Extraction of Abstraction Invariants for Data Refinement*

Marielle Doche and Andrew Gravell

Department of Electronics and Computer Science
University of Southampton, Highfield
Southampton SO17 1BJ, United-Kingdom
marielle.doche@libertysurf.fr, amg@ecs.soton.ac.uk

Abstract. In this paper, we describe an approach to generating abstraction invariants for data refinement from specifications mixing B and CSP notations. A model-checker can be used to check automatically refinement of the CSP parts. However, we need to translate the CSP into B in order to verify data refinement of the whole specification. The Csp2B tool generates the B specification automatically from the CSP parts. Our proposal is to generate in addition the abstraction invariants, by analysing the labelled transition systems provided by a model-checker. The approach is illustrated with a case study in which a simple distributed information system is specified and two refinements are given, both of which have been fully verified using the proposed combination of model-checking with theorem proving (both automatic and interactive).

Keywords: Formal specification, CSP, failure refinement, B, data refinement, distributed system

1 Introduction

To improve the specification and the validation of complex systems, lots of recent research concerns the integration of two different formal notations to use the advantages of each. For example, in [But99,MS98,FW99,MC99,DS00] the authors combine a behaviour-based notation (CSP) with a model-based one (B, Z or Object-Z). Indeed the B method [Abr96] or the Z notation [Spi92] are suitable for modelling distributed systems, but the sequencing of events is difficult to specify. The CSP language [Hoa85] solves this problem more easily.

The work presented in this paper is based on the Csp2B approach [But99]. In that paper, Butler describes how to combine specifications in CSP and in B, and how to derive automatically B specifications from these combinations. This approach can be applied to derived B machines or B refinements. He also proves that, if they do not share variables, the composition of a CSP and a B specifications is compositional with respect to the data refinement process, which allows

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

us to refine independently each part. Although it is easy to check refinement of the CSP subpart with a model-checker such as FDR [For97], in a lot of cases, however, we cannot prove refinement of the B subpart on its own, because this in fact depends on the state of the CSP subpart. In such cases, we need to generate from the CSP and B subparts a B machine and its refinement, which can then be verified.

The classical approach to refinement with the B method is data refinement [Abr96] which can be supported by theorem provers such as AtelierB [Ste96] or the B-Toolkit [B-C99]. In data refinement, however, we need to define abstraction invariants that link the variables of the abstract machine with those of the concrete one. This step is often based on the intuition of the specifier and it is difficult to achieve.
In this paper, we propose an approach for reducing the difficulty of this task by generating automatically the abstraction invariants that relate to the CSP subparts. First the FDR tool is used to check refinement of the CSP subparts alone. To do this, it constructs a labelled transition system (LTS). The LTS can be extracted from FDR and used to generate abstraction invariants in B. These can then be conjoined with the abstraction invariants relating to the B subparts.

The following section introduces our example and the Csp2B approach. We discuss some proof issues and when our approach can be applied in section 3. Section 4 describes our approach on a simple case of data refinement. A more complex case, with parallel decomposition, is given in section 5.

2 Csp2B approach on our example

Our work is based on the Csp2B approach proposed by Butler [But99]. The idea of this approach is to increase the descriptive power of B specifications (B machines or B refinements) with the inclusion of CSP processes, which describe the order of events. Moreover, the Csp2B tool automatically translates the CSP processes into B language which can then be checked with a B prover (Atelier B or the B-Toolkit).

2.1 Basic Example

A customer requests some tokens (typically some data, a pension,..) at an office and then collects them at the same office. This is expressed in Csp2B by:

MACHINE *Tokens*

SEES *TokensDef*

ALPHABET
ReqTokens(off:OFFICE)

toks ← *CollTokens(off:OFFICE)*

PROCESS *Customers* = **Await**
CONSTRAINS *ReqTokens(off) CollTokens(off)* **WHERE**

Await $\widehat{=}$ *ReqTokens?off* → **Transact** *(off)*

Transact *(off_ab : OFFICE)* $\widehat{=}$ *CollTokens.off_ab* → **Await**
END

END

Here → is the classical prefixing operator of CSP, the event $ReqTokens$ has an input parameter $?off$, the event $CollTokens$ has a 'dot' parameter *.off_ab* which means it accepts as input only the value of *off_ab.* The **CONSTRAINS** clause allows us to constrain only a subset of the alphabet or some of the parameters (in this example, the input parameter of the event *CollTokens* is constrained, but not its output). The declaration *Customers* = **Await** defines the initial state of the process *Customers.*This Csp2B description can see the contents of the following B machine, where *home* is a function which associates a home office to each customer (here there are three customers):

MACHINE *TokensDef*

SETS
$HOME = \{O1, O2, CENTRE\}$

CONSTANTS
home, *OFFICE*, *initTokens*

PROPERTIES
$OFFICE = HOME - \{CENTRE\} \wedge$
$initTokens \in \mathbb{N} \wedge initTokens = 6 \wedge$
$home \in \mathbb{N} \rightarrow HOME \wedge home = \{1 \mapsto O1, 2 \mapsto O2, 3 \mapsto CENTRE\}$

END

Then, the Csp2B tool [But99] translates the constraints on the order of events:

- for each CSP process, a new set and new variables 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 (using the B **SELECT** statement).

