.cu.off, querycentre.cu.off, reccentre.cu.off, recdist.cu.off, reqcentre.cu.off\ |\}\ ]|$$
$$\qquad OFF(cu, off, 0))$$
$$\qquad \setminus \{|\ sendcentre.cu.off, querycentre.cu.off, reccentre.cu.off, recdist.cu.off, reqcentre.cu.off\ |\}))$$

`ASYNCHCUST(cu, ctokens)` describes how a customer interacts with the system.

$$ASYNCHCUST(cu, ctokens) \mathrel{\widehat=}$$
$$\quad CUST(cu) \,|[\ \{|\ reqtoks.cu, colltoks.cu\ |\}\ ]|$$
$$\quad (ASYNCHCOM(cu, ctokens)$$
$$\qquad \setminus \{|\ rechome.cu, reqoff.cu, disthome.cu, sendoff.cu, queryhome.cu\ |\})$$

`ASYNCH` describes the whole asynchronous system with 3 customers, with initially 4 tokens for each at the centre.

$$ASYNCH \mathrel{\widehat=} |||_{cu:Cust}\, ASYNCHCUST(cu, 4)$$

A synchronous version (without the medium) is also defined. The $\leftrightarrow$ FDR operators allows us to synchronize two events with different names. For example, for a given customer `cu`, $disthome.cu.off.am \leftrightarrow recdist.cu.off.am \mid off \leftarrow Office, am \leftarrow Tokens$ expresses the synchronization of the events $disthome.cu.off.am$ and $recdist.cu.off.am$ for an office `off` and some tokens `am`.

$$
\begin{array}{l}
SYNCHCOM(cu, ctokens) \mathrel{\widehat{=}} \\
\quad CENTRE(cu, ctokens) \\
\quad\quad |[disthome.cu.off.am \leftrightarrow recdist.cu.off.am, sendoff.cu.off.am \leftrightarrow reccentre.cu.off.am, \\
\quad\quad rechome.cu.off.am \leftrightarrow sendcentre.cu.off.am, reqoff.cu.off \leftrightarrow querycentre.cu.off, \\
\quad\quad queryhome.cu.off \leftrightarrow reqcentre.cu.off \mid off \leftarrow Office, am \leftarrow Tokens]| \\
\quad (|||_{off:Office}\ OFF(cu, off, 0))
\end{array}
$$

The process `SYNCHCUST(cu, ctokens)`, respectively `SYNCH`, is built like the process `ASYNCHCUST(cu, ctokens)`, respectively `ASYNCH`.

Abstract level An abstract property of the system is given by the customer's behavior: any customer collects tokens at an office only after a request at the same office. If there are some tokens in the data base for this customer (`ctokens > 0`), it collects at least one of them, other it collects zero tokens. This is described by the process:

$$
\begin{array}{l}
ABSTCUST(cu, tokens) \mathrel{\widehat{=}} \\
\quad reqtoks.cu?off \rightarrow \\
\quad\quad (tokens > 0 \,\&\, colltoks.cu!off?m : 1..tokens \rightarrow ABSTCUST(cu, tokens - m) \\
\quad\quad \Box \\
\quad\quad tokens == 0 \,\&\, colltoks.cu!off?0 \rightarrow ABSTCUST(cu, 0) \\
\quad\quad)
\end{array}
$$

The *SPEC* process describe the whole abstract system with initially 4 tokens for each customer in the database.

$$SPEC \mathrel{\widehat{=}} |||_{cu:Cust}\ ABSTCUST(cu, 4)$$

3.3 Check

FDR tool. The FDR tool allows us to check if a given process is a refinement of an another one according three models: trace, failures or failures-divergences. In our case study we have used the failures-divergences model:

Q is a failures-divergences refinement of P iff any failure of Q (indeed a pair formed by a finite trace and the set of refused events after this trace) is a failure of P and any divergence of Q (when Q repeats infinitely often an internal event) is a divergence of P:

$$P \sqsubseteq_{FD} Q \mathrel{\widehat{=}} failures(Q) \subseteq failures(P) \ \wedge\ divergences(Q) \subseteq divergences(P)$$

Moreover we can check in a process absence of deadlock (when the process refuses every event) and absence of divergence (when the process performs an infinite sequence of internal events) following the failures-divergences model.

