Combining B Tools for Multi-Process Systems Specification

Christian Attiogbé

LINA - FRE CNRS 2729
Universté de Nantes
France
Christian.Attiogbe@univ-nantes.fr

ABSTRACT. We introduce a systematic approach that combines process-oriented design and abstract systems to specify multi-process system within B. The approach enforces the conjoined use of both theorem proving technique and model checking technique with the associated tools: AtelierB and ProB. This approach makes it easy the specification in Event B of multi-process systems and it also fasters the correctness proof. Our proposal is illustrated with a case study: a multi-process system with the interaction between processes that deal with parts using common resources.

RÉSUMÉ. Nous présentons une démarche systématique combinant la description de processus communicants et les systèmes abstraits pour la spécification en B de systèmes à multi-processus. La démarche préconise l'utilisation conjointe des approches de type preuve de théorème (*theorem proving*) et évaluation de modèle (*model checking*) avec les outils associés AtelierB et ProB. Cette démarche facilite la spécification en B de systèmes multi-processus et permet d'accélérer leurs preuves de correction. Notre proposition est illustrée avec un cas d'étude : un système multi-processus avec interaction entre des processus qui usinent des pièces en se servant de ressources communes.

KEYWORDS : Specification Approach, Interacting Processes, Event B, liveness, ProB

MOTS-CLÉS : Démarche de spécification, Processus communicants, Event B, vivacité, ProB

1. Introduction

The correct design and development of complex software systems is a major research topic as software systems are more and more present in everyday life and more and more complex according to their interactions with the environment. Process-oriented systems are well-suited as they handle the interaction features: concurrency and synchronization. There are several methods that enable the developer to specify and formally analyse her/his systems: finite state machines approaches, Petri nets, process algebra are the most popular (explicit) *state transition system* approaches. They are well researched, they have graphical notations and they are equipped with tools. The latter are often based on *model checking techniques* that explore the state space covered by the studied systems.
However the complexity of state transition systems increases quickly for real life systems; this depends on: the importance of both data and control parts (state space explosion); the number of processes involved in the system; the constraints on the resources used by the processes (sharing, limited number). Besides, there are the emergence of formal development techniques that emphasize the correct construction of software systems using refinement techniques: the B method [1] ranges in this category. Its extension known as Event B [2, 3], enables one to build more generally discrete event systems. The verification of properties and the refinement of B abstract specifications into concrete (executable) codes are supported by proof of properties: *theorem proving techniques* are used. The study of a system is a top-down approach going from an abstract model to a concrete one via refinements and decompositions. This approach is not always trivial even for small-size systems and it may be very tedious for large and complex systems.

The *motivation* of our work is the search of efficient *specification and design methods* that combine process-oriented approach with B method in order to provide more guidance to developer and to get mechanized techniques that scale well so as to be helpful for large interactive (possibly distributed) applications. An important part of today software applications, (Internet-)service oriented applications, process control systems are reactive (distributed) applications. They need to be safe and reliable; formal development techniques help in ensuring these properties.

The *contribution* of the current work is a systematic method (practical guidelines) to specify and verify multi-process systems by combining well-established formal techniques: process-oriented approach, Event B and model checking. We illustrate the proposed method by a case study with interacting processes. The case study reveals some interesting features of B specifications: for example, dealing with the dynamically variable number of the interacting processes.

In the Section 2 we introduce the used methods. The Section 3 is devoted to the proposed design support. In the Section 4 we summarize the results, we conclude the article and we introduce some perspectives.

2. Event B Method and Tools

2.1. Event B: an Overview

Within the Event B framework, asynchronous systems may be developed and structured using *abstract systems*. *Abstract systems* are the basic structures of the so-called *event-driven* B, and they replace the *abstract machines* which are the basic structures of

the earlier *operation-driven* approach of the B method[1]. An abstract system describes a mathematical model of an asynchronous system behaviour[1]; it is mainly made of a *space state* description (constants, properties, variables and invariant) and several *event* descriptions. Abstract systems are comparable to Action Systems [6]; they describe a non-deterministic evolution of a system through guarded actions. Dynamic constraints can be expressed within abstract systems to specify various liveness properties [3]. Abstract systems may be refined like abstract machines.

