Mapping Requirements to B models

Michael Jastram Michael Leuschel
Jens Bendisposto
Heinrich-Heine Universität Düsseldorf

Aryldo G Russo Jr
Research Institute of State of São Paulo (IPT)

May 25, 2009

Abstract

Formal methods in systems engineering are gaining traction, at least in some areas. While the formal specification process from abstraction via refinement to implementation is fairly well understood, the traceability between the initial user requirements and the formal model is still unsatisfying. There are some promising attempts (e.g. KAOS) that inspired some of the work done here.

Our objective is to find a practical way to establish traceability between natural language requirements and B models.

We select a number of existing methods and notations for bringing natural language requirements and B specifications together. Specifically, we use UML-B for building a data model; we use invariants (part of the B method) to model safety requirements; and we use temporal expressions (LTL) to model liveness requirements.

In this paper, we show a pragmatic way that may lead to a method for making traceability between natural language requirements and B models easier to understand, maintain and validate.

1 Overview

Formal methods like Z [22] and B [2, 19], are slowly but steadily gaining ground in several industrial application areas, like railways [14], medical [12] and many more. With these methods, an initial mathematical model is refined in multiple steps, until the final refinement contains enough detail for an implementation in software. With model checking or formal proofs it can be shown that the final refinement implements the initial model.

But where does the initial model come from? It is derived from the user requirements. All too often, the requirements are used to build the model without establishing explicit traceability, even though traceability has become established as a key subject of requirements engineering research [1].

In this paper, we are looking for a pragmatic approach to make traceability between natural language requirements (NLRs) and B models easier. We do this by looking for intermediate constructs that provide a robust interface between the two.

1.1 Structure of this Paper

We first set the stage in Section 2 by introducing the **case study**, a short requirements document for a traffic light, together with one possible solution in B. We also investigate possible mappings without any additional constructs and find a possible mapping between **safety requirements and invariants**. In Section 3, we explore the mapping between **liveness requirements and temporal expressions**, specifically using LTL. In Section 4 we look at the **event structure** and in Section 5 at various ways of building and tracing the **data model**. We identify UML-B as a useful tool for building and maintaining the data model. In Section 6, we look at the implications of **B refinement** on traceability. In Section 7, we conclude by looking at the feasibility of the approach introduced here. Specifically, we analyze whether this approach scales, is maintainable and can be verified in a systematic fashion. We close by discussing the next steps of our research.

1.2 Existing Research

The B method is a formal method based around Abstract Machine Notation (AMN). The original B (referred to as classical B, [19]) has a rich feature set that got consciously reduced in the younger Event-B [17] in order to make the notation easier to use and automated proving easier to implement. Some tools provide their own extensions to B. ProB [15], for instance, allows model checking against LTL expressions.

We use NLRs as the starting point for our experiments. While requirements can be stored in forms other than natural language, it is the most natural way for the customer to express their perception of the model [3]. NLRs are commonly classified as functional and non-functional [21], often also as environmental. This is how we structured the user requirements of our test model. In this paper we focus on functional requirements, as non-functional requirements are very hard to model in B. NLRs can also be processed (semi-)automatically [11], which we won't pursue in this paper.

Some ideas in this paper are related to KAOS [9], a method for requirements engineering that spans from high-level goals all the way to a formal model. KAOS requires the building of a data model in a UML-like notation, and it allows the association of individual requirements with formal realtime temporal expressions. But in contrast to KAOS, the method presented here starts with natural language requirements. While KAOS has a strong focus on analysis, our focus in this paper is modeling. A practical solution for traceability between KAOS requirements and B has been proposed by [18].

Recognizing the need for using temporal logic in Event-B models, [4] proposed a syntactic extension of Event-B incorporating a limited notion of obligations described by triggers.

In this paper, we generally map functional requirements to liveness and safety requirements [13]. In [5] it is shown how for finite systems, liveness properties can be transformed into safety properties for some systems, which may ease the process of verifying the model.

2 The Case Study

We use the specification for the controller of a traffic light system for the work presented here. The system consists of traffic lights for pedestrians and cars that allows the safe passage of pedestrians and cars (Figure 2.1).