Deadlocks and divergences. We prove first that each description of the system (the abstract level and the concrete levels) is deadlock and divergence free. In this table, we give for each description, the size of the transition system (number of states and transitions) built by the FDR model checker and the time it needs. For some concrete versions, the transition system was too big to be checked, so we have made only a partial verification for one customer.

Process	States	Transitions	Time (in seconds)
SPEC	216	864	0
ABSTCUST(1,4)	6	8	0
SYNCH	5,600	20,960	3
SYNCHCUST(1,4)	20	36	0
ASYNCH	-	-	-
ASYNCUST(1,4)	299	481	0

Refinement check. Then we prove that the synchronous version `SYNCH` is a failures-divergences refinement of the abstract version `SPEC`. We can also prove that the process `SYNCHCUST(1,4)` refines `ABSTCUST(1,4)`.

$$SPEC \sqsubseteq_{FD} SYNCH$$
$$ABSTCUST(1,4) \sqsubseteq_{FD} SYNCHCUST(1,4)$$

For the asynchronous version, we can only make the check for one customer, as the transition system built for the whole post office is too large. We can prove that `ASYNCHCUST(1,4)` is a failures-divergences refinement of the abstract version `ABSTCUST(1,4)`.

$$ABSTCUST(1,4) \sqsubseteq_{FD} ASYNCHCUST(1,4)$$

However, the monotonicity of the interleaving operators (see [Ros97,For97]) allows us to deduce that:

$$SPEC \sqsubseteq_{FD} ASYNCH$$

Moreover we can prove for a given customer that the synchronous and asynchronous versions of communication are equivalent for the failures-divergences refinement:

$$SYNCHCOM(1,4) \sqsubseteq_{FD} ASYNCHCOM(1,4)$$
$$ASYNCHCOM(1,4) \sqsubseteq_{FD} SYNCHCOM(1,4)$$

3.4 Discussion

To use the FDR model-checker, we must define a specification which is not going to produce too large automata. Thus we need to reduce the use of the interleaving operator. Moreover we must choose small data types, and it is sometimes sufficient during the check, to consider only one value of the type, for example we consider often in this example only the first customer.

The tool allows us to detect deadlocks or divergences, and to check refinement for small systems (in the previous example the tool checks 5,600 states and 20,960 transitions in a few seconds for the `SYNCH` process). When errors are found, the debugger gives a counter-example. In this way, we can improve our comprehension of the system.

However, with the asynchronous version, we have reached some size limits for the tool: direct checks on the whole system are very difficult (several hours) and sometimes impossible with a crude model-checking strategy. Indeed, we spent several days checking the processes `ASYNCH` and execution was stopped after having analyzed 28,000,000 states. We need to make the checks on small parts. It is furthermore helpful to consider the checking strategy when structuring the specification.

Finally, some problems occur when we will add events to manage the set of tokens, because we cannot have infinite sets for types, and it is difficult to define global variables. For example, in a

more complex version, we try to introduce an event for the allocation of tokens at the centre (see specification B in [HBC+99]). So we need to define the new event `alloctoks` and to modify the process `ABSTCUST` and the process `CENTRE` to take into account this allocation. For this purpose, we need to add in both these processes the line `(1)`, with a constraint on the amount of tokens allocated because we have defined the type tokens as an integer interval (`nametype Tokens = {0..5}`). The ideal specification is to define the type Tokens as all the integers (`nametype Tokens = Int`) in which case we use the line `(2)`.

$channel\ alloctoks : Cust.Tokens$

$$
\begin{array}{l}
ABSTCUST(cu, tokens) \mathrel{\widehat{=}} \\
\quad alloctoks.cu?am : 1..5 - tokens \rightarrow ABSTCUST(cu, tokens + am) \quad --(1) \\
\quad --alloctoks.cu?am \rightarrow ABSTCUST(cu, tokens + am) \quad --(2) \\
\Box \\
\quad reqtoks.cu?off \rightarrow \\
\qquad (tokens > 0 \,\&\, colltoks.cu!off?m : 1..tokens \rightarrow ABSTCUST(cu, tokens - m) \\
\qquad \Box \\
\qquad tokens == 0 \,\&\, colltoks.cu!off?0 \rightarrow ABSTCUST(cu, 0) \\
\qquad)
\end{array}
$$

The ideal is then to check the refinement with the type `Tokens` as the whole set of integers (line `(2)`). But this is impossible with FDR, since the tool can not build the automaton.

