An Approach of Requirements Tracing in Formal Refinement

Accepted for VSTTE 2010

Michael Jastram[1], Stefan Hallerstede[1],
Michael Leuschel[1], and Aryldo G Russo Jr[2]

[1] Heinrich-Heine Universität Düsseldorf
{jastram, halstefa, leuschel}@cs.uni-duesseldorf.de
[2] Research Institute of State of São Paulo (IPT)
agrj@aes.com.br

Abstract. Formal modeling of computing systems yields models that are intended to be correct with respect to the requirements that have been formalized. The complexity of typical computing systems can be addressed by formal refinement introducing all the necessary details piecemeal. We report on preliminary results that we have obtained for tracing informal natural-language requirements into formal models across refinement levels. The approach uses the WRSPM reference model for requirements modeling, and Event-B for formal modeling and formal refinement. The combined use of WRSPM and Event-B is facilitated by the rudimentary refinement notion of WRSPM, which provides the foundation for tracing requirements to formal refinements.
We assume that requirements are evolving, meaning that we have to cope with frequent changes of the requirements model and the formal model. Our approach is capable of dealing with frequent changes, making use of corresponding techniques already built into the Event-B method.

Key words: Requirements Traceability, WRSPM, Formal Modeling, Refinement, Event-B

1 Introduction

We describe an approach for building a formal model from natural language requirements. Our aim is to increase the confidence that the formal model represents the desired system, by explaining how the requirements are "realized" in the formal model. The relationship "realizes" between requirements and formal models is kept informal. Justifications are maintained with each requirement and element of a formal model that are linked by "realizes", tracing requirements into the model, providing the sought explanation. Hence, the technical problem we have to solve is how to trace requirements into a formal model.

Requirements traceability provides a justification for a formal model with respect to the requirements. It is a difficult problem [6, 10, 15]. Furthermore, it is a cross-disciplinary problem connecting requirements engineering and formal

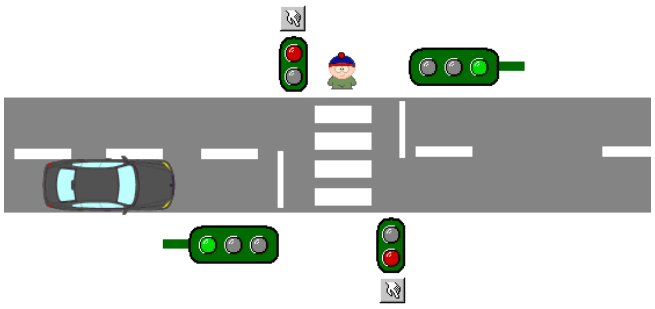

Fig. 1. A traffic light for pedestrians

methods. The benefits of the use of formal methods during requirements engineering has long been recognized. For instance, [5] quantifies the impact of formal methods in requirements engineering based on industrial case studies.

We assume that the requirements and the formal model need to be changed frequently and assume that the requirements are incorporated incrementally into the model. In the process, the requirements may have to be rewritten, corrected, clarified or split. The formal model may have to be modified correspondingly as the requirements become better understood [12].

In this paper, we present an approach for establishing robust traceability between informal requirements and formal models. We focus on natural language requirements and the Event-B formal method [3], but the ideas presented should be applicable more generally. We identified the WRSPM reference model [11] as the foundation for this work.

We consciously limit the scope of our approach. We assume that we start with a set of "reasonable" user requirements, but do not provide a method for eliciting them because good elicitation methods exist [14, 9].

1.1 Running Example

In Section 3, we use a traffic light controller, as depicted in Figure 1, to demonstrate our approach. The traffic light for the cars stays green until a pedestrian requests crossing the street by pressing a button. The requirements also describe the sequence of lights and other details. In our preliminary study, we applied our approach to two other examples, a lift controller and a system that controls the access of people to locations in a building. Moreover, the approach is being used in an industrial case of a train door control system, employing the B-Method [1] rather than Event-B.

We show excerpts of the formal model and requirements in boxes, as follows:

Short description
`Excerpt of formal model`

REQ-1	A textual requirement with the identifier REQ-1

1.2 State-Based Modeling and Refinement

We demonstrate our ideas using Event-B [3], a formalism and method for discrete systems modeling. Event-B is a state-based modeling method. The choice of Event-B over similar methods [7, 16] is mostly motivated by the built-in formal refinement support and the availability of a tool [4] for experimentation with our approach.