For our example, we obtain a set *CustomersState* with the values *Await* and *Transact*. Two variables are introduced: *Customers* and *off_ab*. The tool generates the following B machine:

MACHINE *Tokens*

SEES *TokensDef*

SETS
CustomersState = {*Await*,*Transact*}

VARIABLES
Customers,*off_ab*

INVARIANT
Customers ∈ *CustomersState* ∧ *off_ab* ∈ *OFFICE*

INITIALISATION
Customers := *Await* ||
ANY *new_off_ab* **WHERE**
new_off_ab ∈ *OFFICE* **THEN** *off_ab* := *new_off_ab* **END**

OPERATIONS

ReqTokens(*off*) =
PRE *off* ∈ *OFFICE* **THEN**
SELECT *Customers* = *Await* **THEN**
Customers := *Transact* || *off_ab* := *off*
END
END ;

toks ← **CollTokens**(*off*) =
PRE *off* ∈ *OFFICE* **THEN**
SELECT *Customers* = *Transact* ∧ *off* = *off_ab* **THEN**
Customers := *Await*
END
END

END

2.2 Conjunction

Moreover, a Csp2B machine *M* may constrain the order of the operations of an already defined B machine *MActs*. This is defined in the Csp2B description by

the clause **CONJOINS** *MActs,* and for each event *Op* of the Csp2B description, the B machine *MActs* contains an operation *Op_Act* with the same interface.

In our example, the B machine *TokensActs* specifies the amount of tokens available for the customer in the system:

MACHINE *TokensActs*

SEES *TokensDef*

VARIABLES
tokens

INVARIANT
tokens ∈ ℕ

INITIALISATION
tokens := initTokens

OPERATIONS

ReqTokens_Act(*off*) = **PRE** *off* ∈ *OFFICE* **THEN skip END** ;

toks ← **CollTokens_Act**(*off*) =
PRE *off* ∈ *OFFICE* **THEN**
IF *tokens* = 0 **THEN** *tokens* := 0 || *toks* := 0
ELSE
ANY *tok* **WHERE** *tok* : (1 .. *tokens*) **THEN**
tokens := *tokens* - *tok* || *toks* := *tok*
END
END
END

END

The Csp2B tool generates a B machine *M* from the Csp2B description as previously, but the B machine *M* includes the B machine *MActs.* Now, each operation *Op* contains a guarded call to the operation *Op_Act* of the machine *MActs* . Indeed, if *Op_Csp* is the B statement for the operation *Op* generated from the Csp2B description we obtain:

Op= *Op_Csp* || **SELECT** *grd(Op_Csp)* **THEN Op_Act END**

We generate the following B machine (the beginning is the same as previously):

MACHINE *Tokens*

SEES *TokensDef*

INCLUDES *TokensActs*

DEFINITIONS
grd_Tokens_CollTokens(off)== (*Customers*=*Transact* ∧ *off*=*off_ab*);
grd_Tokens_ReqTokens(off)== (*Customers*=*Await*)
...

OPERATIONS

ReqTokens(*off*) =
PRE *off* ∈ *OFFICE* **THEN**
SELECT *grd_Tokens_ReqTokens(off)*
THEN ReqTokens_Act(*off*)
END
||
SELECT *Customers* = *Await*
THEN *Customers* := *Transact* || *off_ab* := *off*
END
END ;

toks ← **CollTokens**(*off*) =
PRE *off* ∈ *OFFICE* **THEN**
SELECT *grd_Tokens_CollTokens(off)*
THEN *toks* ← **CollTokens_Act**(*off*)
END
||
SELECT *Customers* = *Transact* ∧ *off* = *off_ab*
THEN *Customers* := *Await*
END
END

END

2.3 Data refinement

This same approach can be applied to produce a B refinement, which can be verified entirely in B with one of the B provers (Atelier B [Ste96] or the B-Toolkit [B-C99]).

The classical approach to data refinement in B involves introducing concrete variables and extra (hidden) operations. Moreover, to check data refinement, we

need to define some abstraction invariants to link the abstract variables with the concrete ones.