So we try to make the check with the type `Tokens` as an interval of integers (line `(1)`), first for the synchronous version. But, we meet the following problem (which appears in the asynchronous version too):

- At the concrete level, the centre distributes 4 tokens at office 1, home of the customer 1. Then it allocates 2 tokens (between 0 and 5) at the customer 1: `disthome.1.1.4; alloctoks.1.2;`
- At the abstract level, nothing happens: the allocation of 2 tokens for the customer 1 is impossible because the 4 tokens of the database have not yet been collected by the customer 1 and at most 1 token can be allocated to him.

4 CSP to B

The second step of our approach, is to define a more powerful specification, which allows us to define more complex data structures (especially infinite types) and to check more complex mechanisms. We have chosen to use the B-method [Abr96], which is a model-based notation, to have an improved specification.

4.1 Tools and Methodology

The first step consists of highlighting in the CSP specification the design as it is going to occur in the B design: abstract specification and basic components are defined as abstract machines whereas concrete level are defined as B refinements. Then the csp2B tool [But99] translates the constraints on sequentiality of events of a CSP specification in B:

- for each CSP process, a new type and a new variable are introduced in the B machine to manage the state of the process;
- each CSP event becomes a B operation, guarded by the state variables

For this case study, we follow this approach:

1. To define a B abstract machine with all the types and definitions shared by all the components of the system.
2. To define a csp2B machine from the abstract process of the FDR specification. And then to generate automatically with csp2B the abstract B machines.
3. To define a CSP machine for each component of FDR (the basic processes). And then to generate automatically with csp2B the abstract B machines.
4. To define a B refinement to describe the interactions between the components from the FDR processes which describe the whole system.
5. To define in the B refinement, some refinement invariants between abstract level data and concrete level ones, and to check them.

We start from the CSP-FDR specification described in the previous section and we add the allocation mechanism (cf version B of [HBC+99]). We keep the same process names.

4.2 Application

The structure of the synchronous version is illustrated in the figure 3.

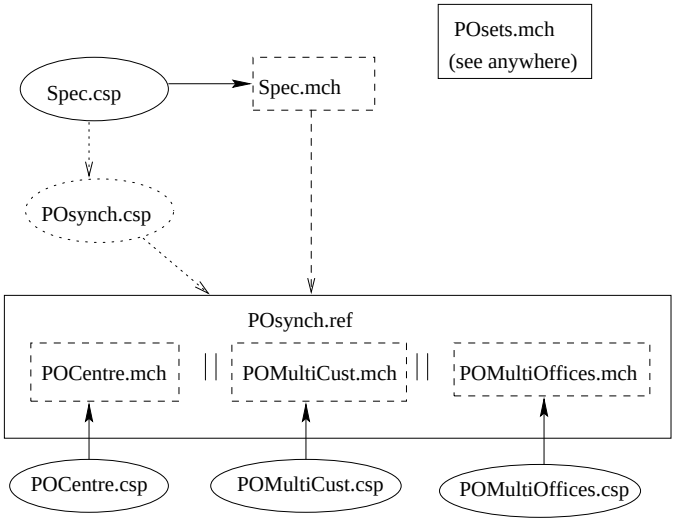

Fig. 3. Synchronous version from FDR.

Sets and global definitions. The preliminary step is to define a B abstract machine with all the types and other definitions shared by all the other components of the design.

For the *Tokens* we are going to use the predefined types NAT, indeed the infinite set of integers.

In B, the definition of the set *OFFICE* as a subset of the set *HOME* is specified by a property. The CSP-FDR constant function *home* is also defined as a constant function in B and it is initialized by a property.

Moreover we need to specify the message type, which is defined in CSP-FDR as a polymorphic type, with functions to access to the values. In B we cannot define polymorphic types. So we define

five constants with values in the set $POMESSAGE$ to specify the type of the messages, some of which are parameterized by the values in the messages.