Event-B models are characterized by *proof obligations*. Proof obligations serve to verify properties of the model. To a large degree, such properties originate in requirements that the model is intended to realize. Eventually, we expect that by verifying the formal model we have also established that the requirements to which they correspond are satisfied.

We only provide a brief summary of Event-B in terms of proof obligations. A complete description can be found in [3]. Variables v define the state of a machine. They are constrained by invariants $I(v)$. Possible state changes are described by means of events. Each event is composed of a *guard* $G(t, v)$ and an *action* $\boldsymbol{S}(t, v, v')$, where t are *parameters* of the event. Actions are usually written in the form $v := E(v)$ corresponding to the predicate $v' = E(v)$. The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. In Event-B two main properties are proved about formal models: consistency, that is, the invariant $I(v)$ is maintained

$$I(v) \wedge G(t, v) \wedge \boldsymbol{S}(t, v, v') \Rightarrow I(v') \ ,$$

and refinement. Refinement links abstract events to concrete events aiming at the preservation of properties of the abstract event when it is replaced by the concrete event. A concrete event with guard $H(u, w)$ and action $\boldsymbol{T}(u, w, w')$ refines an abstract event with guard $G(t, v)$ and action $\boldsymbol{S}(t, v, v')$ if, whenever the gluing invariant $J(v, w)$ is true:

(i) the guard of of the concrete event is stronger than the guard of abstract event, and

(ii) for every possible execution of concrete event there is a corresponding execution of abstract event which simulates the concrete event such that the gluing invariant remains true after execution of both events.

Formally,

$$I(v) \wedge J(v, w) \wedge H(u, w) \wedge \boldsymbol{T}(u, w, w') \Rightarrow \exists t, v' \cdot G(t, v) \wedge \boldsymbol{S}(t, v, v') \wedge J(v', w') \ .$$

The Event-B method derives proof obligations from these two properties that are easier to handle and can be efficiently generated by a tool [4].

1.3 WRSPM

Our approach is based on WRSPM by Gunter et. al. [11]. WRSPM is a reference model for applying formal methods to the development of user requirements and their reduction to a behavioral system specification.

WRSPM distinguishes between artifacts and phenomena (see Figure 2). Phenomena describe the state space (and state transitions) of the domain and system, while artifacts represent constraints on the state space and the state transitions. The artifacts are broadly classified into groups that pertain mostly to the system versus those that pertain mostly to the environment. These are:

Domain Knowledge (W for World) describes how the world is expected to behave.
Proper Requirements (R) describe how we would like the world to behave.
Specifications (S) bridge the world and the system.
Program (P) provides an implementation of S.
Programming Platform (M for Machine) provides an execution environment for P.

In this paper, we use "proper requirements" for the formal artifacts R according the WRSPM terminology. We use just "requirements" when we talk about natural language from the stakeholders. Even though they are called requirements in practice, they may also contain information about the domain, implementation details, general notes, and all kinds of related information. We call those requirements REQ.

Artifacts are descriptions that can be written in various languages. In this paper, we use Event-B. (We discuss some alternatives in Section 4.1.)

We distinguish phenomena by whether they are controlled by the system (belonging to set s) or the environment (belonging to set e). They are disjoint ($s \cap e = \varnothing$), while taken together, they represent all phenomena in the system ($s \cup e =$ "all phenomena"). Furthermore, we distinguish them by visibility. Environmental phenomena may be visible to the system (belonging to e_v) or hidden from it (belonging to e_h). Correspondingly, system phenomena belonging to s_v are visible to the environment, while those belonging to s_h are hidden from it. Those phenomena are disjoint as well ($e_h \cup e_v = e$, $e_h \cap e_v = \varnothing$, $s_h \cup s_v = s$,