The NLRs are written to resemble industrial requirements and adhere to some quality standards [10], but are not necessarily perfect. Table 2.1 shows the requirements. The prefix of the identifier ID indicates the type of the requirement: INF - Information; ENV - Environment; LIV - Liveness Requirement; SAF - Safety Requirement; NF - Non-Functional Requirement.

The B model in Table 1.1 realizes these requirements. For simplicity, we don't use refinement here, which is discussed in Section 6.

Table 1.1: A classical B model realizing a traffic light controller

```
MACHINE trafficlight

SETS
  colors = {green, yellow, red}

VARIABLES
  carsColor,
  pedsColor,
  buttonPressed,
  pedsWereGreen

INVARIANT
  carsColor  ⊆  colors  ∧
  pedsColor  ⊆  colors  ∧
  pedsWereGreen  ∈  BOOL  ∧
  buttonPressed  ∈  BOOL  ∧
  carsColor  ≠  { red }
      ⟹  pedsColor = { red }  ∧
  pedsColor  ≠  { red }
      ⟹  carsColor = { red }

INITIALISATION
  carsColor := { red } ||
  pedsColor := { red } ||
  buttonPressed := FALSE ||
  pedsWereGreen := FALSE

OPERATIONS
  pressButton =
  PRE
    pedsColor = { red }  ∧
    buttonPressed = FALSE
  THEN
    buttonPressed := TRUE
  END;

  carsRedToRedYellow =
  PRE
    pedsColor = { red }  ∧
    carsColor = { red }  ∧
    pedsWereGreen = TRUE
  THEN
    carsColor := {red,yellow} ||
    pedsWereGreen := FALSE
  END;

  carsRedYellowToGreen =
  PRE
    pedsColor = { red }  ∧
    carsColor = { red, yellow }
  THEN
    carsColor := { green }
  END;

  carsGreenToYellow =
  PRE
    buttonPressed = TRUE  ∧
    pedsColor = { red }  ∧
    carsColor = { green }
  THEN
    carsColor := { yellow }
  END;

  carsYellowToRed =
  PRE
    carsColor = { yellow }
  THEN
    carsColor := { red }
  END;

  pedsStopToGo =
  PRE
    pedsColor = { red }  ∧
    carsColor = { red }  ∧
    pedsWereGreen = FALSE
  THEN
    pedsColor := { green }
  END;

  pedsGoToStop =
  PRE
    pedsColor = { green }
  THEN
    pedsColor := { red } ||
    pedsWereGreen := TRUE ||
    buttonPressed := FALSE
  END
END
```