The last property specifies that the type $POMESSAGE$ is exactly the set of the ranges of the five constants.

MACHINE *POsets*

SETS

$CUST = \{ cust1 , cust2 , cust3 \}$;
$HOME = \{ office1 , office2 , CENTRE \}$;
$POMESSAGE$

CONSTANTS

$OFFICE , home , hc , ro , co , rc , di$

PROPERTIES

$OFFICE = HOME - \{ CENTRE \}$
$\wedge\ home \in CUST \rightarrow HOME$
$\wedge\ home = \{ cust1 \mapsto office1 , cust2 \mapsto office2 , cust3 \mapsto CENTRE \}$
$\wedge\ hc \in \mathbb{N} \rightarrow POMESSAGE$
$\wedge\ ro \in POMESSAGE$
$\wedge\ co \in \mathbb{N} \rightarrow POMESSAGE$
$\wedge\ rc \in POMESSAGE$
$\wedge\ di \in \mathbb{N} \rightarrow POMESSAGE$
$\wedge\ POMESSAGE = \mathsf{ran}(hc) \cup \{ ro \} \cup \mathsf{ran}(co) \cup \{ rc \} \cup \mathsf{ran}(di)$

END

Abstract level For the abstract level and for each component we translate directly the CSP-FDR processes in Csp2B machines.

The CSP-FDR channels become Csp2B events, but we need to specify which parameter is input or output.

Then we need to choose if the parameter of the CSP-FDR process becomes a *parameter* or an *index* in csp2B: the parameter allows us to express generic behavior whereas the index manages a variable attached to the process.

In our example, the customer cu becomes a parameter and the set of tokens *tokens* an indice (the amount of tokens in the system for the customer cu).

Moreover, in the Csp2B version, we can have infinite sets, so we have choose to specify the second version (line `(2)` of the CSP-FDR specification) for the allocation of tokens.

MACHINE POSpec

SEES

POsets

ALPHABET

Alloctoks(cu : CUST, am : NAT)
toks ←– Colltokens(cu : CUST, off : OFFICE)
Reqtokens(cu : CUST, off : OFFICE)

PROCESS

Spec = ||| cu.AbstCust[cu](0)

WHERE

AbstCust[cu](tokens: Nat) =

Alloctoks.cu?am → AbstCust[cu](tokens + am)

□

Reqtokens.cu?off →

(IF tokens > 0 THEN Colltokens.cu?off!toks: 1.. tokens → AbstCust[cu](tokens -toks)

□

IF tokens = 0 THEN Colltokens.cu?off!0 → AbstCust[cu](0))

END

The csp2B tool generates automatically the following B abstract machine: for the process Spec the type *SpecState* and the variable *Spec*, whose domain is *CUST* due to the parameter *cu*, have been introduced. Each CSP event becomes a B operation guarded by the value of the variable *Spec*.

For the B-refinement step, we need to declare in this abstract specification the hidden events as `skip` operation: for example here the distribution operation (we do not give the details of all hidden operators).