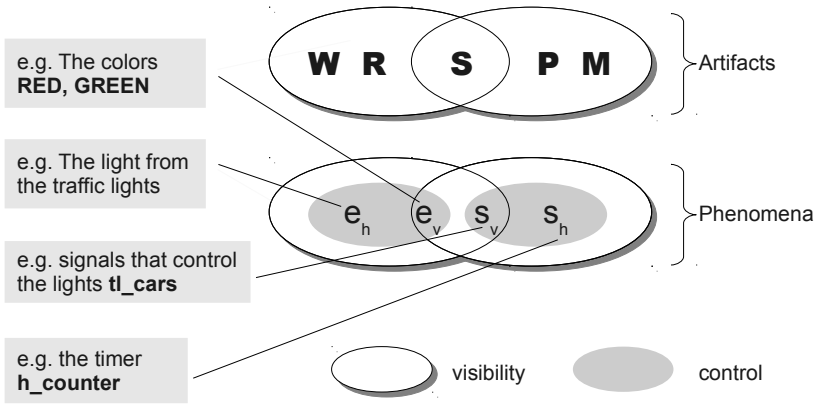

Fig. 2. WRSPM Artifacts and Phenomena, including Examples

$s_h \cap s_v = \varnothing$). Figure 2 illustrates the relationship between artifacts and phenomena, including a few examples for phenomena from the running example.

The distinction between environment and system is an important one; omitting it can lead to misunderstandings during the development. It is sometimes regarded as a matter of taste or convenience where the boundary between environment and system lies, but it has a profound effect on the problem analysis. It clarifies responsibilities and interfaces between the system and the world and between subsystems. If we require ourselves to explicitly make that distinction, we can avoid many problems at an early stage.

In larger projects, where the system is composed of other sub-systems, this concept can be used to determine if all the requirements are covered somewhere in the overall system: Some system phenomena of one sub-system may become the environment phenomena of the other sub-system.

W and R may only be expressed using phenomena that are visible in the environment, which is $e \cup s_v$. Likewise, P and M may only be expressed using phenomena that are visible to the system, which is $s \cup e_v$. S has to be expressed using phenomena that are visible to both the system and the environment, which is $e_v \cup s_v$.

Once a system is modeled following WRSPM, a number of properties can be verified with regard to the model, the first one being *adequacy with respect to* S:

$$\forall e\ s \cdot W \wedge S \implies R \tag{1}$$

Given both hidden and visible environmental (e) and system (s) phenomena, the system specification (S), under the assumption of the "surrounding" world (W), is strong enough to establish the proper requirements (R). The specification is implemented as the program P in the programming environment M, which allows us to rewrite (1) as

$$\forall e\ s \cdot W \wedge M \wedge P \implies R \tag{2}$$

2 Combining WRSPM and Event-B for Requirements Tracing

Our goal is to establish requirements traceability from natural language requirements to an Event-B formal model, using WRSPM to provide structure to both the requirements and the model. In the following, we first show how an Event-B model can be structured according to WRSPM, and then how this structure extends to the natural language requirements to support traceability.

2.1 Relationship between WRSPM and Event-B

As we demonstrate our method with Event-B, we need a relation between WRSPM and Event-B, shown in Table 1. An attempt to create a relation between Problem Frames and Event-B [18] provided similar results, thereby confirming our results. Event-B has a number of features that are useful for traceability:

Table 1. Representation of WRSPM elements in Event-B

WRSPM	**Event-B**
e and s	Phenomena are typically modeled as **constants**, **sets** or **variables**. They are associated with type information (**invariants** for variables, **axioms** for constants and sets). They are associated with **one or two events** that can modify it: Two events are required if both system and environment can modify the phenomenon, otherwise one is sufficient.
e_h	Phenomena hidden from the system are typically not modeled in the formal model. Exceptions are possible (for instance for fault analysis).
W	Domain properties are typically modeled as **invariants** and **axioms**. The line between type information and domain property may be blurry. Domain properties are typically expressed in terms of e. If they require s_v it should be carefully examined whether the artifact is really a domain property and not a proper requirement.
R and S	In Event-B, it is sometimes difficult to separate proper Requirements from Specification. Both are typically expressed in terms of e and s_v. Both are often traced to **invariants** and **axioms**. We also found **dedicated refinements** useful to represent them (see Section 3.4). We can extend tracing further by using **additional formalisms** (see Section 4.1).
P	The final program P is typically implemented with its own execution environment (M). The final refinement is often already an incomplete implementation of P. A conversion into P is often straight forward.

First, many artifacts can be expressed as invariants. Once a proper requirement is expressed as an invariant, we can use proof obligations to guarantee that the invariant will not be violated. Proper requirements that cannot be easily expressed as invariants can be structured using refinement (see Section 3.4), or modeled in a different formalism (see Section 4.1).