SYSTEM $\mathcal{S}$
SETS CS, SS
VARIABLES gv
INVARIANT Inv
INITIALISATION U
EVENTS
ee_1 $\widehat{=}$ /* an event */
  ANY bv WHERE
    $P^1_{(bv,gv)} \wedge P^2_{(gv)}$
  THEN
    $GS_{(gv,bv)}$
  END
; ee_2 $\widehat{=}$
  SELECT $P_{(gv)}$
  THEN $GS_{(gv)}$
  END
END

Within the B approach, an event is considered as the observation of a system transition. Events are spontaneous and show the way a system evolves. An event e is modelled with a guarded substitution: $e \mathrel{\widehat{=}} eG \Longrightarrow eB$ where the predicate eG is a *guard* and the substitution eB is an *action*. It may occur or may be observed only when its guard holds. The guard may use local variables (bv) bound to the ANY substitution and global variables (gv). The shape of an abstract system is given beside. The semantics of an abstract system lies on its invariant and is guaranteed by proof obligations. The consistency of the system is established by proof obligations:
i) the initialization establishes the invariant: $[U]Inv$; *ii)* each event preserves the invariant: $Inv \wedge eG \Longrightarrow [eB]Inv$.

Moreover the events terminate: $I \wedge eG \Rightarrow \mathsf{fis}(eB)$. The predicate $\mathsf{fis}(S)$ expresses that S does not establish *False*: $\mathsf{fis}(S) \Leftrightarrow \neg\ [S]False$. $\mathsf{prd}_v(S)$ is the before-after predicate of the substitution S; it relates the values of the state variable just before (v) and just after (v') the substitution S. The deadlock-freeness should be established for an abstract system (the disjunction of the event guards should be true).
The event-based semantics of an abstract system (A) is viewed as the event traces of A ($traces(A)$); the set of finite event sequences generated by the evolution of A.
As far as practical specification guides are concerned, we introduced in [4, 5] a parallel composition operator to supplement the classical top-down approach of the Event B. This parallel composition allows communication through global variables shared by the abstract (sub)systems. This permits a bottom-up approach for the Event B. This parallel composition is commutative and associative; therefore it is also defined for n systems. More details on the parallel composition can be found in [5].
The B method is supported by theorem provers which are industrial tools (Ateler-B and B-Toolkit[2]). The Event B extension has not dedicated tools but the specifications are translated into classical B and the standard B tools are used.

2.2. Overview of the ProB Tool

The ProB tool [9] is an animator and a model checker for B specifications. It provides functionalities to display graphical view of automata. It supports automated consistency

1. A system behaviour is the set of its possible state transitions beginning from an initial state.
2. www.clearsy.com, www.b-core.com

checking of B specifications (an abstract machine or a refinement with its state space, its initialization and its operations). The consistency checking is performed on all the reachable states of the machine. The ProB also provides a constraint-based checking; with this approach ProB does not explore the state space from the initializations, it checks whether applying one of the operation can result in an invariant violation independently from the initialization. The ProB offers many functionalities. The main ones are organized within three categories: *Animation*, *Verification* and *Analysis*. Several functionalities are provided for each category but here, we just list a few of them which are used in this article.
Random Animation: it starts from an initial state of the abstract machine and then, it selects in a random fashion one of the enabled operations, it computes the next state accordingly and proceeds the animation from this state with one of the enabled operations;
View/Reduced Visited States: it displays a minimized graph of the visited states after an animation;
View/Current State: it displays the current state which is obtained after the animation.
Temporal Model Checking: starting from a set of initialization states (initial nodes), it systematically explores the state space of the current B specification. From a given state (a node), a transition is built for each enabled operation and it ends at a computed state which is a new node or an already existing one. Each state is treated in the same way.
Compute Coverage: the state space (the nodes) and the transitions of the current specification are checked, some statistics are given on deadlocked states, live states[3], covered and uncovered operations.
Analyse Invariant: it checks if some parts of the current invariant are true or false;
The ProB tool is used to check liveness properties. Besides, note that if a B prover has been used to perform consistency proof, the invariant should not be violated; the B consistency proof consists in checking that the initialization of an abstract machine establishes the invariant and that all the operations preserve the invariant. In the case where the consistency is not completely achieved ProB can help to discover the faults.

3. A Design Support for Multi-Process Systems

A given system may be efficiently specified and verified in B by combining a process-oriented approach and the Event B tools ProB and AtelierB. We introduce a case study to illustrate our proposal.