MACHINE ***POSpec***

SEES

POsets

SETS

$SpecState = \{ AbstCust , AbstCust\_1 \}$

VARIABLES

$Spec , off\_1 , tokens$

INVARIANT

$Spec \in CUST \rightarrow SpecState \wedge$

$off\_1 \in CUST \rightarrow OFFICE \wedge$

$tokens \in CUST \rightarrow \mathbb{N}$

INITIALISATION

$Spec := \lambda\ cu\ .\ (\ cu \in CUST \mid AbstCust\ )\ \|$

$tokens := \lambda\ cu\ .\ (\ cu \in CUST \mid 0\ )\ \|$

ANY new_off_1 **WHERE**

$new\_off\_1 \in CUST \rightarrow OFFICE$

THEN $off\_1 := new\_off\_1$

END

OPERATIONS

$Alloctokens\ (\ cu , amount\ ) \mathrel{\widehat{=}}$

PRE $cu \in CUST \wedge amount \in \mathbb{N}$ **THEN**

SELECT

```
        Spec ( cu ) = AbstCust
      THEN
        tokens ( cu ) := tokens ( cu ) + amount
      END
    END ;

  toks ⟵ Colltokens ( off , cu ) ≙
    PRE  off ∈ OFFICE ∧ cu ∈ CUST  THEN
      SELECT
        tokens ( cu ) > 0 ∧ Spec ( cu ) = AbstCust_1
      THEN
        ANY  tok  WHERE  tok ∈ 1 .. tokens ( cu )
        THEN
          Spec ( cu ) := AbstCust  ||
          tokens ( cu ) := tokens ( cu ) − tok  ||  toks := tok
        END
      WHEN
        tokens ( cu ) = 0 ∧ Spec ( cu ) = AbstCust_1
      THEN
        Spec ( cu ) := AbstCust  ||  tokens ( cu ) := 0  ||  toks := 0
      END
    END ;

  Reqtokens ( cu , off ) ≙
    PRE  off ∈ OFFICE ∧ cu ∈ CUST  THEN
      SELECT
        Spec ( cu ) = AbstCust
      THEN
        Spec ( cu ) := AbstCust_1  ||  off_1 ( cu ) := off
      END
    END ;

  Disthome ( cu , off ) ≙
    PRE  cu ∈ CUST ∧ off ∈ OFFICE  THEN
      skip
    END ;
  ...
END
```

The basic components. In the CSP-FDR specification, we have used parametrized processes to describe the components with an interleaving operator on parameter cu introduced at the last step of the composition. This solution provides a more efficient use of the FDR model-checker. Conversely, for the B specification, it is better to define directly machines which describe the behaviour of a set of customers. The B syntax does not allow us to reuse a machine which describes one customer n times, and we need directly defined a machine which describes the n customers.

For this purpose, the CSP-FDR parameter cu becomes in csp2B language the parameter cu of an interleaving operator in each component (see the example of the centre bellow).

However we can prove that our CSP-FDR expression:

$$SYNCH \mathrel{\widehat{=}} |||_{cu:Cust}\ CUST(cu) \,|[\, X(cu) \,]|\ ((CENTRE(cu) \,|[\, Y(cu) \,]|\ |||_{off:Office}\ OFF(cu, off)) \setminus \{Y(cu)\})$$

is equivalent to the expression[1] (we have the same result for the asynchronous expression):

$$SYNCH \mathrel{\widehat{=}} ((|||_{cu:Cust}\ CUST(cu)) \,|[\, X \,]|\ ((|||_{cu:Cust}\ CENTRE(cu)) \,|[\, Y \,]|\ (|||_{cu:Cust,off:Office}\ OFF(cu, off)))) \setminus \{Y\})$$

In the following we are giving details only for the centre, the machines of the other components are obtained by the same way.

We add at the centre specification the allocation mechanism. Once again, we choose the more generic solution (line `(2)` of the FDR specification).

The CSP-FDR process `CENTRE` becomes the Csp2B process `Await`. We need to cut it in two parts (with the process `TransHome`) to specify that the amount of tokens at the centre for a given customer is modified between the events `rechome` and `sendoff`.

MACHINE POCentre

SEES

POsets

ALPHABET

Alloctokens(cu : CUST, amount : NAT)

toks ←– Disthome(cu : CUST, off : OFFICE)

toks ←– Sendoff(cu : CUST, off : OFFICE)

Rechome(cu : CUST, off : OFFICE, amount : NAT)

Reqoff(cu : CUST, off : OFFICE)

Queryhome(cu : CUST, off : OFFICE)

PROCESS

WholeCentre = ||| cu.Centre[cu](0)

WHERE

Centre[cu](ctokens : NAT) =

Alloctokens.cu?am → Centre[cu](ctokens +am)

□

IF ctokens > 0 ∧ not(home(cu) = CENTRE) ∧ off = home(cu)

THEN

[1] In our case we have the following alphabet: $\alpha(CUST(cu)) = X(cu)$, $\alpha(CENTRE(cu)) = Y(cu)$ and $\alpha(|||_{off:Office}\ OFF(cu, off)) = Z(cu) = X(cu) \cup Y(cu)$. First $X(cu) \cap Y(cu) = \{\}$, so we can apply the law of distribution of the hiding operator on parallel and interleaving operators [Hoa85]:

$$SYNCH \mathrel{\widehat{=}} (|||_{cu:Cust}\ CUST(cu) \,|[\, X(cu) \,]|\ (CENTRE(cu) \,|[\, Y(cu) \,]|\ |||_{off:Office}\ OFF(cu, off)) \setminus \{Y(cu)\})$$

Then we apply the associativity law on ‖ operator.

