When a Formal Model Rhymes with a Graphical Notation

Akram Idani[1,2](✉) and Nicolas Stouls[3]

[1] LIG, University of Grenoble Alpes, 38000 Grenoble, France
[2] LIG, CNRS, 38000 Grenoble, France
Akram.Idani@imag.fr
[3] CITI-INRIA, Université de Lyon, INSA-Lyon, 69621 Lyon, Villeurbanne, France
Nicolas.Stouls@insa-lyon.fr

Abstract. Formal methods are based on mathematical notations which allow to rigorously reason about a model and ensure its correctness by proofs and/or model-checking. Unfortunately, these notations are complex and often difficult to understand from a human point of view especially for engineers who are not familiar with formal methods. Several research works have proposed tools to support formal models using graphical views. On the one hand, such views are useful to make formal documents accessible to humans, and on the other hand they ease the verification of some behavioral properties. However, links between graphical and formal models proposed by these approaches are often difficult to put into practice and depend on the targeted formal language. In this paper, we discuss these links from a practical approach and show how a behavioral description can be computed from a formal model based on two complementary paradigms: under-approximation (or animation-based) and over-approximation (or proof-based). We applied these paradigms in order to produce behavioural state/chart views from B models and we carried out an empirical study to assess the quality and relevance of these graphical representations for humans.

Keywords: B method · Symbolic LTS · Animation · Abstraction

1 Introduction

Several research works are devoted to bridge the gap between formal and semi-formal methods considering their complementary aspects and cross contributions. Indeed, on the one hand, semi-formal methods (thanks to their support for graphical notations such as UML) are synthetic, structuring and more intuitive for humans, and on the other hand, formal methods (thanks to their mathematical notations) are precise and support automated reasonings. These works were widely interested by translations from a semi-formal UML model to a formal specification: from UML to B [15], from UML to Z [10], from UML to Alloy [3], etc. Their main motivations are to provide precise semantics to UML notations in order to remedy the lack of tools for formally analyzing UML models.

C. Canal and A. Idani (Eds.): SEFM 2014 Workshops, LNCS 8938, pp. 54–68, 2015.
DOI: 10.1007/978-3-319-15201-1_4

Despite of these numerous tools dedicated to such translation, several companies have an established software development process entirely based on a formal method. For example, Siemens Transport [7], Clearsy [13], Gemplus [5] have used the B method as its core development method without any accompanying UML model. Indeed, since the targeted formal language is not object oriented, translations from UML often lead to a complex specification which is, on the one hand, far from what a developer could write directly, and on the other hand incomplete for a safety critical system. This motivates other kind of works to define a formal link between a formal model and its behavioral representation. For example, works of C. Snook and M. Butler [15] ant its support tool iUML-B [14], provide a graphical front end, used conjointly with a formal B specification in order to keep the distance between both formal and graphical models as thin as possible. We can also cite the ProB tool [11] which is an animator and a model-checker able to draw an accessibility graph after an exhaustive exploration of the specification state space. However, it considers concrete states rather than symbolic ones and the resulting graphical representation is complex because of the combinatorial explosion problem. In order to remedy this shortcoming, [12] provides some heuristics to reduce the accessibility graph size by using a symmetry analysis technique. Furthermore, it was not dedicated to make the focus on the understanding of some particular properties, since the abstract state space could not be provided by an expert user.

In this paper, the starting point is a B or Event-B formal model [1,2]. Our aim is to provide tools able to extract graphical views representing some properties of the formal model and hence increase its understanding by humans who are not trained with such a formal notation. We discuss and compare two paradigms: **under-approximation** (or animation-based) and **over-approximation** (or proof-based). We applied these paradigms in order to produce behavioural views from B models and then we carried out an empirical study to assess the quality and relevance of these graphical representations for humans.

In Sect. 2 we give a simple example in order to illustrate contributions of this paper. Section 3 discusses the under-approximation approach and presents an algorithm which improves automation of this technique. Section 4 describes the over approximation technique and presents the *GénéSyst*-tool. Results of our empirical study are discussed in Sect. 5. Finally, Sect. 6 summarizes our comparative study of both techniques and draws the conclusions and perspectives of this work.