3.1. A Case Study

The system to be studied comes from Milner [10]. The system (called JHM) is made of jobbers and tools (hammer, mallet) and some conveying belts. Two jobbers have to treat the incoming parts (coming on an input belt). The jobbers may use a hammer or a mallet to work the parts. There is only one hammer and only one mallet (they are shared tools/resources). The task of a jobber is as follows: it gets a part, then it gets one of the (free) tool, it deals with the part, it frees the used tool and then the jobber becomes available for another task. The already worked parts leave the system (via an output belt).

We consider an extension of this *Jobber processes* case study: not only two jobbers are considered but several ones. Moreover, new jobbers may enter the system at any time. Now think about the complexity of the specification with a state transition approach. To

3. those already computed

deal with this complexity, very often state transition approaches make some restrictions on the data and the number of interacting processes.

The formal specification of the case study in Event B is not straightforward. We have to build the right B abstract system: a state space with a set of events that describe the system evolution.

3.2. Practical Limitations of the Separated Approaches

The advantages of each one of the methods involved in the study are underlined above.

State Transitions. Capturing a process behaviour is intuitive but state transition systems lack high level structures for complex processes. Handling an undefined and variable number of processes is not tractable. Dealing with several instances of the same processes is not possible. Synchronization of processes should be made explicit.

B Approach. A difficult point is that of completeness with respect to event ordering (liveness properties): did the specification covers all the possible evolution (event sequences) expressed in the requirement? Indeed one can have a consistent system (with respect to the stated invariant) which does not meet the desired logical behavioural requirements. Several works investigate the proof of liveness properties of B specifications [7, 8].

In the proposed approach, we show that the combination of both approaches help to fight their practical limitations.

3.3. The Proposed Approach

The focus is on the correct logical behaviour of the given system; its specification should fulfill the stated informal requirements. Accordingly it should ensure safety and the ordering of the system events should be the expected one. Several steps are distinguished.

Step 1. General frame of the abstract system

- Begin the construction of an abstract system $\mathcal{A}$ that is the formal model of the studied system. $\mathcal{A}$ is made of *S*, *E*, *follow*, *Evts* where *S* is the state space that will be described with variables and predicates, *E* is an event alphabet, *follow* is a transition relation on *E* and *Evts* is a set of event specifications. As $\mathcal{A}$ is a multi-process system, several process types will contribute to define its behaviour. These process types are described in the following.

- Identify the set $\mathcal{P}_r$ of (types of) processes that interact within $\mathcal{A}$. Put into the SET clause an abstract set *P* for each process type. For instance a process type *JOBBER* is considered for the current case study. It does not matter the number of the process of each type.

For each process type $P \in \mathcal{P}_r$ consider a new variable to model the subset of processes of this type: $jobbers \subseteq JOBBER$;

- Identify the system resources or data and distinguish the shared ones. Here we have a set of parts which are processed: *PART*; a set of tools: $TOOL = \{hammer, mallet\}$.

- Fill in the SET clause with the identified abstract sets: *PART*, *TOOL*, ...

The shared resources need a specific access policy. For each kind of these resources, define an *event* to access/get/free the resource. Typically an event that reads a common data is to be distinguished from the one that write the data. Each event has a guard which expresses the conditions and the constraints to get/free the resource. Those events are: $getHam, getMal, putHam, putMal$.

- Identify the set of events that make the system evolves: *E*. These events may be split into two classes of events: the *general events* (GE) that affect the whole system and the *process-specific events* (SE) which correspond to the evolution of the identified (type of) processes. This distribution of the events into classes favours the decomposition of the

abstract system since the events may be shared between subsystems.
As far as the case study is concerned, we have the following events: $GE = \{inPart, outPart, getPart\}$; $SE = \{getHam, getMal, putHam, putMal, processPart\}$.

Step 2. Modelling the state space: *S*

- Identify the global resources and properties with an invariant predicate that characterizes *S*. For the JHM we have a set of input parts which are processed (*inParts*), and the related output parts (*outParts*).
- Fill in the VARIABLES clause and the INVARIANT clause of $\mathcal{A}$ with the resource declarations and the identified properties.

$$jobbers \subseteq JOBBER \wedge inParts \subseteq PART \wedge outParts \subseteq PART$$

The identified relations and properties are gathered in the INVARIANT clause of $\mathcal{A}$.