Disthome.cu?off!ctokens → Centre[cu](0)

□

Reqoff.cu?off → TransHome[cu](ctokens,0)

TransHome[cu](ctokens, homecall: NAT) =

IF ctokens>0 or home(cu)= CENTRE or off=home(cu) or homecall = 1

THEN

SendOff.cu?off!ctokens → Centre[cu](0)

ELSE

IF off = home(cu) THEN QueryHome.cu?off →

IF off = home(cu) THEN RecHome.cu?off?am → TransHome[cu](ctokens+am, 1)

END

Concrete level To specify the whole system at the concrete level, we must translate the CSP-FDR processes in a B refinement which describes the interactions between the components. Indeed we specify in B the shared events, but we add moreover what is an input, and what is an output. For example, *POsynchro* (see bellow) describes a synchronous refinement of *POSpec*. It contains three components: a centre *ce*, a set of offices *mo* and a set of customers *mc*. Its operations have the same names than those of the abstract level. Each is described as a sequential composition of the operations of the components (for more details of this approach, see [But97]).

Moreover, we need to specify some refinement invariants. They link abstract and concrete variables to check data refinement. The best way to define them is to associate at most one abstract value with each concrete one (we call this also a functional abstraction invariant).

REFINEMENT ***POSynchro***

REFINES ***POSpec***

SEES ***POsets***

INCLUDES ***ce . POCentre , mo . POMultiOffices , mc . POMultiCust***

INVARIANT

$\forall\ customer\ .\ (\ customer \in CUST \Rightarrow$
$tokens\ (\ customer\ ) = ce\ .\ ctokens\ (\ customer\ ) +$
$\sum\ off\ .\ (\ off \in OFFICE \ \mid\ mo\ .\ otokens\ (\ off\ ,\ customer\ )\ )\ )$
$\wedge$
$\forall\ customer\ .\ (\ customer \in CUST \Rightarrow Spec\ (\ customer\ ) = AbstCust$
$\Leftrightarrow \forall\ office\ .\ (\ office \in OFFICE \Rightarrow$
$mo\ .\ Offices\ (\ office\ ,\ customer\ ) \in \{\ Off\ ,\ Off\_1\ \}\ )\ )$
$\wedge$
$\forall\ customer\ .\ (\ customer \in CUST \Rightarrow$
$\exists\ office\ .\ (\ office \in OFFICE \wedge$
$mo\ .\ Offices\ (\ office\ ,\ customer\ ) \in \{\ Transaction\ ,\ Transaction\_1\ \}\ )$
$\Leftrightarrow (\ Spec\ (\ customer\ ) = AbstCust\_1\ )\ )$
$\wedge$
...

OPERATIONS

Alloctokens (*cu* , *amount*) $\widehat{=}$ *ce* . *Alloctokens* (*cu* , *amount*) ;

toks $\longleftarrow$ *Colltokens* (*off* , *cu*) $\widehat{=}$
PRE *off* $\in$ *OFFICE* $\wedge$ *cu* $\in$ *CUST*
THEN
toks $\longleftarrow$ *mo* . *Colltokens* (*off* , *cu*) ; *mc* . *Colltokens* (*cu* , *off* , *toks*)
END ;

Reqtokens (*cu* , *off*) $\widehat{=}$
PRE *off* $\in$ *OFFICE* $\wedge$ *cu* $\in$ *CUST*
THEN
mc . *Reqtokens* (*cu* , *off*) ; *mo* . *Reqtokens* (*off* , *cu*)
END ;

Disthome (*cu* , *off*) $\widehat{=}$
PRE *off* $\in$ *OFFICE* $\wedge$ *cu* $\in$ *CUST*
THEN
VAR *tok1* **IN**
tok1 $\longleftarrow$ *ce* . *Disthome* (*cu* , *off*) ; *mo* . *Recdist* (*off* , *cu* , *tok1*)
END
END ;

...

END

For the asynchronous version, the concrete level is obtained in the same way.