2 Case-Study

Figure 1 gives a simple scheduler specification taken from [6] and written in B. It models exclusive access of processes to a unique resource. Variables *waiting*, *ready* and *active* model states of processes managed by the system. The set of all processes is an abstract set (set *PID*). An idle process which doesn't request access to the unique resource is introduced by the system using the *waiting* variable. Variable *ready* is the set of processes that have requested access to the resource. Finally, variable *active* contains the active process to which the resource is assigned. Evolutions of these variables are performed by three events. Event NEW(pp) creates a new waiting process. Event READY(pp) changes process pp from the waiting state to the ready state. If there is no active process it directly activates process pp. Finally, event SWAP puts the active process in the waiting state and activates non-deterministically some ready process.

```
SYSTEM SCHEDULER
SETS
  PID
VARIABLES
  active, ready, waiting
INVARIANT
  active ⊆ PID ∧ ready ⊆ PID ∧ waiting ⊆ PID ∧
  ready ∩ waiting = ∅ ∧ active ∩ waiting = ∅ ∧ active ∩ ready = ∅ ∧
  card(active) ≤ 1 ∧ (active = ∅ ⇒ ready = ∅)
INITIALISATION
  active, ready, waiting := ∅, ∅, ∅
EVENTS
  NEW(pp)  ≙      SELECT (pp ∈ PID) ∧ (pp ∉ (ready ∪ waiting ∪ active))
                  THEN   waiting := waiting ∪ {pp}
                  END;
  READY(pp)  ≙    SELECT pp ∈ waiting
                  THEN   waiting := (waiting - {pp})||
                         IF (active = ∅) THEN active := {pp}
                         ELSE  ready := ready ∪ {pp}
                         END
                  END;
  SWAP  ≙         SELECT active ≠ ∅
                  THEN   waiting := waiting ∪ active ||
                         IF (ready = ∅) THEN active := ∅
                         ELSE
                             ANY pp WHERE pp ∈ ready THEN
                                 active := {pp} || ready := ready - {pp}
                             END
                         END
                  END
END
```

Fig. 1. Scheduler Specification from [6]

A palette of graphical representations that can be issued from the scheduler example can be found in [8]. These representations provide a graphical documentation of the behaviour of B specifications and allow to identify different viewpoints potentially useful for humans. IFor example, a B analyst may be interested by a graphical representation of the SCHEDULER that intuitively show a process life cycle. Hence, the abstract graphical view may deal with three states corresponding to the fact that a process p_i (such that $p_i \in PID$) is in state *waiting*, *ready* or *active*. In other words, states of the abstract view are: (1) $p_i \in waiting$, (2) $p_i \in ready$, and (3) $p_i \in active$. Figure 2, built manually, gives an abstract view of the SCHEDULER system based on these three states.

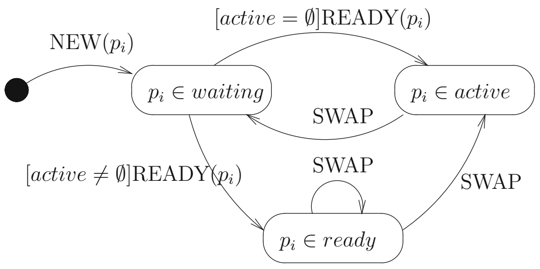

NEW(p_i)
[$active = \emptyset$]READY(p_i)
$p_i \in waiting$
$p_i \in active$
SWAP
[$active \neq \emptyset$]READY(p_i)
SWAP
SWAP
$p_i \in ready$

Fig. 2. Example of an intuitive abstract view of the SCHEDULER system

From a documentation point of view the interest of this representation is to emphasize graphically some intrinsic properties of the SCHEDULER system, for example:

- The process equity property, indicating that every process may be activated, is not verified by the specification. Indeed, Fig. 2 shows that in state $p_i \in ready$, SWAP has a non-deterministic behaviour justified by the existence of two transitions with the same label. This means that event $SWAP$ can block a process p_i indefinitely in state $p_i \in ready$.
- The non-blocking property, indicating that after being active a process does not stop the system is verified by the specification. Indeed, in Fig. 2 the transition $SWAP$ is triggered on from the state ($p_i \in active$) and always leads to state ($p_i \in waiting$).

This paper shows how these graphical representations can be extracted automatically using two kinds of techniques: **under-approximation** (or animation-based) and **over-approximation** (or proof-based).

3 Under-Approximation Approach

Under-approximation is based on exploration of a useful subset of the state space. We apply this technique in order to draw a graphical representation which is useful from a documentation point of view but which may miss some behaviours.

3.1 Construction Method and Usability Constraints

One way to build an under-approximating graphical abstraction is to exhaustively explore the concrete state space of the B specification and then to apply an abstraction algorithm to group concrete states. For a bounded state space, animators such as ProB [11] can help to explore all states. In other cases, such as for the SCHEDULER example, we must start by bounding unbounded elements (i.e. specifying PID with a bounded set). If we introduce only two processes p_1 and p_2 in the system, we obtain ten accessible states (Fig. 3). If the number of processes increases, the accessibility graph becomes too large and difficult to understand. For example, having $PID = \{p_1, p_2, p_3\}$ we obtained thirty five accessible states with numerous transitions.