Step 3. Describing event ordering: *follow*

Identify the properties of the behaviour of the whole system; that is a specific ordering of the events occurrence according to the requirements: *liveness* requirements.
Let $follow : E \leftrightarrow E$ be a relation that captures the required ordering of the events. It remains to complement *follow* according to the classes (GE, SE) of events in *E*.

Step 4. Defining the general events: $Evts_{GE}$

Complement $\mathcal{A}$ with the B specifications of the general events: *inputPart*, *outputPart*, $\cdots$.
The first one is specified as follows:

```
inputPart ≙            /* A new part enters the system */
  ANY part WHERE       /* input parts and output parts are disjoints */
    part ∈ PART ∧ part ∉ outParts
  THEN
    inParts := inParts ∪ {part}
  END
```

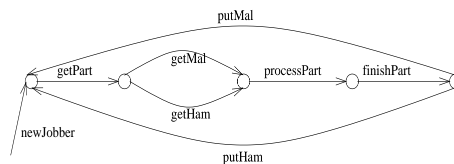
Step 5. Defining specific process behaviour: $Evts_{SE}$

For each process type $P \in \mathcal{P}_r$ we supplement $\mathcal{A}$ with $\mathcal{A}_P$ which is made of S_P, E_P, $follow_P$, $Evts_P$. Consider *JOBBER* for an illustration of *P*.

1) E_P: identify the subset of *SE* events that make the evolution of *P*;

2) $follow_P$: define an ordering relation on these events by considering the specific requirements on *P*; this results in describing a subset of *follow* related to *P*: $follow_P$. A rather small state transition system may help here.

$follow_{JOBBER}$	
getPart	$\{getHam, getMal\}$
getHam	$\{processPart\}$
getMal	$\{processPart\}$
processPart	$\{finishPart\}$
...	...



Some variables describing disjoint sets of processes are necessary. They capture the states of each process. This ensures (using the event guards) that a process will evolve according to its current state. For instance *readyJbrs* describes the set of processes that get a part and that get one tool (hammer or mallet).

3) $Evts_P$: describe each event of the current process *P* as a B event (guard and substitution). Increase the EVENTS clause of $\mathcal{A}$ with the new described event. The following example gives the specification of the *processPart* event.

```
processPart ≙            /* A part is processed*/
  ANY jbr WHERE          /* one jobber ready to process its part */
    jbr ∈ jobbers ∧ jbr ∈ readyJbrs
  ∧ jbr ∈ dom(hastool) ∧ jbr ∈ dom(hasPart)
  THEN
    workingj := workingj ∪ {jbr}
  || readyJbrs := readyJbrs − {jbr}
  END
```

In the same way the B specifications are described for all the events of the E_P alphabet.

After the current step, the abstract system $\mathcal{A}$ under construction is equipped with the B specifications of all the events.

Step 6. Complement and prove the consistency of the $\mathcal{A}$ abstract system using Atelier B (theorem proving aspect). All the proof obligations should be discharged. However we have not yet the way to guarantee the liveness requirements captured within *follow*. We shall prove that $traces(\mathcal{A})$ coincides with the *follow* relation. That is the role of the following step.

Step 7. Analyze and improve the $\mathcal{A}$ abstract system; this is achieved with the help of animation and model checking with the ProB tool. First, model-check (see 2.2) $\mathcal{A}$ to detect deadlocks. Correct the specification accordingly. When $\mathcal{A}$ is deadlock-free, check that it fulfills *follow*. Using the ProB analysis tool (see 2.2) check that all the events are enabled (this corresponds to no uncovered operations). Moreover, each event (evt) should enable the events in $follow(evt)$. This is checked by visualizing the reduced visited states (see 2.2). Another way to check this is by stepwise animations; the ProB displays the operations enabled by each activated operation; we should have the correspondence with *follow*. The A abstract system is improved accordingly.

As the *follow* relation expresses the set of all possible orderings of the events that could happen in the system, after the **Step 7.**, the logical behaviour analysis is complete and we get a *correct* specification of A with respect to the requirements.