Second, Event-B supports refinement as described above. WRSPM comes with a simplified view of refinement very similar to the one described in the introduction of [13]:

$$\forall e\ s \cdot W \wedge M \wedge P \implies S \tag{3}$$

If (1) holds, then (3) holds as well, P being a refinement of S. In practice, an Event-B model consists of several refinements, forming a chain of machines. Refinements can be used to incorporate more proper requirements R, to make modeling decisions S or to provide implementation detail P. Event-B allows to mix these three purposes in one refinement, but we suggest to give every refinement just one single purpose (described in Section 3.4).

Third, if all Event-B proof obligations are discharged, then we know that the model is consistent in the sense described above.

Last, Event-B has no intrinsic mechanism to distinguish W, R, P and S. This means that we have to be careful to track the meaning of Event-B elements in the context of WRSPM. We suggest to use refinements for structuring and naming conventions.

To demonstrate that the WRSPM model does the right thing, we want to show that

$$\forall e\ s \cdot W \wedge R \wedge S \wedge P \text{ realize } REQ \tag{4}$$

We use "realize" instead of an implication, because REQ is informal. We cannot prove that (4) holds, we can merely justify it. The aim of our approach is to make this justification systematic and scalable (see Section 2.3).

The requirements REQ are rarely ready to be modelled according to our approach in their initial form, as provided by the stakeholders. Figure 4 depicts the iterative process for building the WRSPM-artifacts from the requirements.

2.2 Traceability to Natural Language Requirements

A key contribution of this paper is the traceability between natural language requirements and the Event-B model which is structured according to WRSPM. It allows us to cope with changes in the model and changes in the requirements. Our approach distinguishes the following three types of traces:

Evolution Traces: As the model evolves over time, there is traceability from one iteration to the next (as indicated by the horizontal arrows in Figure 3). This is particularly useful for the stakeholders to verify that changes to the requirements reflect their intentions. This can be done by exploring the requirement's evolution over time, allowing the stakeholder to compare the original requirement to the modeler's revision.

Explicit Traces: Each non-formal requirement is explicitly linked to at least one formal statement. These traces are annotated with a justification that explains why the formal statement corresponds to the non-formal requirement.

Implicit Traces: There is implicit traceability within the Event-B model. Those traces can be discovered via the model relationships (e.g. refinement relationships, references to model elements or proof obligations). For instance, a guard that ensures that an invariant holds is implicitly linked to that invariant via a proof obligation. Furthermore, it is possible to use the identifiers of phenomena in the non-formal requirements, in addition to their use in the formal model. This would allow for implicit traceability to REQ as well, if we use the identifiers consistently in the natural language requirements.

Tracing an element of the formal model to an original requirement may require following a chain of traces.

2.3 Dealing with Change in Requirements and Model

The established traceability allows us to *validate* systematically that every requirement has been modeled as intended. We validate a requirement by using the justifications of the traces to reason about the requirement and the corresponding model elements.

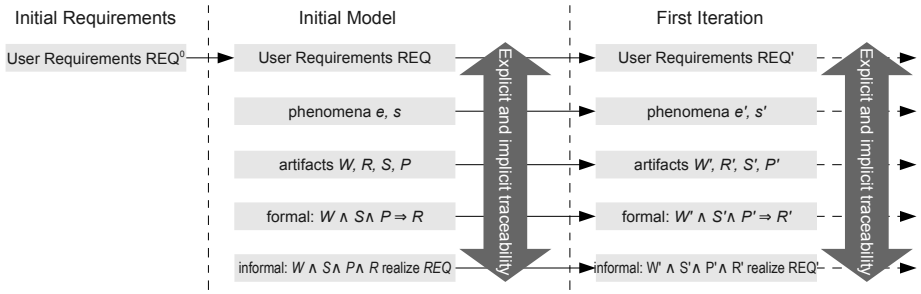

Fig. 3. Traceability between Iterations and within Iterations

The Event-B model may contain elements that are not directly associated with a requirement through a trace. These are elements that are necessary for making the model consistent. For instance, events may have guards that are necessary to prevent invariants from being violated. Such elements are implicitly traced, and can ultimately be traced all the way back to a requirement through a chain of traces. Allowing to annotate those implicit traces could be useful at times to explain the shape of a model.