3.2 Graph Abstraction Algorithm

We note $\mathcal{G} = (N, T)$ an accessibility graph issued from a B system, where N is the set of concrete states of graph $\mathcal{G}$, and T is the set of transitions between states of N. A concrete state S_v ($S_v \in N$) gives particular values assigned to state variables v ($v = \{v_1, \ldots, v_n\}$) of the B system. Consequently, each state s

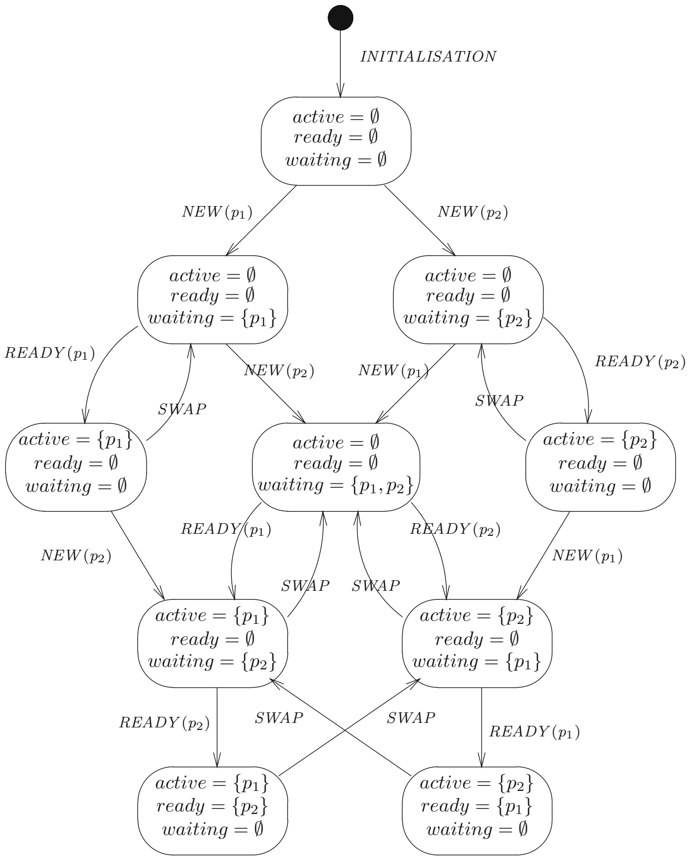

INITIALISATION
active = ∅ ready = ∅ waiting = ∅
NEW(p1)
NEW(p2)
active = ∅ ready = ∅ waiting = {p1}
active = ∅ ready = ∅ waiting = {p2}
READY(p1)
SWAP
NEW(p2)
NEW(p1)
SWAP
READY(p2)
active = {p1} ready = ∅ waiting = ∅
active = ∅ ready = ∅ waiting = {p1, p2}
active = {p2} ready = ∅ waiting = ∅
NEW(p2)
READY(p1)
SWAP
SWAP
READY(p2)
NEW(p1)
active = {p1} ready = ∅ waiting = {p2}
active = {p2} ready = ∅ waiting = {p1}
READY(p2)
SWAP
SWAP
READY(p1)
active = {p1} ready = {p2} waiting = ∅
active = {p2} ready = {p1} waiting = ∅

Fig. 3. Accessibility Graph of the SCHEDULER for $PID = \{p_1, p_2\}$

can be formally expressed by a predicate $P(S_v)$ as the conjunction of equality predicates that associate to each state variable v_i its value in S_v:

$$P(S_v) \triangleq \bigwedge_{i=1}^{n} (v_i = val_j)$$

Where val_j is a value of v_i allowed by the invariant. A concrete state S_v satisfies an abstract state $S_{abstract}$ (noticed $S_v \vdash S_{abstract}$), defined by a predicate R (*e.g.* $p_1 \in ready$), if and only if we can prove that $P(S_v) \Rightarrow R$.

Hence, according to an accessibility graph and a set of abstract states, the following algorithm can produce a symbolic representation by grouping concrete states satisfying a same abstract state predicate. The inputs are: (i) an accessibility graph $\mathcal{G} = (N, T)$ and (ii) a set of abstract state predicates $N_{abstract}$ (Fig. 4).

```
For each abstract state $S_{abstract}$ from $N_{abstract}$ do
|   $N := N \cup \{S_{abstract}\}$
|   For each concrete state $S_v$ such that
|   |       $S_v \in N - N_{abstract} \ \wedge \ S_v \vdash S_{abstract}$
|   do
|   |   For each state $S$ and transition $t$ such that
|   |   |   $S \in N \ \wedge \ t \in T \wedge t = S \xrightarrow{o} S_v$
|   |   do
|   |   |   $t := (S \xrightarrow{o} S_{abstract})$
|   |   End do
|   |   For each state $S$ and transition $t$ such that
|   |   |   $S \in N \ \wedge \ t \in T \wedge t = S_v \xrightarrow{o} S$
|   |   do
|   |   |   $t := (S_{abstract} \xrightarrow{o} S)$
|   |   End do
|   |   $N := N - \{S_v\}$
|   End do
|   Delete redundant transitions
End do
```

Fig. 4. Under approximation algorithm.