The following refinement is defined to refine the previous *TokensActs* machine. Here we give some hints on the internal structure of our system: it is composed of two offices (*O1* and *O2*) and a *Centre*. The tokens about the customer can be held by any of the offices or the centre, thus we introduce the new variables *otokens* and *ctokens*. The abstraction invariant $tokens = ctokens + otokens(O1) + otokens(O2)$ means that the global amount of tokens for a customer is the sum of the tokens at the centre and both the offices.

Moreover we introduce new operations to describe the internal communications between the centre and the offices. A customer requests some data at an office (operation **ReqTokens_Act**). If this office holds the data, the customer directly collects them (**CollTokens_Act**), else the office requests the data from the centre (**ReqOff_Act**). If the centre holds the data, it sends them to the office (**SendOff_Act**), else it requests and receives them from the home office of the customer (**QueryHome_Act** and **RecHome_Act**), where the home office of our customer is defined by the *home* function in the machine *TokensDef*. An original description, and some models of this example in different formalisms are given in [HBC+99].

REFINEMENT *TokensRefActs*

REFINES *TokensActs*

SEES *TokensDef*

VARIABLES
otokens, *ctokens*

INVARIANT
$ctokens \in \mathbb{N} \ \wedge \ otokens \in OFFICE \rightarrow \mathbb{N} \ \wedge$
$tokens = ctokens + otokens(O1) + otokens(O2)$

INITIALISATION
$ctokens := \text{initTokens} \ || \ otokens := OFFICE \times \{ 0 \}$

OPERATIONS

ReqTokens_Act(*off*) = **PRE** $off \in OFFICE$ **THEN skip END**;

$toks \leftarrow$ **CollTokens_Act**(*off*) =
PRE $off \in OFFICE$ **THEN**
$otokens(off) := 0 \ || \ toks := otokens(off)$

END;

SendOff_Act(*off*) =
PRE *off* ∈ *OFFICE* **THEN**
ctokens := 0 || *otokens*(*off*) := *otokens*(*off*) + *ctokens*
END;

RecHome_Act(*off*) =
PRE *off* ∈ *OFFICE* **THEN**
otokens(*off*) := 0 || *ctokens* := *ctokens* + *otokens*(*off*)
END;

ReqOff_Act(*off*) =
PRE *off* ∈ *OFFICE* **THEN**
SELECT *otokens*(*off*) =0 **THEN skip END**
END;

QueryHome_Act(*off*) =
PRE *off* ∈ *OFFICE* **THEN**
SELECT *ctokens* = 0 ∧ *home*(1) = *off* **THEN skip END**
END

END

Unfortunately, it is not possible to prove that *TokensRefActs* refines *TokensActs*: in the case where *tokens>0* and *otokens(off)=0* the concrete operation **CollTokens_Act** does not refine the abstract one. We need more information on the evolution of the variables *ctokens* and *otokens*.

The following Csp2B specification describes the order of the events.[1] Here □ is the external choice of Csp2B.

REFINEMENT *TokensRef*

REFINES *Tokens*

SEES *TokensDef*

CONJOINS *TokensRefActs*

ALPHABET
ReqTokens(off:OFFICE)
toks ← *CollTokens(off:OFFICE)*

[1] In practice we cannot include a B refinement in a B machine. Thus in this example, the conjoined B specification *TokensRefActs* is the transcription of the previous B refinement example in a B abstract machine. The abstraction invariant already defined will be introduced in the generated B refinement.

```
    SendOff(off: OFFICE)
    RecHome(off: OFFICE)
    ReqOff(off: OFFICE)
    QueryHome(off: OFFICE)

PROCESS  System = Asleep
CONSTRAINS  ReqTokens(off)  CollTokens(off)  SendOff(off)
  RecHome(off)  ReqOff(off)  QueryHome(off)  WHERE

    Asleep ≙ ReqTokens?off → Request (off)

    Request (off_co :OFFICE) ≙
          IF otokens(off_co) > 0
          THEN CollTokens.off_co → Asleep END
        □ ReqOff.off_co → Answer

    Answer ≙
          IF not(off_co=home(1)) THEN (QueryHome.home(1)
              → RecHome.home(1) → SendOff.off_co → Collect)
          ELSE (SendOff.off_co → Collect)
          END

    Collect ≙ CollTokens.off_co → Asleep
END

END
```

This description can be compiled by the Csp2B tool to produce a B refinement of the machine *Tokens*. The tool produces a new set $SystemState = \{$ *Asleep, Request, Answer, Answer_1, Answer_2, Collect* $\}$ and new variables *off_co* and *System*. In this example, the process **Answer** contains some implicit states (indeed between the events $QueryHome.home(1)$ and $RecHome.home(1)$ and the events $RecHome.home(1)$ and $SendOff.off\_co$). For these implicit states, the Csp2B tool generates then some fresh names (the name of the process followed by an underscore character and a number).