There are also Event-B elements that are part of the design or implementation. Such elements are not always traced, as the information contained in them may not be part of *REQ*. They should be annotated in order to make the model understandable.

3 Application of the Approach

Now that we introduced our approach, we will demonstrate the concepts with the running example from Section 1.1. We follow the process depicted in Figure 4 by selecting a requirement to start with.

REQ-2	The traffic light for the cars has the colors red, yellow and green

3.1 Modeling Phenomena

We identify the following five phenomena in the text of REQ-2. We provide the Event-B identifier in parentheses[3]:

s_v**: traffic light for the cars (**`tl_cars`**)** We model the traffic light as the variable `tl_cars`, controlled by the system and visible to the environment.

e_v**: colors (**`COLORS`**)** We model colors as a set. This is a phenomenon of the environment that is visible to the system and provides typing of `tl_cars`.

[3] As a convention, we write environmental phenomena in uppercase and system phenomena in lowercase

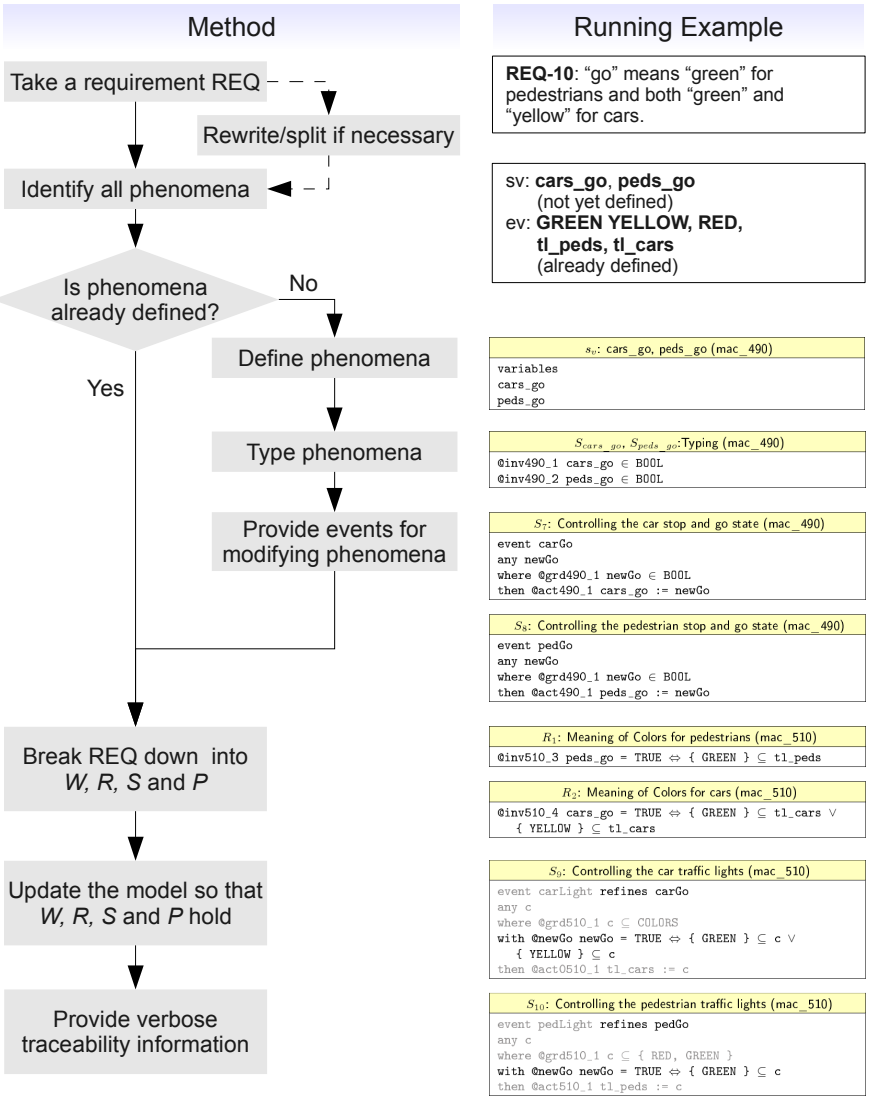

Fig. 4. Processing individual requirements. The running example is described in Section 3.3

e_v**: red, yellow, green (**`RED`**,** `YELLOW`**,** `GREEN`**)** We model the actual color values as constants of type `COLOR`.