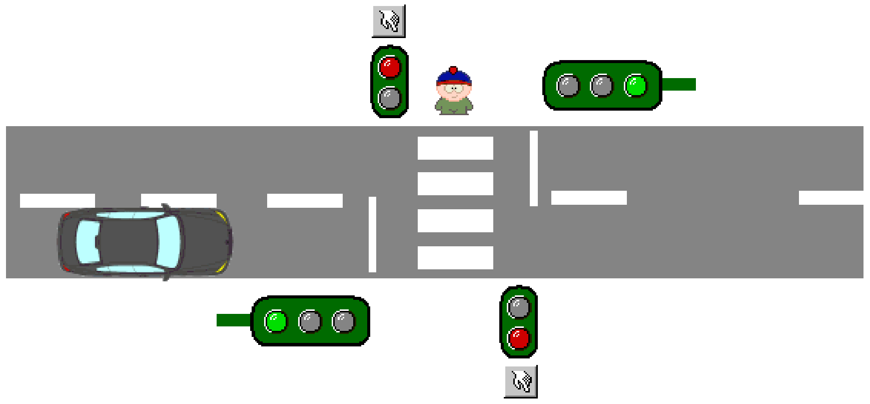

Figure 2.1: A traffic light for pedestrians

Table 2.1: Requirements for a traffic light

ID	Natural Language Requirement
INF-1	This Specification describes the controller of a traffic light that allows pedestrians to cross a busy street.
ENV-6	The System is controlling cars on a road and pedestrians crossing the road.
ENV-1	The System is equipped with two traffic lights for the cars, with the colors red, yellow and green.
ENV-2	The System is equipped with two traffic lights for the pedestrians with the colors red and green.
ENV-3	The lights for the cars stop the cars on both sides of a crosswalk.
ENV-4	The lights for the pedestrians stop the people on both sides of the crosswalk.
ENV-5	Underneath the lights for the pedestrians, two call buttons are mounted (one on each side of the street).
SAF-2	The traffic lights for the cars are in sync (i.e. can be treated as one).
SAF-3	The traffic lights for the pedestrians are in sync (i.e. can be treated as one).
SAF-4	The lights for pedestrians and cars must never be "go" at same time.
SAF-5	"go" means green for pedestrians and both green and yellow for cars.
INF-2	The traffic light sequence is behaves like a German traffic light.
LIV-6	The traffic light for the cars always follows the sequence: green - yellow - red - red/yellow.
LIV-7	The traffic light for the pedestrians always follows the sequence red - green.
LIV-8	Usually, the light for the cars is green.
LIV-9	Upon pressing the call buttons, the car light sequence is initiated until the light is red.

ID	Natural Language Requirement (cont.)
LIV-10	Once the car light is red, the pedestrian light sequence is initiated until the light is green.
LIV-11	After some time (to be defined), the pedestrian light sequence is continued until the light is red.
LIV-12	Once the pedestrian light is red, the car light sequence is continued until the light is green.
NF-13	All time intervals of the sequences can be configured.
NF-14	The time interval for the green state of the car light sequence has a minimum value (see LIV-8).
SAF-15	A pedestrian may push the call button multiple times (before their light has turned green) without initiating the sequence multiple times.
SAF-16	The call button only initiates the sequence if the pedestrian light is red.
LIV-17	Upon turning from green to red of the pedestrian light, the call button status is reset.

2.1 *Mapping Requirements*

Our objective is to establish traceability between the requirements (Table 2.1) and the B model (Table 1.1). This can be done fairly easily for some functional requirements, especially the safety requirements. For instance:

SAF-4	The lights for pedestrians and cars must never be "go" at same time.

This requirement is reflected by the following two conjuncts of the invariant[1] (lines 17-20):

$$\begin{aligned} carsColor \neq \{red\} &\implies pedsColor = \{red\} \wedge \\ pedsColor \neq \{red\} &\implies carsColor = \{red\} \end{aligned}$$

We have a clear and robust traceability between a requirement and B: Robust in the sense that the invariant can be verified and is indifferent to the event structure of the machine.

[1]This is not quite true, we implicitly used SAF-5 to interpret "go" in terms of actual traffic light colors. The proper approach would be to work with refinement, where the invariant is stated in terms of "stop" and "go" in the initial model, which is mapped with a glueing invariant to the actual colors in a later refinement. We will do exactly that in Section 6. To keep this example simple, we omitted that step here.

Unfortunately, not all requirements can be traced that easily. Consider the following:

LIV-6	The traffic light for the cars always follows the sequence: green - yellow - red - red/yellow

To show that this requirement is fulfilled, we have to demonstrate for every event that it only modifies the state variable carsColor according to the given sequence. For instance, if the color of the car light is green, then events modifying the color may only be enabled if the color stays green or is set to yellow.

This requirement is fulfilled by this machine. This can be verified by inspecting ever single event in the machine.

This approach is cumbersome, as with every addition of a new event, this check has to be performed. Also, B makes it difficult to capture such a requirement in a proveable way (e.g. with an invariant). And last, there is no group of elements in the B model that we can link to for tracing requirement LIV-6: While we could link to the guards and actions (line numbers 23, 40, 43, 49, 51, 58, 60, 65 and 67), this leaves out an important aspect: We don't capture the fact that three events don't modify the value of carsColor in the first place. This information is crucial for fulfilling LIV-6 (and all other LIV requirements as well, but can't be linked to easily.

Now that we have shown that it is awkward to establish a clear traceability for all functional requirements, not even with a system as simple as a traffic light, we will look at ways for alleviating this. While at least some safety requirements can be formulated using invariants, there are no corresponding constructs in B for liveness requirements.