The algorithm checks each concrete state against each abstract state predicate, using the *AtelierB* prover. If the proof succeeds, then an abstract state has been found for the concrete state. The next step in the construction of the abstract state-transition diagram is to identify the transitions. Since each node of the concrete graph corresponds to a node of the abstract diagram, each transition of the concrete graph can be translated into a transition in the abstract diagram. In order to decrease the number of transitions, the tool groups all transitions which correspond to the same pair of nodes, and to the same B event.

Our algorithm links concrete states to abstract ones, and hence the nodes of the abstract state-transition diagram are: (a) the abstract state predicates given by the user, and (b) the concrete nodes which don't appear in the domain of the abstraction function. This guarantees that each concrete node will correspond to a node of the abstract diagram. Furthermore, in order to obtain a relevent abstract view, two conditions should be verified:

1. abstract state predicates are disjoint, *i.e.* each concrete state corresponds to at most one abstract state.
2. abstract state predicates cover all the state space allowed by the invariant, *i.e.* the nodes of the abstract diagram only correspond to the abstract predicates.

The abstract view of Fig. 2 respects only the first condition because it misses all concrete states reached from the initialization. These states can be grouped in an abstract state $p_i \notin (\textit{waiting} \cup \textit{active} \cup \textit{ready})$ which is reached when the system is initialized. The left hand side of Fig. 6 shows the result of this technique when applied to accessibility graph of Fig. 3 in which set PID contains two processes p_1 and p_2.

4 Over-Approximation Approach

The under-approximation technique is useful when the accessibility graph explores a relevant finite subset of state space from which we can exhibit a useful abstract view for a documentation purpose. If some interesting behaviours are not included in the concrete graph, they will not appear in the abstract diagram. An over-approximation technique is then more interesting because it allows to produce a symbolic transition system that represent a potentially infinite set of values. Such tools reason on event *enabledness* and state *reachability* properties.

4.1 Construction Method and Usability Constraints

Our objective is to directly compute an abstract view from the B model properties, rather than to reason on a concrete graph. For instance, if an over-approximation view shows that a state is not reached by any transition, then one can conclude that associated concrete valuations could not be reachable by any execution of the B model.

Our approach tries first to prove, for each event e and each couple of abstract states S_1 and S_2, that no execution of event e from state S_1 can reach state S_2. This goal is a proof obligation (PO) assuming that if state predicate $P(S_1)$ is true then event e establishes the negation of state predicate $P(S_2)$:

$$P(S_1) \Rightarrow [e]\neg P(S_2)$$

This first step allows to identify by proofs, all uncrossable transitions between states S_1 and S_2. In fact, if the above PO is solved, then we assert that event e never reaches S_2 from S_1. Variations of this PO allow to compute whether S_2 is always or possibly reached by e from S_1:

- S_2 always reached from S_1: $P(S_1) \Rightarrow [e]P(S_2)$
- S_2 possibly reached from S_1: $P(S_1) \Rightarrow \neg[e]\neg P(S_2)$.

For example, the following proofs (but not only) should succeed for event SWAP[1]:

- it always deactivate an active process: $(p_i \in \mathit{active}) \Rightarrow [\mathrm{SWAP}](p_i \in \mathit{waiting})$
- it never activate a waiting process: $(p_i \in \mathit{waiting}) \Rightarrow [\mathrm{SWAP}](p_i \notin \mathit{active})$
- it may activate a ready process: $(p_i \in \mathit{ready}) \Rightarrow \neg[\mathrm{SWAP}](p_i \notin \mathit{active})$

As for the under-approximation approach, two conditions must be verified: abstract state predicates are disjoint and cover all the state space allowed by the invariant. The first condition avoids states overlapping and the second one allows to have a global view on the complete system. An important proof obligation is then to establish the completeness of the state predicates according to the invariant:

$$I \implies \bigvee_{i=1}^{n} P(S-v)$$

4.2 The *GénéSyst* Tool

The *GénéSyst* tool[2] [4] implements the ideas of this approach. It computes a Symbolic Labelled Transition System (SLTS) describing all possible behaviours of a given event-B model, according to a given set of disjoint state predicates. Generated proof obligations are discharged by means of the *AtelierB* automatic prover.