With this information we can rephrase REQ-2 and model the phenomena in Event-B. First we decompose REQ-2 into two requirements REQ-2-1 and REQ-2-2. These are connected by evolution traces to REQ-2:

REQ-2-1	`tl_cars` consists of `COLORS`

REQ-2-2	`COLORS` is the set of `RED`, `YELLOW` and `GREEN`

After declaring the phenomena in Event-B, we can create explicit declaration traces to the corresponding requirements REQ-2-1 and REQ-2-2.

The phenomena are defined through typing invariants (for instance, the typing of `tl_cars` is $tl\_cars \subseteq COLORS$). These traces are implicit, as they can be extracted from the formal Event-B model.

The requirement REQ-2-1 is realized as an invariant (the same as the typing invariant) and REQ-2-2 as an axiom (the partitioning of colors). These are explicit traces that we have to establish by hand.

Last, we have to provide an event to modify the state of the traffic light. There is an implicit trace ("changed by") between this event and the variable `tl_cars`. This is expressed in Event-B as follows:

Controlling the car traffic lights

```
event carLight
any c where c ⊆ COLORS
then tl_cars := c
```

Note that there is nothing yet constraining which of the three lights are on or off. At this stage, the system could still evolve into a disco light, because REQ-2 describes the domain, rather than how it is supposed to behave (the model elements are part of W). In subsequent refinements, the behavior is constrained more and more, as new requirements and design are incorporated into the model. We will demonstrate this in Section 3.4.

3.2 Modeling Requirements as Invariants

If possible, we model requirements as invariants. Once modeled this way, Event-B ensures that the invariant will never be violated (assuming that all proof obligations are discharged). Consider REQ-9:

REQ-9	The lights for pedestrians and cars must never be "go" at the same time

We omit the declaration and definition of the phenomena for brevity, and go straight to the rewritten requirement:

REQ-9-1	**car_go** and **ped_go** must never be **TRUE** at the same time

This can be traced using a "realizes" trace to the following invariant:

Formal representation of REQ-9-1

```
¬ (cars_go = TRUE ∧ peds_go = TRUE)
```

3.3 Traceability and Data Refinement

In Section 3.1 we introduced `tl_cars` and in Section 3.2 we introduced `car_go`. These two variables are connected through REQ-10, and we can realize this connection through refinement in Event-B. This is also depicted in Figure 4.

REQ-10	"go" means green for pedestrians and green or yellow for cars.

This requirement can be rewritten using the previously introduced names for the phenomena in question:

REQ-10-1	**peds_go = TRUE** means **GREEN** is active for **tl_ped**

REQ-10-2	**cars_go = TRUE** means **GREEN** or **YELLOW** is active for **tl_cars**.

The phenomena relating to the colors would be introduced in a machine that refines the one that introduced stop and go. Thus, the relationships (and thus REQ-10-1 and REQ-10-2) are realized through gluing invariants:

Meaning of Colors for pedestrians
`peds_go = TRUE ⇔ { GREEN } ⊆ tl_peds`

Meaning of Colors for cars
`cars_go = TRUE ⇔ { GREEN } ⊆ tl_cars ∨` `  { YELLOW } ⊆ tl_cars`

REQ-10-1 and REQ-10-2 and their gluing invariants are connected via explicit traces. The Event-B model contains a number of relevant implicit relationships that ensure that the model is consistent. For instance, the event that modifies `peds_go` has a corresponding event in the refinement that modifies `tl_peds`. Due to the gluing invariant, we can only discharge all proof obligations if the refinement preserves the properties of the abstract model. The corresponding Event-B is depicted in Figure 4.

3.4 Structuring Requirements using Refinement

Some requirements are difficult to model as invariants. Consider the following:

REQ-12	The pedestrian light always follows the sequence red – green

REQ-12 is difficult to express as an invariant due to its temporal nature. We realize it by refining the event `pedLight` into two distinct events, `pedsRedToGreen` and `pedsGreenToRed`. This is the point where the traffic light is forced to behave differently from a "disco light". We can verify by inspection, model checking or animation whether the formal model reflects the requirement. In this particular case, we could animate the refinement (e.g. using ProB for Rodin [17]) to convince ourselves that red follows green and green always follows red. (This could also be stated in temporal logic, see Section 4.1).