3 Temporal Logic

As we have seen in the previous Section, some requirements are hard to trace to elements of the B model. In this Section, we attempt to formulate these requirements as temporal logic expressions. We use LTL (Linear Temporal Logic), because tool support is available: ProB [16] can evaluate LTL expressions for classical B machines, and at the time of publication, support for EventB should be available through the ProB-Plugin for Rodin. While there are more powerful temporal logics available (e.g. CTL* [7]), so far LTL has been sufficient (and is both more understandable and useful in practice than the branching time logics [23]).

LTL consist of path formulas with the temporal operators X (next), F (future), G (global), U (until) and R (release). Expressions between curly braces are B predicates which can refer to the variables of the B specification.

Let's try to express some of the liveness requirements from Table 2.1 using LTL. We'll start with LIV-6 that we already discussed in the previous Section.

LIV-6	The traffic light for the cars always follows the sequence: green - yellow - red - red/yellow

This requirement can be expressed with the following LTL expression:

$$\begin{aligned}
&G(\{lightsCars = \{green\}\} \\
&\qquad\qquad \implies (\{lightsCars = \{green\}\}\ U\{lightsCars = \{yellow\}\})) \wedge \\
&G(\{lightsCars = \{yellow\}\} \\
&\qquad\qquad \implies (\{lightsCars = \{yellow\}\}\ U\{lightsCars = \{red\}\})) \wedge \\
&G(\{lightsCars = \{red\}\} \\
&\qquad\qquad \implies (\{lightsCars = \{red\}\}\ U\{lightsCars = \{red, yellow\}\})) \wedge \\
&G(\{lightsCars = \{red, yellow\}\} \\
&\qquad\qquad \implies (\{lightsCars = \{red, yellow\}\}\ U\{lightsCars = \{green\}\}))
\end{aligned}$$

For example, the first two lines say that globally (G) if the lights are green in some state of the system, then ($\implies$) it must stay green until (U) they eventually change to yellow.

In contrast to our earlier attempt, this expression represents our requirement in a robust way. It is completely independent from the transition events, and only makes a statement regarding the state of the system - not on how we get into the state in the first place.

Of course, there is still the question on how to validate a statement like this. In this particular case, we could validate it automatically with the ProB model checker.

Let's look at some of the other liveness requirements as well:

LIV-9	Upon pressing the call buttons, the car light sequence is initiated until the light is red.

This requirement can be formulated in LTL as follows:

$$G(\{lightRequested = TRUE\} \implies F(\{lightsCars = \{red\}\}))$$

Attention has to be paid to fairness (or some other mechanism, like probability). For instance, assume that we omit the precondition in line 32 (buttonPressed = FALSE) from the event pressButton - not an unreasonable assumption. Then the event could be triggered as often as one wishes. Without fairness, a model checker would find a violation of this LTL expression, as the pressButton event may get triggered infinitely often, without requiring the event for switching the light to be triggered at all. However, the requirements don't mention fairness, it's an unwritten (and quite reasonable) assumption. We don't see this as problematic, however: We still have an expression that captures the requirement. We simply have to add fairness constraints to our model in order to satisfy the requirement and the LTL expression.

Fairness is not the only possible issue. Consider the following requirement:

LIV-8	Usually, the light for the cars is green.

This requirement could be formulated in LTL as follows:

$$G(\{carsColor \neq \{green\}\} \implies F(\{carsColor = \{green\}\}))$$

Unfortunately, this doesn't capture the essence of the requirement, just one aspect of it. There are various ways for dealing with this. A pragmatic solution could be to separate the two aspects and to express the "usually" as a non-functional requirement, e.g.:

LIV-8a	When red, the Light for the cars will eventually turn green.
FUN-8b	Typically (when left alone), the traffic light for the cars is green.

Note that the second requirement is marked "FUN", indicating that it is a functional requirement, but not with an associated formal expression (as "SAF" and "LIV" requirements would). It would have to be validated separately.

A more formal solution could work with probabilities to express this requirement.

Yet another problem can be found with LIV-11:

LIV-11	After some time (to be defined), the pedestrian light sequence is continued until the light is red.