4.3 Check

To check our B specification we can use both Atelier B [Ste96] and the B-Toolkit [B-C99].

For the abstract B machines (in our case *POsets*, *POspec*, and the machines which describe the components) the tools generate proof obligations to check that constants and variables satisfy the specification properties or invariants after initialization and each operations (usually these properties and invariants describe type relation). In our case, few proof obligations have been generated which are easy to prove automatically or with the help of simple user rules (see table below with results for the B-Toolkit, similar results are obtained with Atelier B).

For the refinement (*POsynchro* and *POasynchro*), the tools generate proof obligation to check that the initialization and the operations of the refinement satisfy the abstract machine according to the refinement invariant. Lots of proof obligations are generated during this step, and the proof of them is complex (we have easily proved only a third).

Machine	Proof obligations	Automatically proved	Remaining after automatic proof	Interactively proved	Remaining after interactive proof
POsets	4	2	2	2	0
POSpec	10	10	0	-	-
POCentre	11	11	0	-	-
POMultiCust	3	2	1	1	0
POMultiOffices	21	19	2	2	0
Multiplex	30	30	0	-	-
Demultiplex	17	17	0	-	-
InOut	4	2	2	2	0
POsynchro	76	10	66	17	49
POasynchro	123	16	107	26	81

4.4 Conclusion

In this second approach, we have seen how a small CSP-FDR model, can be translated and completed to obtain a more detailed and more generic model. Most of the translation could be made automatically (with the Csp2B tool [But99]). But it is still a complex step to specify the refinement invariants. Another manual step consists of choosing what is an input and what is an output of an operation, because we do not make a distinction in CSP-FDR.

We can notice that this approach allows the generation of fifty per cent fewer proof obligations, which are easier to prove, than a previous approach which consists to define directly a B specification without using a model-checker [Doc00].

5 Discussion

The B specification allows us to have a more generic and more complex model than the CSP-FDR specification. However, data refinement is difficult to prove, and needs lots of time. So we propose to reuse the results of the failures-divergences refinement to make it easier.

5.1 Correspondence between Failures-Divergences refinement and data refinement

Woodcock and Morgan [WM90] have established a correspondence between failures-divergences refinement and *simulation* for action systems. Butler [But97] has extended this result to the B machine. A machine *Concrete* simulates a machine *Abstract* if *Concrete* is a data refinement of *Abstract* and some progress and non divergence conditions are verified on *Concrete*.

The csp2B tool [But99] allows us to define a first B specification equivalent to the CSP-FDR specification (we keep a finite set for the tokens and we do not add the allocation mechanism). We can then assume that all the proof obligations generated from this B specification to prove data refinement are verified.

In our case study, we meet two kinds of proof obligations to check data-refinement:

1. For each invariant of the refinement (*POSynchro*), a proof obligation is defined to check that the initialization satisfies it without contradiction with the initialization statement of the abstract machine (*POSpec*);
2. For each operation (there are the same operations names in the abstract machine and the refinement), for each invariant of the refinement (*POSynchro*), a proof obligation is defined to check that the operation preserves it without contradiction with the initialization statement of the abstract machine (*POSpec*);

where the invariants of refinement, describe the link between the abstract and the concrete variables and the types of some concrete variables.

5.2 To infinite type

The first change we have made, is to use an infinite sets for the tokens. However most of the proof obligations generated after this change remain identical.

Indeed, if we change a type, variables of this types are modified, and so are the invariants that contain those variables. Thus, we need to re-prove all the proof obligations generated for those invariants (for the initialization and the operations).

However, some other invariants do not contain expressions which depend (directly or not) on the changed type. For them, we can assume that their proofs do not depend on the modified type, and so it is not necessary to prove them again.

In our example, we have changed the types *Tokens* (see the B machines *POsets* and *POSpec* and the B refinement *POSynchro*). So the type of the variables (*tokens*, *ce.ctokens* and *mo.otokens*) which appear in the first invariant of *POSynchro* has been modified, and all the proof obligations generated to check this invariant must be re-proved.

But in the second invariant of *POSynchro*, the type *Tokens* (or *NAT* after change) does not appear, nor the variables (*Spec* and *Offices*) which depend on this type, so it is not necessary to re-prove the proof obligations for this invariant.