The overall *GénéSyst* algorithm is presented in Fig. 5, where we distinguish transitions from the initialization, and transitions associated to other events. In this algorithm, conditions are written under a negative form (*i.e.* if $\neg A$ can not be established), since a formula that has not been proved is not necessarily true. In this algorithm, no any information is presented to consider simplification of the conditions. The reader can refer to [4] for further semantical details.

In order to restrict the undecidability problem of proofs, heuristics are used to compute the over-approximation graph (the SLTS). One of them is to split proofs into two parts: ***enabledness*** and ***reachability***. In this approach, for each pair of abstract states S_1 and S_2, and each event e, a transition t of the SLTS is defined by $(S_1 \overset{(D,A,e)}{\longrightarrow} S_2)$, where D is the *enabledness* condition (condition under which the event e can be triggered from S_1) and A is the *reachability* condition (condition under which the event e can reach the state S_2). We define *enabledness* and *reachability* as follows:

- *Enabledness* condition D : $P(S_1) \Rightarrow (D \Leftrightarrow \mathit{guard}(e))$
- *Reachability* condition A : $P(S_1) \ \wedge \ D \Rightarrow (A \Leftrightarrow \neg[\mathit{action}(e)]\neg P(S_2))$.

[1] These properties are not all properties of event SWAP.

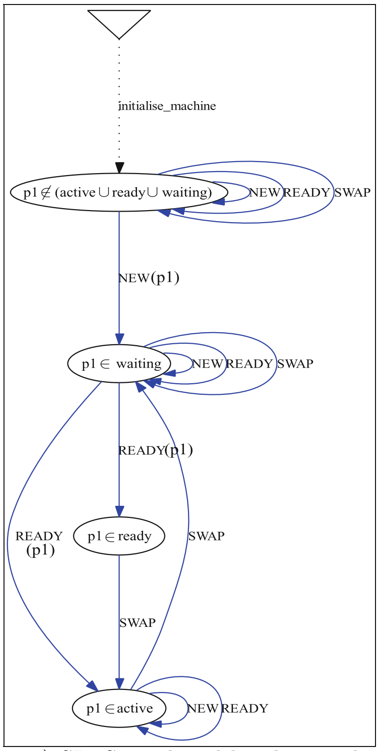
[2] *GénéSyst*: http://perso.citi.insa-lyon.fr/nstouls/?ZoomSur=Logiciels.

```
// Initialization of the result: set of transitions is empty
$T$ := $\emptyset$

// Defining states reachable by the initialization
For each state $S$, except $S_{init}$ do
|    // Define the condition $A$ under which the initialization reaches $S$
|    $A := \neg[init]\neg P(S)$
|    If $\neg A$ can not be established then
|    |    $T := T \cup \{(S_{init} \overset{(true,A,init)}{\longrightarrow} S)\}$
|    End if
End do

// Defining existing transitions
For each state $S_1$, except $S_{init}$, and each event $e$ do
|    //Define the condition $D$ under which $e$ can be trigger from $S_1$
|    $D := guard(e)$
|    If $(P(S_1) \Rightarrow \neg D)$ can not be established then
|    |    For each state $S_2$, except $S_{init}$, do
|    |    |    // Define the condition $A$ under which event $e$ reaches $S_2$
|    |    |    $A := \neg[action(e)]\neg P(S_2)$
|    |    |    If $(P(S_1) \wedge D \Rightarrow \neg A)$ can not be established then
|    |    |    |    $T := T \cup \{(S_1 \overset{(D,A,e)}{\longrightarrow} S_2)\}$
|    |    |    End if
|    |    End do
|    End if
End do
```