Proof: The occurrences of event orderings of $\mathcal{A}$ are checked with respect to *follow*. Formally, the occurrence of an event $e_1 \mathrel{\widehat{=}} eG_1 \implies eB_1$ is: $\exists\, v_i, v_{i+1}.[v := v_i]eG_1 \wedge [v, v' := v_i, v_{i+1}]\mathsf{prd}_v(eB_1)$. The occurrence of a sequence of two events $e_1.e_2$ is:
$\exists\, v_i, v_{i+1}, v_{i+2}.[v := v_i]eG_1 \wedge [v, v' := v_i, v_{i+1}]\mathsf{prd}_v(eB_1) \wedge$
$[v := v_{i+1}]eG_2 \wedge [v, v' := v_{i+1}, v_{i+2}]\mathsf{prd}_v(eB_2)$. This generalizes easily to sequences of n events and it is $traces(\mathcal{A})$.
Note that the closure of *follow* (noted $follow^*$) is the event occurrences that correspond to the requirements captured by *follow*. Therefore we have $\boxed{follow^* = traces(\mathcal{A})}$.

4. Summary of Results and Conclusion

Using the described approach, we build incrementally a correct B abstract specification of the jobber case study. The feedbacks from model checking help to discover deadlocks and incompleteness in the earlier specifications. The specification is then improved step by step. The B theorem prover is used to discharge all the proof obligations related to the invariant. Liveness properties are expressed using the *follow* relation on events.The ProB tool is used to establish the correctness with respect to liveness. Moreover we handle the dynamic structure of the system as processes may enter and take part of the application at any time.

Conclusion. We presented a practical method as a design support to guide the specification of multi-process systems. The proposal is illustrated with the *jobbers* case study. The obtained specification is proved correct by combining theorem proving via Atelier B and model checking via the ProB tool.

The specificity of the case study, apart of the multi-process aspect, lies on the fact that it describes a generic kind of software systems: several interacting processes that share a set of resources. This is frequently encountered and yet well-known in operating systems: processes accessing the system resources (disks, files, cpu), the dinning philosophers, the readers/writers, etc. These systems are used as the basic blocks for building various software applications including Internet service-oriented ones. The proposed approach is yet experimented with several other case studies. We have not consider in this paper refinement of the specification into code, however this step remains in the scope of the standard B approach. But model checking with the ProB tool may also help to faster the discharging of refinement proof obligations. The short-term perspective of the work is about fairness properties management. Indeed, the liveness is established but fairness is not guarantee. This property is important in multi-process systems since a starved process may decrease the performance of the system. Think about a situation where a process (*jobber*) has one tool and then is not enabled to work (due to lack of fairness); then the tool it uses is not released to serve other processes. We are investigating the works described in [8] as a basis to complement our study with fairness aspects.

5. References

[1] J-R. Abrial. *The B Book*. Cambridge University Press, 1996.

[2] J-R. Abrial. Extending B without Changing it (for developping distributed systems). *Proc. of the 1st Conf. on the B method, H. Habrias (editor), France*, pages 169–190, 1996.

[3] J-R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In *Proc. of the 2nd Conference on the B method, D. Bert (editor)*, volume 1393 of *Lecture Notes in Computer Science*, pages 83–128. Springer-Verlag, 1998.

[4] C. Attiogbé. A Mechanically Proved Development Combining B Abstract Systems and Spin. In *Proceedings of the 4th International Conference on Quality Software (QSIC 2004)*, pages 42–49. IEEE Computer Society Press, 2004.

[5] C. Attiogbé. A Stepwise Development of the Peterson's Mutual Exclusion Algorithm Using B Abstract Systems. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *Proc. of ZB'05*, volume 3455 of *LNCS*, pages 124–141. Springer, 2005.

[6] R.J. Back and R. Kurki-Suonio. Decentralisation of Process Nets with Centralised Control. In *Proc. of the 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142. ACM, 1983.

[7] F. Bellegarde, S. Chouali, and J. Julliand. Verifi cation of Dynamic Constraints for B Event Systems under Fairness Assumptions. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB'2002 – Formal Specifi cation and Development in Z and B*, volume 2272 of *LNCS*, pages 477–496. Springer-Verlag, 2002.

[8] H. R. Barradas and D. Bert. A Fixpoint Semantics of Event Systems with and without Fairness Assumptions. In J.M.T. Romijn, G.P. Smith, and J.C. van de Pol, editors, *Proc. of IFM'2005*, volume 3771 of *LNCS*, pages 327–346, UK, 2005. Springer-Verlag.

[9] M. Leuschel and M. Butler. ProB: A Model Checker for B. In Keijiro A., Stefania G., and Dino M., editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer, 2003.

[10] Robin Milner. *Communication and Concurrency*. Prentice-Hall, NJ, 1989.