Unfortunately, the tool does not define abstraction invariants to link these new variables with the concrete ones. This must be done manually, which can be difficult in some cases.

3 Discussion

In this section, we are going to discuss the different cases of refinement checking, and when the approach we propose in the sequel can be used.

Figure 1 summarises the Csp2B approach: a Csp description M_CSP is defined which may conjoin a B machine M_Act. The Csp2B tool generates automatically a B machine M. The same approach is applied to define a machine R which is a refinement of M.

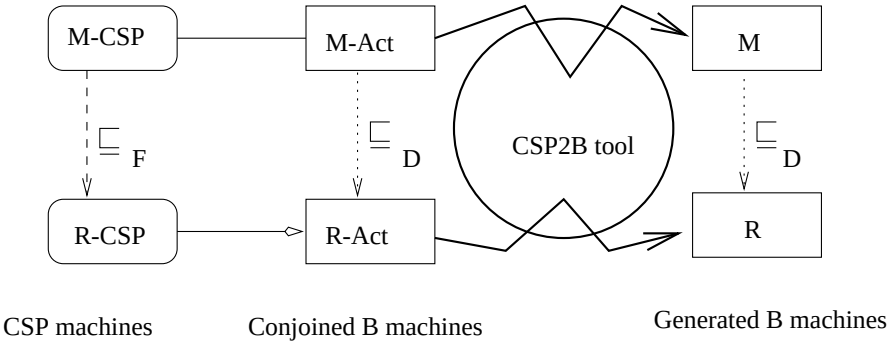

Fig. 1. Csp2B process

To check refinement, three cases are possible:

- there are no conjoined B machines,
- the conjoined B machine M_Act is refined by R_Act,
- we cannot prove that M_Act is refined by R_Act (see our first example)

3.1 Without conjoined machines

Morgan [Mor90] has defined a correspondence between action systems and CSP. In [WM90] the authors 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*. Thus in theory, if we have proved failures refinement on a CSP specification, we do not need to prove data refinement on its translation (failures refinement is sufficient because at present data refinement with B does not consider hidden events and divergence).

3.2 Refinement between conjoined machine is proved

In [But99], Butler has shown that, if the CSP machine doesn't refer the state variables of the conjoined B machine, the parallel operator used to generate the B operations is monotonic with respect to refinement. This allows us to refine independently conjoined B machines and CSP processes. In this case, if

$M\_CSP \sqsubseteq_F R\_CSP$ and $M\_Act \sqsubseteq_D R\_Act$ then $M \sqsubseteq_D R$.

If the CSP machine refers to variables of the conjoined B machine, as in our second example, section 5, this result is not valid and the refinement must be proved on the generated B machines. In such a case we can apply our following proposed approach to generate some of the abstraction invariants.

3.3 Refinement between conjoined machine is not proved

In this case, we have to prove refinement of the generated B machine (see our first example in sections 2.3 and 4), with some abstraction invariants.

However, when the CSP machines do not refer to variables of the conjoined machines, we can reduce the proof task. In such a case, the parallel operator is monotonic with respect to data refinement, thus the CSP part can be refined independently. Our proposed approach generates abstraction invariants that are independent of the variables of the conjoined B machine, and hence are obviously preserved by this part. Thus in this case, the proof obligations regarding our generated abstraction invariants can easily be discharged.

4 Simple data refinement

The aim of this paper is to propose an automatic approach to defining abstraction invariants for the state variables introduced by the Csp2B tool. This approach is based on the analysis of the Labelled Transition Systems (LTS) built from the Csp2B descriptions. Such LTS can easily be obtained with a model-checker like FDR [For97].

4.1 Proposed approach

Refinement mapping. The FDR tool provides easily and automatically three kinds of checks, increasing in strength:

1. *Trace refinement*: all possible sequences of events for the implementation are possible sequences for the specification.
2. *Failures refinement*: any failure of the implementation (indeed a pair formed by a finite trace of events and the set of refused events after this trace) is a failure of the specification.
3. *Divergence refinement*: any failure of the implementation is a failure of the specification and any divergence of the implementation (when it repeats infinitely often an event) is a divergence of the specification.

Failures refinement, quickly checked with the FDR tool, is a necessary condition for data refinement (indeed B refinement checking does not currently allow to detect divergence of a system).

To make these checks, the tool builds a LTS for the specification and one for the implementation and considers inductively pairs of nodes from the abstract LTS and the concrete one (for more details see [For97,Ros97]). Thus, in case of trace refinement, we can define a relation between abstract states and concrete ones. In [AL91], Abadi and Lamport show that if a concrete transition system refines an abstract one, there is a mapping from the state space of the concrete transition system to the state space of the abstract one, if necessary by adding auxiliary variables.