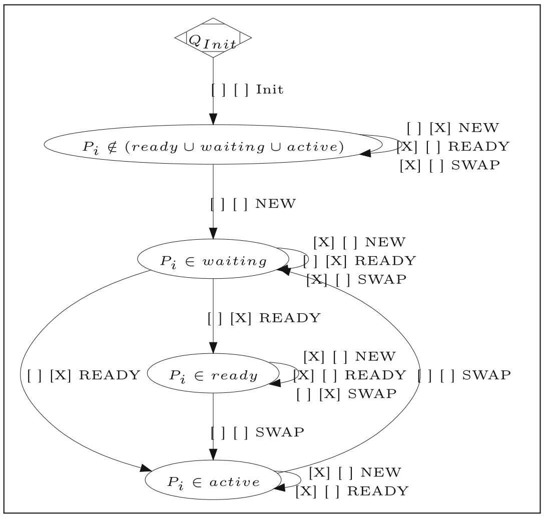
Fig. 5. *GénéSyst* main algorithm

In the context of the starting state, the *enabledness* condition D is equivalent to the B event guard (denoted $guard(e)$). Technically, the tool asks *AtelierB* to prove if D can be reduced to *true*, by trying to proof the assertion $I \wedge P(S_1) \Rightarrow guard(e)$, where I is the invariant. If this proof succeeds, it concludes that e can always be enabled from S_1; otherwise it asks *AtelierB* to prove if D can be reduced to $false$, by trying to proof $I \wedge P(S_1) \Rightarrow \neg guard(e)$. If this second proof succeeds then the tool concludes that e can't be enabled from S_1. The same principle is applied for the *reachability* property. Condition A is equivalent to $\neg[action(e)]\neg P(S_2)$ which means that if A is reduced to true, then e may reach S_2. But if it is reduced to $false$ then e cannot reach S_2. By the way, a transition $(S_1 \overset{(D,A,e)}{\longrightarrow} S_2)$ is said valid if and only if $\exists x \cdot (P(S_1) \wedge D \wedge A)$.

Right hand side of Fig. 6 is produced by this technique without any bounding of set PID. It describes all possible behaviours around states focused on a process life-cycle. The two crossing conditions between brackets represent respectively conditions D and A, for *enabledness* and *reachability*. Empty brackets mean that the condition is proved true. A cross X on a condition means that this condition has not been proved neither true, nor false.



6.a) SLTS produced by the graph abstraction tool for $PID = \{p_1, p_2\}$



6.b) SLTS Produced by the *GénéSyst* Tool for $PID = \{..., P_i, ...\}$

Fig. 6. Results of Under and Over-Approximation Techniques

Compared to the under-approximation diagram produced for two processes, some transitions exhibited by *GénéSyst* don't correspond to any transition of the concrete representation (*New*, *ready* and *swap* transitions, reflexive on state $P_i \in Ready$). Indeed, limited to two processes for this example the under-approximation technique didn't explore a sufficient number of states.

5 Human Oriented Empirical Study

Techniques discussed in the previous sections allow to produce behavioural views from a formal B specification depending on the abstraction chosen by the analyst. We have conducted experimentally a qualitative study with students from a master's degree specialized on software engineering, and who have finished a detailed course about the B method. We formed two groups of 17 students to which we provided two different specifications: the scheduler example discussed in this paper, and a B specification modelling access control mechanisms to buildings. We applied our tools to these specifications and produced various

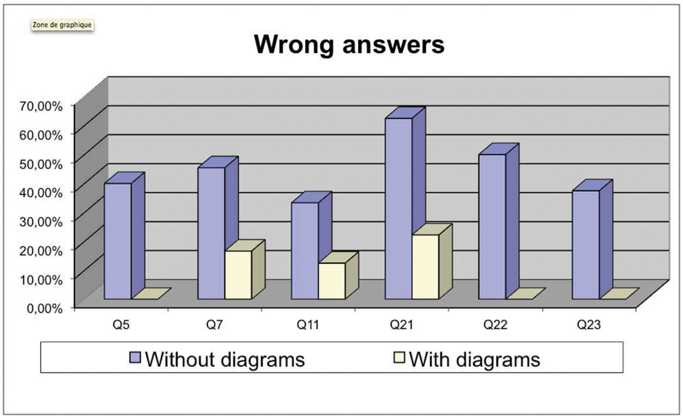


Fig. 7. Diagrams reduce significantly error rate for some questions

diagrams in order to graphically document them. Every group had two lists of questions about two different specifications where only one was supported by diagrams. Our intension was to evaluate the error rate variations of answers to quiet simple questions about these specifications when diagrams are provided. This study allowed to observe an error rate decrease from 26.14 % to 15.60 % when specifications are documented graphically. A variation about 11 % is interesting, because it hightlights the contribution of diagrams to the understanding of B specifications, but it may not seem very promising. We believe that the inverse would be surprising because specifications provided to students are not complex and should be accessible for them. Indeed, a global error rate near 26 % may be acceptable for persons who are not skilled with formal techniques, but 15 % is better. More specifically, we observed that diagrams reduce significantly the number of wrong answers for several questions. Figure 7 gives wrong answers proportions with and without diagrams and shows that the error rate can be divided by three and sometimes it is reduced from around 50 % to zero.