Thus we have only twenty per cent of proof obligations to re-prove.

5.3 Addition of events

The second modification has been to add a new event without changing the state space of our specification. Indeed, we have add an operation *Alloctokens*, but we have not changed the set of variables nor the invariants.

To check data refinement, we have exactly the same set of proof obligations as before modification and a set of proof obligations for the new operation. Because we have not changed the set of variables and the types, it is necessary to prove only this second set of proof obligations.

6 Conclusion

In this paper we have presented a methodology to specify a distributed system, which involves two formal languages CSP and B. The use of a model-checker during the early step of the specification allows us to specify easily a well-structured small model of our system. With the translation in B, we can build a more complex specification which deals with more realistic data types. Moreover, the results of failures-divergences refinement automatically obtained by model-checking on our first specification are reused to prove with a theorem prover the data refinement on our second specification.

For the moment, the translation from CSP to B is partially automatic: the B refinement which describes the concrete level has been translated manually. We are improving the csp2B tool to deal with this step, and to assist the verification of data refinement, by flagging the proof obligations we have not to re-prove.

References

[Abr96] J-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[B-C99] B-Core (UK) Limited, Oxon, UK. *B-Toolkit, On-line manual.*, 1999. http://www.b-core.com/ONLINEDOC/Contents.html.

[But97] M. J. Butler. An approach to the design of distributed systems with B AMN. In J. Bowen, M. Hinchey, and D. Till, editors, *Proc. 10th Int. Conf. of Z Users: The Z Formal Specification Notation (ZUM), LNCS 1212*, pages 223–241, Reading, UK, April 1997. Springer-Verlag, Berlin.

[But99] M. J. Butler. csp2B: A practical approach to combining CSP and B. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proc. FM'99: World Congress on Formal Methods*, LNCS 1708, pages 490–508. Springer-Verlag, Berlin, September 1999.

[Doc00] M. Doche. Form FDR to B: Specification of the post office. Unpublished specification description, July 2000.

[For97] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement- FDR2 user manual*, Octobre 1997. Available at www.formal.demon.co.uk/fdr2manual/index.html.

[FW99] C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In *First international Conference on Integrated Formal Methods (IFM99)*, pages 315–334. Springer-Verlag, 1999. Available at http://semantik.Informatik.Uni−Oldenburg.DE/persons/clemens.ficher/eindex.html.

[HBC+99] P. Hartel, M. Butler, A. Currie, P. Henderson, M. Leuschel, A. Martin, A. Smith, U. Ultes-Nitsche, and B. Walters. Questions and answers about ten formal methods. In S. Gnesi and D. Latella, editors, *Proc. 4th Int. Workshop on Formal Methods for Industrial Critical Systems*, volume II, pages 179–203, Trento, Italy, July 1999. ERCIM, STAR/CNR, Pisa, Italy.

[HBS99] M. Hapner, R. Burridge, and R. Sharma. *Java Message Service, Version 1.0.2*. Sun Microsystems, Java Software, November 1999. Available at java.sun.com/products/jms/index.html.

[Hoa85] C.A.R Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[MC99] I. MacColl and D. Carrington. Specifying interactive systems in Object-Z and CSP. In *First international Conference on Integrated Formal Methods (IFM99)*. Springer-Verlag, 1999. Available at http://archive.csse.uq.edu.au/ ianm/.

[Mor90] C.C. Morgan. Of wp and CSP. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our business: a birthday salute to Edsger W. Dijkstra*. Springer Verlag, 1990.

[MS98] A. Mota and A. Sampaio. Model-checking CSP-Z. In *Fundamental Approach of Software Engineering (FASE98)*, number 1382 in LNCS, pages 205–220. Springer Verlag, 1998. Available at http://www.di.ufpe.br/ acm/.

[Ros97] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[Ste96] Steria, Aix-en-Provence, France. *Atelier B, User and Reference Manuals*, 1996. http://www.atelierb.societe.com/index_uk.html.

[WM90] J.C.P. Woodcock and C.C. Morgan. Refinement of state-based concurrent systems. In D. Bjorner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90*, volume 428 of *LNCS*. Springer-Verlag, 1990.