Formally, we call $M : \Sigma_C \rightarrow \Sigma_A$ the refinement mapping, where Σ_C and Σ_A are the sets of nodes respectively of the concrete LTS and of the abstract LTS. $dom(M)$ and $ran(M)$ are respectively the domain and the codomain of M. Given a in $ran(M)$, we denote by $M^{-1}(a)$ the set of nodes of Σ_C which have a as image by M:

$$M^{-1}(a) \mathrel{\widehat{=}} \{c | c \in dom(M) \wedge M(c) = a\}$$

In practice, such refinement mappings can be easily obtained with state space reduction algorithms [For97,Ros97]: for any node n of the concrete LTS, we group with n all the nodes reachable from n by an internal event. All these nodes have the same image by M, which can be computed inductively from the initial state.

Acceptance sets. Moreover, the FDR tool provides for each node of an LTS the *acceptance* set, the set of events the node must accept. If trace refinement is verified, the acceptance set of a concrete node is included in the acceptance set of the corresponding abstract node union the set of hidden events (the events or operations present only in the concrete LTS). If failures refinement is verified, we can be sure that for a concrete node if the set is empty, the set of the corresponding abstract node is also empty (since the deadlocks of the implementation are deadlocks of the specification).

The refinement mapping between the LTS provided by FDR and the acceptance sets are our starting point for defining the B abstraction invariants.

For any node n of an LTS, we call $G(n)$ the acceptance set of n. We extend this notation to several nodes : $G(n_1, ..., n_k) = G(n_1) \cup ... \cup G(n_k)$. So, for $a \in ran(M)$ we obtain:

$$G(M^{-1}(a)) = \bigcup_{c \in M^{-1}(a)} G(c)$$

Failures refinement ensures that if $G(M^{-1}(a))$ is empty then $G(a)$ is also empty.

If we call H the set of hidden events, trace refinement ensures:

$$\forall c \in \Sigma_C, G(c) \subseteq G(M(c)) \cup H$$

Finally for any event e, we denote by $grd(e)$ the guard of this event, the condition under which e is enabled, expressed by a predicate.

Abstraction invariant. For each node of Σ_A, we define an invariant.

For the abstract LTS, in a node $a \in \Sigma_A$, at least one guard of an event of $G(a)$ is satisfied. This means that $\bigvee_{e \in G(a)} grd(e)$ is true.

If $a \in ran(M)$, the refinement mapping ensures that at least one of the guard of an event of $G(M^{-1}(a))$ is also satisfied. For each $a \in ran(M)$, we define thus an invariant :

$$(\bigvee_{e \in G(a)} grd(e)) \Rightarrow (\bigvee_{f \in G(M^{-1}(a))} grd(f))$$

If $a \in (\Sigma_A - ran(M))$, this node is not an image of a concrete one, so it can never be reached in the concrete model. So we define an invariant:

$$\neg(\bigvee_{e \in G(a)} grd(e))$$

In practice, the Csp2B tool computes the guards of each CSP event as predicates (see for example *grd_Tokens_ReqTokens(off)* in the B machine *Tokens* of section 2.1).

4.2 Results on our example

Figure 2 shows the concrete LTS directly produced by the FDR model-checker, respectively from the CSP description *Tokens* and *TokensRef*. The dotted ovals and lines show the refinement mapping. The dashed arrows are internal, or hidden, events.

In the following table we give for each node of the abstract LTS *Tokens*, the acceptance $G(a)$, the set $M^{-1}(a)$, and the set $G(M^{-1}(a))$:

a	$G(a)$	$M^{-1}(a)$	$G(M^{-1}(a))$
0	ReqTokens(O1) ReqTokens(O2)	0	ReqTokens(O1) ReqTokens(O2)
1	CollTokens(O1)	1, 7, 8	CollTokens(O1) ReqOff(O1) SendOff(O1)
2	CollTokens(O2)	2, 3, 4, 5, 6	CollTokens(O2) ReqOff(O2) SendOff(O2) QueryHome(home(1)) RecHome(home(1))

The set of hidden events is:
$H = \{ReqOff(O1), ReqOff(O2), SendOff(O1), SendOff(O2),$
$QueryHome(O1), QueryHome(O2), RecHome(O1), RecHome(O2)\}$
We can then generate the three following invariants:

((grd_Tokens_ReqTokens(O1) ∨ grd_Tokens_ReqTokens(O2)) ⇒
(grd_TokensRef_ReqTokens(O1) ∨ grd_TokensRef_ReqTokens(O2)))
∧
(grd_Tokens_CollTokens(O1) ⇒