Questions G5-Q23 will be detailed further. 30 % of students to whom we didn't provide diagrams, misunderstood the equity property and considered that a process can't be bloqued indefinitely in the ready state (question Q11

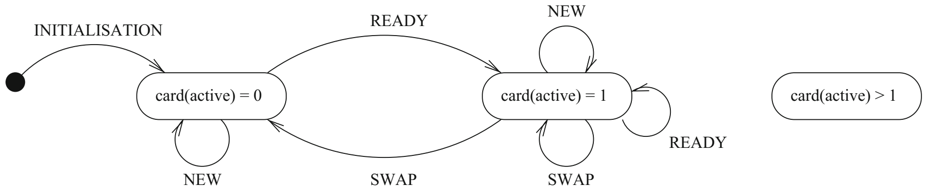


Fig. 8. State/Transition diagram focused on active processes

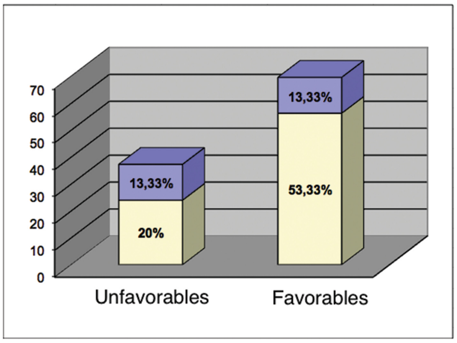


Fig. 9. Did the diagrams help you to understand specifications?

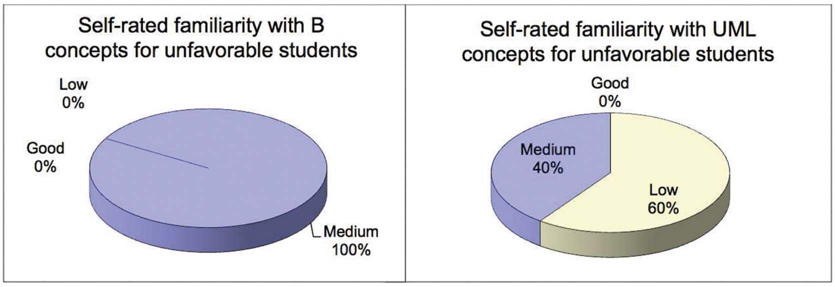


Fig. 10. Self-rated familiarity with B and UML for unfavourable students

in Fig. 7). However, when the diagram of Fig. 2 is provided, only 10 % gave the wrong answer. We believe that such a property is somehow difficult to perceive from a human point of view. Indeed, in order to be verifyied, the equity property needs more automated tool analysis or other formal languages, such as LTL, because it is a kind of behavioural properties not explicit in the B model. Nevertheless, invariant properties can be illustrated graphically using state transition diagrams. For example, Fig. 8, produced by our tools, shows that state $card(active) > 1$ is not reached by any transition and hence it is conformant to invariant $card(active) \leq 1$.

Without this diagram, about 40 % of students said that there may be several active processes at the same time (question Q5 in Fig. 7). Although invariant $card(active) \leq 1$ is clearly mentioned in the scheduler specification, students were not able to attest that the scheduler operations preserve such a trivial property. This result emphasizes the interest to document graphically an invariant property for a better human understanding. Indeed, we obtained 100 % of good answers when Fig. 8 is provided.

An overall appreciation of the graphical views is given in Fig. 9 and shows that two out of three students think that diagrams helped them to understand specifications and the remaining one third expresses an unfavourable opinion.

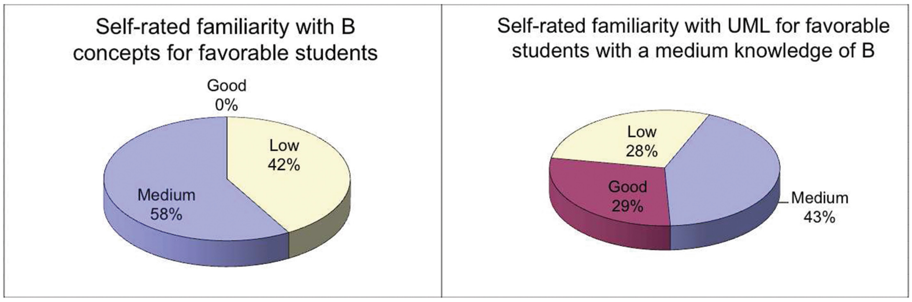


Fig. 11. Self-rated familiarity with B and UML for favourable students

In these two proportions, 13.33 % of students say that diagrams didn't help them at all and 13.33 % of them have the opposite opinion. In order to refine these results we asked students to evaluate their knowledge of B and UML notations (Figs. 10 and 11). A great part of students who disagree with the interest of diagrams seem to be uncomfortable with UML notations and has a better familiarity with the formal B notation. Basing on this self-rated familiarity with B and UML, one may conclude that although graphical views seem to be a way for making a formal specification more accessible, they can have the inverse effect because they also require some knowledge. This observation is confirmed by the proportion of students who appreciated diagrams and who has obviously a better mastering of graphical UML notations.