We can express the second part of this requirement using LTL:

$$G(\{pedsGo = TRUE\} \implies F(\{pedsGo = FALSE\}))$$

However the first part “after some time (to be defined)” is a real-time property of the system that is difficult to express with LTL. Here we would recommend to break this requirement into two and to state the real-time property as a non-functional requirement, similar to NF-14 in the requirements. Alternatively, it would be possible to work with counters or abstract time or timed LTL.

Statements about State or Events?

We just identified a practical way to capture liveness requirements by creating LTL formulas that make a statement in regard to the system’s state, regardless of the events that are used to get to that state. While this works in this case, it is by no means the only way to formulate the requirements. There is a ProB specific extension to LTL that allows us to make statements about the events as well. In LIV-9, for instance, the requirement states “Upon pressing the call buttons...”. When we rephrased that requirement in LTL, we made a statement regarding the variable lightRequested. Instead, we could have made a statement in regard to the triggering event, pressButton (Listing 1.1, line 29):

$$[pressButton] \implies F(\{carsColor = \{red\}\})$$

Note that this is a ProB-specific extension of LTL, not part of standard LTL.

4 The Event Structure

A B model changes its state by triggering events. Thus, in order to implement the behavior of the B model, we need to implement events that perform state transitions. To ensure that the invariants of the system are not being violated, events have preconditions (also called guards) that must be true for the event to be triggered. Thus, the events determine the temporal behavior of the system and determine whether the LTL expressions can be validated.

There are many possibilities for structuring the events for our model. We used a common approach and simply identified the verbs in the requirements. The result is shown in Table 5.1. Note that here we already chose names for the events that could be used directly in the B model. Also note that we built a hierarchy of verbs. We can use this to define our refinement strategy (see Figure 5.1): in the initial abstraction, we have the “carGo” and “carStop” events, which are later refined into the color transition events.

Table 5.1: Data Structures and Actions, discovered by analyzing the requirements

Data Structures	**Actions**
Traffic Lights	**carLightSequence**
• LightsCars	• carStop
• LightsPedestrians	– carYellowToRed
Colors	• carGo
• Go	– carRedToRedYellow
• Red	– carRedYellowToGreen
• Yellow	– carGreenToYellow
• Green	**pedestrianLightSequence**
CallButtons	• pedStop
	– pedGreenToRed
	• pedGo
	– pedRedToGreen
	pressCallButton

Also note that events may appear in multiple requirements. The event “pressCall-Button”, for instance, appears in LIV-9 (“pressing the call buttons”) and SAF-15 (“push the call button”). Thus, in the process of building the event structure, we can either build an event glossary, or we could clarify the requirements by picking a consistent terminology.

While there may be more sophisticated ways for structuring events, it is beyond the scope of this paper.

5 The Datamodel

Mapping functional requirements to B constructs is our core objective. But in order to make that possible, we need a data model as well. We manually built the data model for the B model (Table 1.1); in this Section we will look for better ways for building it, and at ways for establishing traceability both to the requirements and to the B model.

The data model is defined in the environmental requirements, while it is used by

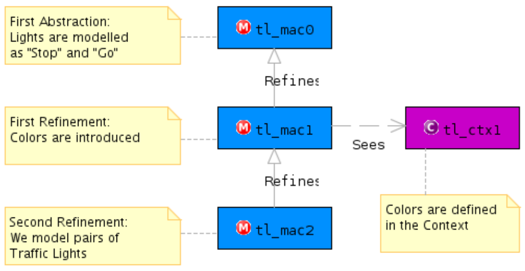

Figure 5.1: The high-level structure of the traffic light in UML-B

the functional and non-functional requirements.

We manually processed all requirements by identifying all nouns that are part of the data structures, similar to our approach structuring the events. This is a common approach that allows us to build a comprehensive glossary and to identify inconsistencies in the terminology at the same time. A good tool would support traceability between the glossary and the requirements on one hand, and traceability between the glossary and the relevant elements in the B model.

We see that for every data element, we have a clear counterpart in the B model in the form of sets (e.g. colors), constants (e.g. red, yellow, green) and variables (e.g. LightsCars).