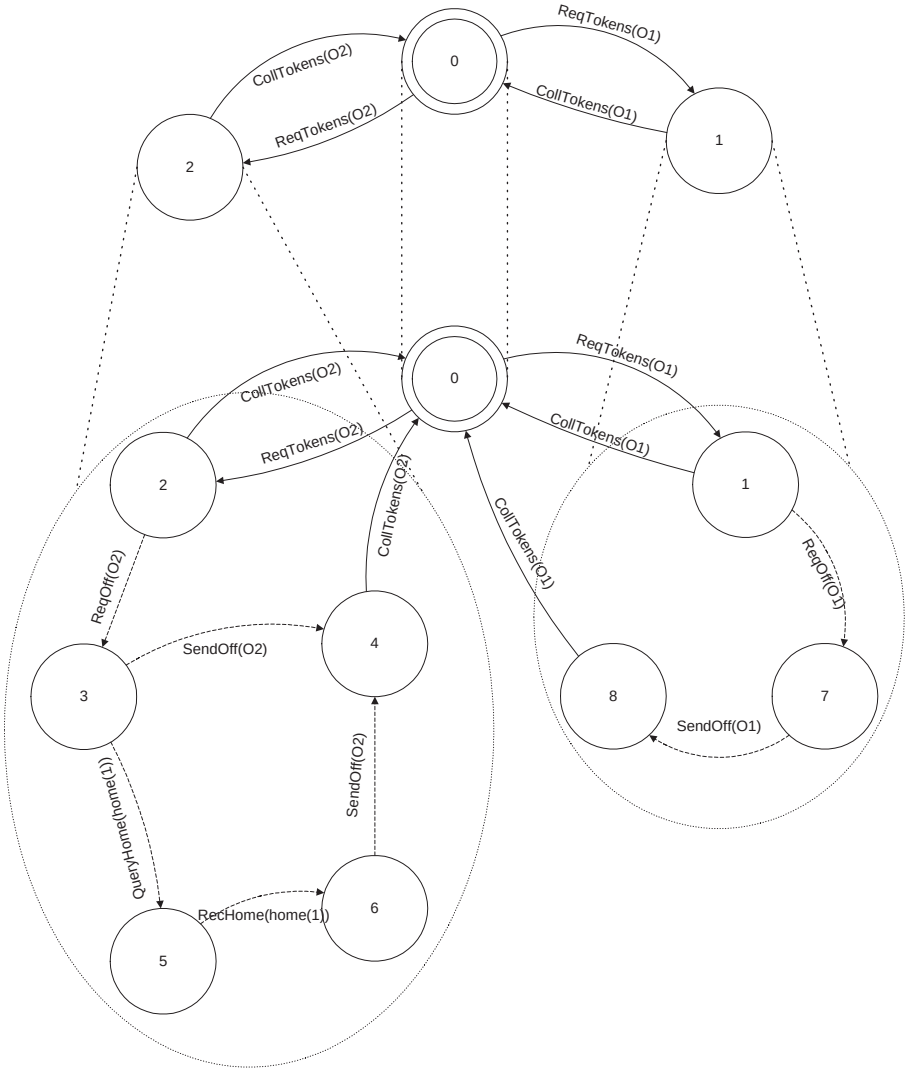

Fig. 2. Abstract and concrete LTS

$$
\begin{aligned}
&\textit{(grd\_TokensRef\_CollTokens(O1) or grd\_TokensRef\_ReqOff(O1)}\\
&\quad \vee\ \textit{grd\_TokensRef\_SendOff(O1)))}\\
\wedge\\
&\textit{(grd\_Tokens\_CollTokens(O2)} \Rightarrow\\
&\quad \textit{(grd\_TokensRef\_CollTokens(O2) or grd\_TokensRef\_ReqOff(O2)}\\
&\quad \vee\ \textit{grd\_TokensRef\_SendOff(O2) or grd\_TokensRef\_QueryHome(home(1))}
\end{aligned}
$$

$\vee$ *grd_TokensRef_RecHome(home(1))))*

Unfortunately, these invariants and the one defined in the section 2.3 (*tokens* = *ctokens* + *otokens*(*O1*) + *otokens*(*O2*)) are not sufficient to prove in B that the B refinement *TokensRef* refines the B machine *Tokens*. We have the same problem with the **CollTokens** operation as in our example of the section 2.3.