Formal representation of REQ-12

```
event pedsRedToGreen refines pedLight
where
 ¬({GREEN} ⊆ tl_cars ∨ {YELLOW} ⊆ tl_cars)
 tl_peds = { RED }
with @c c = { GREEN }
then tl_peds := { GREEN }

event pedsGreenToRed refines pedLight
where
 ¬({GREEN} ⊆ tl_cars ∨ {YELLOW} ⊆ tl_cars)
 tl_peds = { GREEN }
with @c c = { RED }
then tl_peds := { RED }
```

This is an example where the model must be changed in various places. By using a dedicated refinement for this requirement, the changes in the model are comprehensible. Thus, we would establish an explicit trace to this Event-B machine, and we would not make any other changes to this refinement.

3.5 Requirements Outside the Formal Model

Some requirements are very hard to model in Event-B. Consider the following requirement REQ-16:

REQ-16	The length of the green phase of the pedestrian light has a specified duration.

Due to the temporal nature of the requirement, this requirement is hard to express in the formalism we chose. One option would be to introduce the concept of "ticks" that progress time on a regular basis. But even if we do that, it is not clear how long a tick is. We could also leave this requirement completely out of the model, leaving an aspect of the system that is not accounted for in the formal model. In our approach, this would manifest as a requirement without a trace to the formal model. Such untraced requirements are easily identified and must then be accounted for by other means, typically by providing for them directly in the implementation.

4 Related Work

The issue of traceability has been analyzed in depth by Gotel et. al. [10]. Our research falls into the area of post-requirements specification traceability.

Abrial [2] recognizes the problem of the transition from informal user requirements to a formal specification. He suggests to construct a formal model for the user requirements, but acknowledges that such a model would still require informal requirements to get started. He covers this approach in [3].

The WRSPM reference model [11] was attractive, because it deliberately left enough room to be tailored to specific needs, as opposed to more concrete methods like Problem Frames [14] or KAOS [9]. It is also more formal and complete than the functional-documentation model [20], another well-known approach.

The idea of the WRSPM reference model has been advanced in current research. In [19], the authors introduce a model of formal verification based on non-monotonic refinement that incorporates aspects of WRSPM. Problem Frames [14] could be useful for identifying phenomena and for improving the natural language requirements that we start out with, thereby complementing our approach. In [18], the authors show how Event-B and Problem Frames are being applied to an industrial case study. We drew some inspiration from this work, especially with regard to the relation between WRSPM and Event-B.

Some ideas in this paper are related to KAOS [9], a method for requirements engineering that spans from high-level goals all the way down 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 real-time temporal expressions. Our approach distinguishes itself from KAOS by being very lightweight: KAOS uses many more model elements and relationships. KAOS also covers many more aspects of the system development process than our approach, which results in an "all or nothing" decision. We believe that our approach can easily be integrated into existing workflows and processes.

Reveal [22] is an engineering method based on Michael Jackson's "World and the Machine" model, which is compatible with WRSPM. Therefore we believe that our approach could be integrated nicely with Reveal.

4.1 Other Formalisms

Rather than using Event-B to model all artifacts, nothing is preventing us from choosing different formalisms. We demonstrate this in the following, where we model a requirement using linear temporal logic (LTL). LTL can actually be understood as an extension to Event-B, complementing its standard proof obligations.

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 Event-B model.

REQ-11	The traffic light for the cars always follows the sequence: green → yellow → red → red/yellow

REQ11 Sequence of car-lights (LTL)
$G(\{tl\_cars = \{green\}\} \implies (\{tl\_cars = \{green\}\}\ U \{tl\_cars = \{yellow\}\})) \wedge$ $G(\{tl\_cars = \{yellow\}\} \implies (\{tl\_cars = \{yellow\}\}\ U \{tl\_cars = \{red\}\})) \wedge$ $G(\{tl\_cars = \{red\}\} \implies (\{tl\_cars = \{red\}\}\ U \{tl\_cars = \{red, yellow\}\})) \wedge$ $G(\{tl\_cars = \{red, yellow\}\} \implies (\{tl\_cars = \{red, yellow\}\}\ U \{tl\_cars = \{green\}\}))$

This requirement can now be validated through model checking. Rodin can evaluate LTL expressions with the ProB model checker [21], which exists as a well-integrated Plug-in for Rodin.