There are some methods that were proposed to help gather this information from the requirements. Abstfinder [11] for instance is based on traditional signal processing methods to help finding abstraction in natural language texts. NaLER [6] (Natural language for E-R) is an approach where Entity-Relationship is used to documenting the data requirements of information systems. As the data acquisition is not in the main focus of this work, we won't pursue these approaches further at this point.

5.1 *UML-B*

Our data model is fairly simple, but for more complex systems it would be useful to have a more powerful tool for building the data model. A tool designed for mapping data structures onto B is UML-B [20], which provides mapping between UML-like constructs and Event-B. Tool support for Rodin [8] is available. UML-B is attractive, as it presents the data model in a form that is easier to understand by the stakeholders then the B constructs, as UML is well established, even with non-technical people.

In UML-B, a package diagram shows the high-level structure of the model (Figure 5.1). Each Box can be "opened" to show more details. For this paper, we constructed one

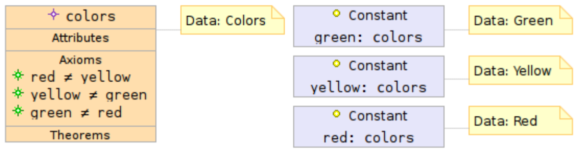

Figure 5.2: Context (introduced with first refinement)

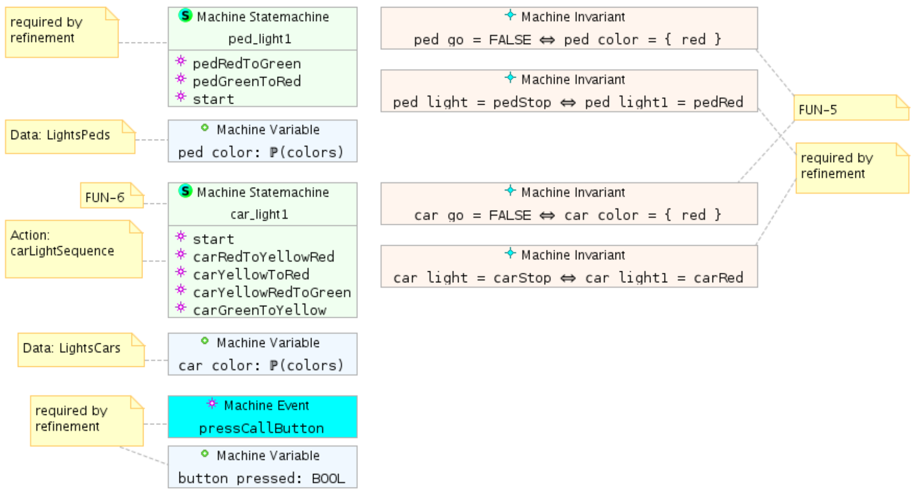

Figure 5.3: First refinement

initial model, one context and two refinements (for brevity, we only print the context and the first refinement).

In Event-B (the dialect supported by UML-B) the model is divided into context and machine (in classical B, everything is part of the machine). The context contains mainly constants and sets, while the machine contains mainly variables and events. Figure 5.2 shows the context, modelled in UML-B, of our traffic light.

The individual colors are modelled as constants and the available colors as a set. While this is not much different from what is captured in the B notation, it may be easier for the analyst to represent more complex data structures. Likewise, we can represent variables in the machine notation of UML-B, as shown in Figure 5.3 (the first refinement). The variables are shown in the three boxes labeled “Machine Variable” and are called “ped_color”, “car_color” and ”button_pressed“.

The machine shown in Figure 5.3 models not only the data structures, but the complete machine, including events and invariants. Of particular interest is the ability of UML-B to represent statemachines. The box labeled ”car_light1“ contains a statemachine that can be edited as shown in Figure 5.4. Thus, we have some limited possibilities for modelling temporal requirements (the statemachine car_light1 models requirement LIV-6).