Indeed, the previous approach provides only invariants on the variables introduced by Csp2B. We need to add the following invariants, which give conditions on the amount of tokens in different states, for example the first one expresses that when the home office has send the tokens to the centre, there are no more tokens at the home office:

$$
\begin{array}{l}
\quad ((Customers = Answer\_2) \Rightarrow otokens(home(1)) = 0) \\
\wedge \\
\quad ((Customers = Collect \vee Customers = Answer\_1) \Rightarrow ctokens = 0) \\
\wedge \\
\quad ((\neg (Customers = Collect) \vee off\_1{=}O1) \Rightarrow otokens(O2) = 0) \\
\wedge \\
\quad ((Customers = Collect \wedge off\_1 = O2 \wedge tokens{>}0) \Rightarrow otokens(O2) {>}0)
\end{array}
$$

With these additional invariants added by hand, it is possible to verify the refinement completely. In this verification, 181 proof obligations were generated with the AtelierB tool, of which 36 were proved manually, the others automatically.

5 Parallel decomposition

The previous step of refinement has shown different parts in our system:

- a single *Centre*,
- some *Offices*, which can be the home-office of a customer.

A new step of refinement implements this decomposition: in a B refinement we include a machine for each part and we define operations as interactions of the operations of each part. For example the B refinement *TokensRefRefActs* includes the B machine *Centre* and *Offices* (with the renaming in *ce* and *oo* respectively):

REFINEMENT *TokensRefRefActs*

REFINES *TokensRefActs*

SEES *TokensDef*

INCLUDES *ce.Centre*, *oo.Offices*

INVARIANT
$ctokens = ce.ctokens \wedge$
$otokens(O1) = oo.otokens(O1) \wedge otokens(O2) = oo.otokens(O2)$

OPERATIONS

ReqTokens_Act(*off*) = **oo.ReqTokens**(*off*) ;

toks ← **CollTokens_Act**(*off*) = *toks* ← **oo.CollTokens**(*off*) ;

ReqOff_Act(*off*) =
  PRE *off* ∈ *OFFICE* **THEN ce.ReqOff**(*off*) || **oo.ReqOff**(*off*) **END**;

SendOff_Act(*off*) =
  PRE *off* ∈ *OFFICE* **THEN**
    ce.SendOff(*off*) || **oo.SendOff**(*off*, *ce.ctokens*)
  END;
...

END

In a Csp2B notation we can express decomposition as a parallel composition of several processes:

REFINEMENT *TokensRefRef*

REFINES *TokensRef*

SEES *TokensDef*

CONJOINS *TokensRefRefActs*

ALPHABET
ReqTokens(off:OFFICE)
toks ← *CollTokens(off:OFFICE)*
SendOff(off: OFFICE)
RecHome(off: OFFICE)
ReqOff(off: OFFICE)

QueryHome(off: OFFICE)

PROCESS *Centre* = **CentreAsleep**
CONSTRAINS *SendOff(off)* *RecHome(off)*
ReqOff(off) *QueryHome(off)*
WHERE

CentreAsleep $\widehat{=}$ *ReqOff?off* → **CentreAnswer(off)**

CentreAnswer(ce_off : OFFICE) $\widehat{=}$
IF *not(ce_off=home(1))* **THEN** (*QueryHome.home(1)*
→ *RecHome.home(1)* → *SendOff.ce_off* → **CentreAsleep)**
ELSE (*SendOff.ce_off* → **CentreAsleep**)
END
END

PROCESS *Offices* = **OfficeAsleep**
CONSTRAINS *ReqTokens(off)* *CollTokens(off)*
SendOff(off) *ReqOff(off)*
WHERE

OfficeAsleep $\widehat{=}$ *ReqTokens?off* → **OfficeRequest(off)**

OfficeRequest(oo_off : OFFICE) $\widehat{=}$
IF *otokens(oo_off)* > *0*
THEN *CollTokens.oo_off* → **OfficeAsleep END**
□ *ReqOff.oo_off* → *SendOff.oo_off* →
CollTokens.oo_off → **OfficeAsleep**

PROCESS *Home* = **HomeAsleep**
CONSTRAINS *RecHome(off)* *QueryHome(off)*
WHERE

HomeAsleep $\widehat{=}$ *QueryHome?off* → **HomeRequest(off)**

HomeRequest(ho_off : OFFICE) $\widehat{=}$
RecHome.(ho_off → **HomeAsleep**
END

END

We can apply exactly the same approach as with simple data refinement: we define a refinement mapping between the concrete and the abstract LTSs, and then we build the abstraction invariant from this mapping and the acceptance sets.

For our example, we obtain for *TokensRefRef* the same LTS as with *TokensRef* (indeed the FDR tool proves they are equivalent).