6 Conclusion

It is commonly known that formal specifications are complex because of notations that need a great mathematical background. In this paper, we focused our interest on a B specification which is based on a verbose notation, close to a programming language, and which should be more affordable than other formal notations. Our empirical study showed that the language itself is not the main reason to be less at ease with a formal method. Obviously, the difficulty for humans is to have an overall view on the formal model.

This paper has presented two complementary approaches providing a behavioural abstract view from a formal specification, in order to ease its understanding. Figure 6 shows an example of results issued from under and over-approximation techniques. We can observe from these diagrams that the *GénéSyst* tool associates guards to events in order to describe their *enabledness* and *reachability* properties. However, reflexive transitions SWAP, READY and NEW in state $p_i \in ready$ are not possible when the scheduler system deals with only two processes. For this particular set of processes the graph abstraction tool produced a more precise diagram. *GénéSyst* being based on proof techniques, it suffers the usual limitations of automatic provers: some theorems cannot be proved automatically and require user interaction. Furthermore, if the under-approximation approach can be used to verify reachability properties, then the over-approximation approach is mainly interesting in case of safety properties. Both techniques have some restrictions such as a limited state space for the first one and a too large abstraction in case of leak of proof for the second one.

We also measured the computational time of each approach and we noticed that the graph abstraction tool produced the state/transition diagram of a process life cycle in 7 s for $PID = \{p_1, p_2\}$ and 13 s for $PID = \{p_1, p_2, p_3\}$; while the *GénéSyst* tool produced this diagram in 80 s for an unbounded state space. This confirms that under-approximation tools are interesting when the state space can be reduced to a small finite space. Furthermore, if some interesting behaviours are not included in the concrete graph, they will not appear in the abstract diagram. Given sets, such as *PID*, can be turned into enumerated sets but numerical data structures such as *NAT* are less easy to address. Over-approximation tools are much more interesting for such complex data structures because they may be used to provide more formal evidence on the diagram transitions.

Over-approximation, can be dedicated to verify safety properties as proposed in [4] and [16]. It has the advantage to preserve infinite concrete state space without any constraint, and hence safety properties could be established on the symbolic transition system. The resulting LTS could also be used like a test oracle which brings some interesting perspectives [9].

References

1. Abrial, J.-R.: Extending B without changing it (for developing distributed systems). In: Habrias, H. (ed.) First Conference on the B method, France, pp. 169–190 (1996)
2. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
3. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: a challenging model transformation. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 436–450. Springer, Heidelberg (2007)
4. Bert, D., Potet, M.-L., Stouls, N.: GeneSyst: a tool to reason about behavioral aspects of B event specifications. Application to security properties. In: Treharne, H., King, S., C. Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 299–318. Springer, Heidelberg (2005)
5. Casset, L.: Development of an embedded verifier for java card byte code using formal methods. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 290–309. Springer, Heidelberg (2002)
6. Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications. In: Woodcock, J.C.P., Larsen, P.G. (eds.) FME '93: Industrial Strength, Formal Methods. LNCS, vol. 670, pp. 268–284. Springer, London (1993)
7. Essamé, D., Dollé, D.: B in large-scale projects: the Canarsie line CBTC experience. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 252–254. Springer, Heidelberg (2006)
8. Idani, A., Ledru, Y.: Dynamic graphical UML views from formal B specifications. Int. J. Inf. Softw.Technol. **48**(3), 154–169 (2006). Elsevier
9. Julliand, J., Stouls, N., Bué, P.-C., Masson, P.-A.: B model slicing and predicate abstraction to generate tests. Softw. Qual. J. **21**(1), 127–158 (2013)

10. Ledru, Y.: Using Jaza to animate RoZ specifications of UML class diagrams. In: SEW, pp. 253–262. IEEE Computer Society (2006)
11. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
12. Leuschel, M., Butler, M., Spermann, C., Turner, E.: Symmetry reduction for B by permutation flooding. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 79–93. Springer, Heidelberg (2007)
13. Pouzancre, G.: How to diagnose a modern car with a formal B model? In: Bert, D., Bowen, J.P., King, S., Waldén, M. (eds.) ZB 2003. LNCS, vol. 2651, pp. 98–100. Springer, Heidelberg (2003)
14. Savicks, V., Snook, C.: A framework for diagrammatic modelling extensions in Rodin. In: Rodin Workshop (2012)
15. Snook, C., Butler, M.: UML-B: formal modeling and design aided by UML. ACM Trans. Softw. Eng. Method. (TOSEM) **15**(1), 92–122 (2006)
16. Vu, D-H., Chiba, Y., Yatake, K., Aoki, T.: Model checking conformance of design model to its formal specification, Research report (2014)