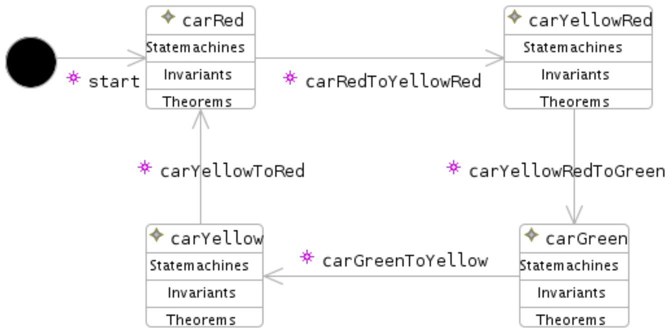

Figure 5.4: Requirement LIV-6 as a UML-B state machine

Unfortunately, UML-B is inadequate to model much of the other liveness requirements, because it doesn't support temporal logic. The statemachine shown here is implemented by new constructs, and not by a temporal logic.

UML-B can represent invariants, which are simply shown in their respective boxes. But as the content of the boxes doesn't differ from the invariants written in the B model, we don't gain anything by using those UML-B features over writing plain B.

Ultimately, UML-B is an attractive bridge between requirements and B: On one hand, it is understandable to stakeholders and allows them to model the data structures in an intuitive, graphical way; on the other hand, it is a precise representation of the B data structures, with no ambiguities.

6 Refinement

In order to handle large systems, the B method supports the concept of refinement. With refinement, only some aspects of the system are modelled in a first, step, while subsequent refinements add more and more detail. The various refinements are held together by refinement rules and glueing invariants.

In addition to the model shown in Table 1.1, we created a model that uses refinement in Event-B. Figure 5.1 shows the refinement structure using the UML-B diagram notation. Table 6.1 shows how the event carGoToStop got refined into the event carYellowToRed.

The initial model and refinement are held together by the following glueing invariant:

$$car\_go = FALSE \Leftrightarrow car\_color = \{red\}$$

Table 6.1: Refining event carGoToStop into carYellowToRed

Initial Model	**First Refinement**
`carGoToStop =`	`carYellowToRed =`
	`REFINES`
	`  carGoToStop`
`WHEN`	`WHEN`
`  car_go = TRUE  ∧`	`  car_color = { yellow }  ∧`
`  button_pressed = TRUE`	`  button_pressed = TRUE`
`THEN`	`THEN`
`  car_go := FALSE`	`  car_color := { red }`
`END`	`END`

Refinement has different implications for safety and liveness requirements, or invariants and LTL expressions, respectively. Invariants hold across subsequent refinements, which can be proven. This is not necessarily true for LTL formulas. To be more precise: It is true for LTL formulas, provided the last system is not deadlocking *and* the formula does not reason about enabledness of operations.

7 Conclusion

In this paper, we identified a number of constructs for bringing requirements and Event-B models together. We discovered UML-B as an intuitive tool for building the data model from the data structures extracted from the environmental requirements. We identified invariants as a suitable tool for describing safety requirements. And we found that LTL expressions are suitable for describing liveness requirements.

When we put the discoveries in this paper together, we find a pragmatic approach for building B models from requirements:

1. **Start with quality requirements** – we assume that we have requirements of sufficient quality to work with.
2. **Categorize the requirements** into environmental, liveness and safety requirements. There will be other categories (non-functional, information, etc.) that must be accounted for elsewhere.
3. **Built the data structures**, using the environmental requirements. This could be done in UML-B, manually or otherwise.

4. **Formulate the safety requirements**, as invariants.
5. **Formulate the liveness requirements**, as LTL expressions.
6. **Build the B events**, starting with the actions derived from the requirements. The invariants and LTL statements will drive the creation of guards, actions, etc. to satisfy the constraints of the system.
7. **Iteratively improve the model**. With good tool support, we have a clear interface between the non-formal requirements and the formal model. Along this interface we can analyze the system, deal with changes, etc.

There are still a number of unanswered questions here. Specifically, it is not clear how to best use the LTL expressions. Rodin can proof a system to be correct in respect to invariants, but not in respect to LTL expressions. Currently, model checking is the only option, which is tricky even with moderately sized problems. We did verify the trafficlight system with the ProB modelchecker.

On the other hand, we believe that this approach allows for a systematic (albeit manual) verification of the model. We can systematically iterate over the requirements to make sure that they are all modelled. It's trickier the other way around to inspect every B element, as there will be a number of B elements that are required for enforcing the system constraints, but not necessarily the user requirements (glueing invariants are an example for this).