The following table describes the refinement mapping and the acceptance sets; the set of hidden events is empty:

a	$G(a)$	$M^{-1}(a)$	$G(M^{-1}(a))$
0	ReqTokens(O1) ReqTokens(O2)	0	ReqTokens(O1) ReqTokens(O2)
1	CollTokens(O1) ReqOff(O1)	1	CollTokens(O1) ReqOff(O1)
2	CollTokens(O2) ReqOff(O2)	2	CollTokens(O2) ReqOff(O2)
3	SendOff(O2) QueryHome(home(1))	3	SendOff(O2) QueryHome(home(1))
4	CollTokens(O2)	4	CollTokens(O2)
5	RecHome(home(1))	5	RecHome(home(1))
6	SendOff(O2)	6	SendOff(O2)
7	SendOff(O1)	7	SendOff(O1)
8	CollTokens(O1)	8	CollTokens(O1)

The sets, variables and guards automatically computed by the Csp2B tool are :

REFINEMENT *TokensRefRef*

REFINES *TokensRef*

SEES *POsets*

INCLUDES *TokensRefRefActs*

SETS
CentreState= {*CentreAsleep*,*CentreAnswer*,*CentreAnswer_1*,
CentreAnswer_2};
OfficesState= {*OfficeAsleep*,*OfficeRequest*,*OfficeRequest_1*,*OfficeRequest_2*};
HomeState = {*HomeAsleep*,*HomeRequest*}

VARIABLES
Centre, *ce_off*, *Offices*, *oo_off*, *Home*, *ho_off*

DEFINITIONS
grd_TokensRefRef_CollTokens(off) ==
((*mo.otokens* (*oo_off*) > 0 ∧ *Offices* = *OfficeRequest* ∧ *off* = *oo_off*)
∨ (*Offices* = *OfficeRequest_2* ∧ *off* = *oo_off*)) ;

grd_TokensRefRef_ReqTokens(off) == (*Offices* = *OfficeAsleep*) ;

grd_TokensRefRef_SendOff ==
(((*Centre* = *CentreAnswer_2* ∧ *off* = *ce_off*)
∨ (¬ (¬ (*ce_off* = *home*(1))) ∧ *Centre* = *CentreAnswer*

$\wedge$ *off* = *ce_off*))
$\wedge$ (*Offices* = *OfficeRequest_1* $\wedge$ *off* = *oo_off*)) ;

grd_TokensRefRef_RecHome ==
((*Centre* = *CentreAnswer_1* $\wedge$ *off* = *home* (1))
$\wedge$ (*Home* = *HomeRequest* $\wedge$ *off* = *ho_off*)) ;

grd_TokensRefRef_ReqOff ==
((*Centre* = *CentreAsleep*)
$\wedge$ (*Offices* = *OfficeRequest* $\wedge$ *off* = *oo_off*)) ;

grd_TokensRefRef_QueryHome ==
(($\neg$ (*ce_off* = *home* (1)) $\wedge$ *Centre* = *CentreAnswer*
$\wedge$ *off* = *home* (1))
$\wedge$ (*Home* = *HomeAsleep*))

...

END

With the abstraction invariants defined following our approach and those defined in the B refinement *TokensRefRefActs*, 398 proof obligations are generated by AtelierB, of which 41 have been proved manually, the others automatically.

6 Conclusion

In this paper, we are interested in an existing approach which combines CSP processes and B descriptions. A tool allows us to generate automatically from this combination B abstract machines or their refinements. However to check refinement we have to define some abstraction invariants to link abstract variables to concrete ones. We have proposed an approach that generates automatically some of these invariants. This approach is based on the labelled transition systems obtained by model-checking the CSP processes. A classical B proof step is then applied to verify refinement of the generated B machines. Moreover, when the conjoined B machines do not depend on the CSP processes, this last refinement step can be discharged. Otherwise, additional invariants must be added manually before verifying refinement.

Acknowledgements. We would like to thank the other members of the ABCD project for fruitful discussions and useful coments on this work. We are especially grateful to Michael Butler for his explaination on the Csp2B approach and his comments on an early version of this paper.

References

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

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2), May 1991.

[B-C99] B-Core (UK) Limited, Oxon, UK. *B-Toolkit, On-line manual.*, 1999. Available at 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. Available at http://www.dsse.ecs.soton.ac.uk/.

[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. Available at http://www.dsse.ecs.soton.ac.uk/.

[DS00] J. Derrick and G. Smith. Structural refinement in object-z / csp. In *IFM'2000 (Integrated Formal Methods)*, volume 1945 of *LNCS*. Springer-Verlag, 2000. Available at http://www.cs.ukc.ac.uk/research/tcs/index.html.

[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/.

[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. Available at http://www.dsse.ecs.soton.ac.uk/.

[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.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[Ste96] Steria, Aix-en-Provence, France. *Atelier B, User and Reference Manuals*, 1996. Available at 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. Available at http://www.iro.umontreal.ca/labs/teleinfo/PubListIndex.html.