5 Conclusion

In this paper, we presented an approach for building a formal model from natural language requirements. With our approach, the boundary between informal requirements and formal model is clearly defined by annotated chains of traces, which keep track of model evolution and explicit and implicit links. We present a number of approaches for modeling requirements and for providing traceability: Some requirements can be traced elegantly to invariants, and those that can not, can be structured using refinement. We can validate the traces in a systematic fashion and analyze the impact of changes in the requirements or the model.

In addition to the explicit traceability between requirements and model, we take advantage of the implicit traceability within the formal model to support us in verifying the model against the requirements. In particular, we take advantage of traceability through proof obligations: When all proof obligations are discharged, we know that the model is consistent. If we trust our traceability, then we have confidence that our requirements are consistent as well. Common identifiers can be used in the informal requirements and formal model. A supporting tool could support the user by pointing out matching identifiers.

We will also explore change management further. Requirements model and formal model are closely linked via the traceability information. Changes in either model will affect the other.

Our approach has proven successful with a number of small projects. In the near future, we will tackle bigger case studies; we will incorporate ongoing research like decomposition [8]. As of this writing the effort for building tool support within the Rodin platform is well under way [4].

Acknowledgements

The work in this paper is partly funded by Deploy[5]. Deploy is an European Commission Information and Communication Technologies FP7 project.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 2005.
2. J.-R. Abrial. Formal Methods in Industry: Achievements, Problems, Future. In *Proc. of the 28th int. conf. on Software engineering*, pages 761–768, 2006.
3. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, June 2010.

[4] http://www.pror.org
[5] http://www.deploy-project.eu

4. J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In *International Conference on Formal Engineering Methods (ICFEM)*, LNCS, New York, NY, 2006. Springer-Verlag.
5. D. M. Berry. Formal Methods: The Very Idea – Some Thoughts About Why They Work When They Work. *Science of computer Programming*, 42(1):11–27, 2002.
6. D. Bjørner. From Domain to Requirements. In *Concurrency, Graphs and Models: Essays dedicated to Ugo Montanari on the Occasion of his 65th Birthday*, pages 278–300. Springer, 2008.
7. Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
8. M. Butler. Decomposition Structures for Event-B. In M. Leuschel and H. Wehrheim, editors, *IFM*, volume 5423 of *LNCS*, pages 20–38. Springer, 2009.
9. R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering. In *Proc. of the 19th int. conf. on Software engineering*, pages 612–613. ACM, 1997.
10. O. Gotel and A. Finkelstein. An Analysis of the Requirements Traceability Problem. In *Proc. of the First Int. Conf. on Requirements Engineering*, pages 94–101, 1994.
11. C. A. Gunter, M. Jackson, E. L. Gunter, and P. Zave. A Reference Model for Requirements and Specifications. *IEEE Software*, 17:37–43, 2000.
12. Stefan Hallerstede and Michael Leuschel. How to Explain Mistakes. In Jeremy Gibbons and José Nuno Oliveira, editors, *TFM*, volume 5846 of *LNCS*, pages 105–124. Springer, 2009.
13. C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
14. M. Jackson. *Problem Frames: Analysing and Structuring Software Development Problems*. Addison-Wesley/ACM Press, 2001.
15. M. Jastram. Requirements Traceability. Technical report, U. Southampton, 2009.
16. C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.
17. M. Leuschel and M. Butler. ProB: An Automated Analysis Toolset for the B Method. *Int. Journal on Software Tools for Technology Transfer*, 10(2):185–203, 2008.
18. F. Loesch, R. Gmehlich, K. Grau, C. Jones, and M. Mazzara. Report on Pilot Deployment in Automotive Sector. Technical Report D7, DEPLOY Project, 2010.
19. J. Marincic, H. Wupper, A. H. Mader, and R. J. Wieringa. Obtaining Formal Models Through Non-Monotonic Refinement. 2007.
20. D. L Parnas and J. Madey. Functional Documents for Computer Systems. *Science of Computer programming*, 25(1):41–61, 1995.
21. D. Plagge and M. Leuschel. Seven at One Stroke: LTL Model Checking for High-Level Specifications in B, Z, CSP, and More. *Int. Journal on Software Tools for Technology Transfer*, (1), 2008.
22. Praxis. Reveal – A Keystone of Modern Systems Engineering. Technical report, 2003.