7.1 Next Steps

While the approach described here is promising, the case study of the traffic light is not representative. Next we will build models from industrial requirements, as provided by the partners of the Deploy project. If the results are positive, we will build a prototype for the Rodin platform.

Further, we need to investigate how we can incorporate LTL expressions into the Rodin tool platform. This is currently being done by using the existing ProB-Plug-In, but is an unsatisfying approach, as the Rodin platform has a strong focus on proving correctness. Thus, we will look further into ways for converting LTL expressions into provable invariants.

Acknowledgements

The work in this paper is partly funded by Deploy [24]. Deploy is an European Commission Information and Communication Technologies FP7 project.

Bibliography

[1] J Abrial. Formal methods: Theory becoming practice. *Journal of Universal Computer Science*, Jan 2007.

[2] J.-R. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.

[3] Vincenzo Ambriola and Vincenzo Gervasi. Processing natural language requirements. In *In Proceedings of ASE 1997*, pages 36–45. IEEE Press, 1997.

[4] Juan Bicarregui, Alvaro Arenas, Benjamin Aziz, Philippe Massonet, and Christophe Ponsard. Towards modelling obligations in Event-B. In Egon Börger, Michael Butler, Jonathan P. Bowen, and Paul Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, pages 181–194. Springer, 2008.

[5] Armin Biere and Cyrille Artho. Liveness checking as safety checking. In *In FMICS'02: Formal Methods for Industrial Critical Systems, volume 66(2) of ENTCS*, 2002.

[6] Atkins C. Naler: a natural language method for interpreting entity-relationship models. *Campus-Wide Information Systems*, 17:85–93(9), 1 March 2000.

[7] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, 1999.

[8] Joey Coleman, Cliff Jones, Ian Oliver, Alexander Romanovsky, and Elena Troubitsyna. RODIN (rigorous open development environment for complex systems). In *EDCC-5, Budapest, Supplementary Volume*, pages 23–26, April 2005.

[9] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. GRAIL/KAOS: an environment for goal-driven requirements engineering. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 612–613, New York, NY, USA, 1997. ACM.

[10] Alan M. Davis. *Software requirements: objects, functions, and states.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[11] L. Goldin and D. Berry. Abstfinder, a prototype natural language text abstraction finder for use in requirements elicitation, 1997.

[12] R Jetley, S Iyer, and P Jones. A formal methods approach to medical device review. *COMPUTER*, Jan 2006.

[13] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[14] T Lecomte, T Servat, and G Pouzancre. Formal methods in safety-critical railway systems. *Proc. Brazilian Symposium on Formal Methods: SMBF*, Jan 2007.

[15] M. Leuschel and M. Butler. ProB: a model checker for B. pages 855–874, 2003.

[16] Michael Leuschel and Daniel Plagge. Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. In Yamine Aït Ameur, Frédéric Boniol, and Virginie Wiels, editors, *Proceedings Isola 2007*, volume RNTI-SM-1 of *Revue des Nouvelles Technologies de l'Information*, pages 73–84. Cépaduès-Éditions, 2007.

[17] C. Métayer, J.-R. Abrial, and L. Voisin. Event-B language. Deliverable D7, EU-IST "RODIN" Project, 2005.

[18] Christophe Ponsard and Emmanuel Dieul. From requirements models to formal specifications in B. In Régine Laleau and Michel Lemoine, editors, *ReMo2V*, volume 241 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[19] Steve Schneider. *The B-Method: An introduction.* Palgrave Macmillan, 2001.

[20] C. Snook and M. Butler. U2B - a tool for translating UML-B models into B. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems Design*, chapter 5. Springer, 2004.

[21] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A Good Practice Guide.* John Wiley & Sons, Inc., New York, NY, USA, 1997.

[22] J.M. Spivey. *Understanding Z.* Cambridge University Press, 1988.

[23] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In Tiziana Margaria and Wang Yi, editors, *TACAS'01*, LNCS 2031, pages 1–22. Springer, 2001.

[24] WWW. http://www.deploy-project.eu/, 2008.