Verification of UML models by translation to UML-B

Colin Snook, Vitaly Savicks and Michael Butler

University of Southampton

Abstract. UML-B is a 'UML like' notation based on the Event-B formalism which allows models to be progressively detailed through refinements that are proven to be consistent and to satisfy safety invariants using the Rodin platform and its automatic proof tools. UML, on the other hand, encourages large models to be expressed in a single, detailed level and relies on simulation and model testing techniques for verification. The advantage of proof over model-testing is that the proof is valid for all instantiations of the model whereas a simulation must choose a typical instantiation. In the INESS project we take an extant UML model of a railway interlocking system and explore methodical ways to translate it into UML-B in such a way as to facilitate proof that the model satisfies certain safety properties which are expressed as invariants. We describe the translation attempted so far and insights that we have gained from attempting to prove a safety property. We propose some possible improvements to the translation which we believe will make the proof easier.

1 Introduction

The aim of the INESS project [1] is to develop specifications and associated material to assist in the development of a European common standard for railway interlocking systems. The role of the WP4 group within INESS is to develop ways to verify UML models of such interlocking systems. Other partners in WP4 are using model-checkers to verify a translation of the UML models. While this method benefits from a high degree of automation, the size of models that can be verified is limited by performance constraints and each instantiation of the model (i.e. interlocking layout) has to be verified separately. We are exploring the alternative approach of using theorem provers to verify (a translation of) the model. While this approach generally requires a higher degree of expert intervention depending on the size and complexity of the model, it does not suffer from state-explosion to the same degree and once completed is valid for all possible layouts that satisfy the constraints of the model. Proof however, can quickly become intractable in real-world problems. The key to proving something, therefore, is to abstract away from the details and prove much simpler properties which can then be used as lemmas in the proof at the more detailed level. Refinement is the process of introducing more detail into the model in order to move from the simpler abstract version to the detailed concrete level and

requires further proof to ensure that the concrete model is consistent with the abstraction. The refinement that we use can be broadly categorised into horizontal, superposition where more details are added without altering the representation of the abstract model, and vertical data refinement where the abstract model is replaced with a more complex version. The latter is, in general, a more powerful form of abstraction and involves more substantial proof to show the correspondence between the models. In fact, the intention of the refinement is generally given in the form of a 'gluing invariant' which specifies the relationship between the abstract variables and the concrete ones. The gluing invariant may also be used as a lemma in the proof of properties at the detailed level.

In this paper we report on an investigation into translating a UML model of an interlocking system into a formal refinement based notation so that safety properties can be formally proven to hold. We use a small given example of an interlocking model which is constructed in a style which is proposed by the INESS project for use in the railway industry sector and we select and prove one safety property from its requirements specification to demonstrate our method. We would normally recommend considering safety properties at the earliest possible stage [2] but this requires expertise in finding useful abstractions. Instead, we adopt a rather naive approach where our refinements are constructed mechanistically from the UML structure without abstracting the main concepts involved in the safety property and we then attempt to add the safety property at a late stage. The purpose of taking this approach was to determine to what degree a mechanistic, low expertise, approach would succeed and where we would need to introduce more abstraction in order to succeed in the proof. Indeed, when we initially tackled the proof of the safety requirement, not only did we find it difficult, but also, we could not detect whether it should be provable or whether there was a problem in the model. For a second attempt, we introduced a more useful abstraction into the model which enabled us to detect some errors in the original UML source model and then prove the safety requirement in a more abstract form. We were then able to prove the more concrete form of the safety requirement using the abstract property.

Our goal in the INESS project is to develop techniques for using our methods to prove safety requirements in UML models of interlocking systems. As depicted schematically in Fig. 1, to do this we need to translate the UML model and its associated safety requirements into our notation where we can apply the automatic prover. The primary aim of this paper is to report on our ongoing work on INESS. However, we also see a more general contribution emerging which is a set of guidelines for building formal models from semi-formal entity relationship and state machine models (including UML models) so that the process is better defined and it is clearer where stronger abstractions will be needed than those that can be inferred mechanistically.

The paper is structured as follows. In section 2 we describe the methods that we use. In section 3 we describe the source UML model that we were given to verify. In section 4 we describe our mechanistic approach to translating the UML model into a series of UML-B refinements. In section 5 we describe our

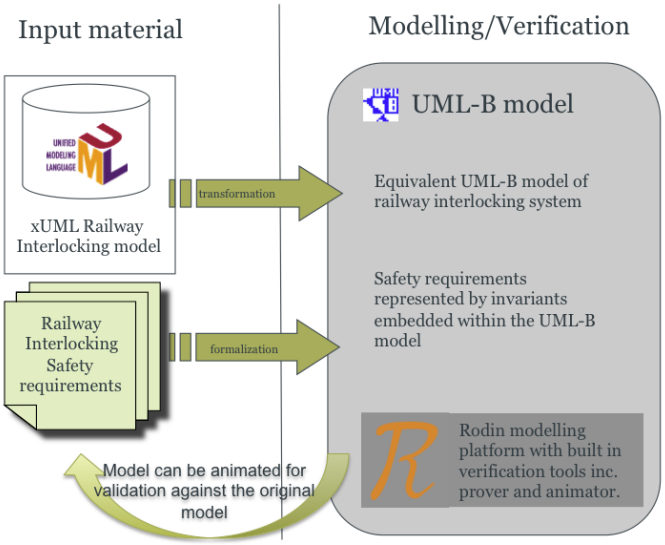

Fig. 1. Schematic block diagram illustrating the translation of the UML model and associated safety invariants into the UML-B notation

interpretation of a safety requirement and how the pursuit of a proof led us to revise our modelling approach and to uncover safety problems in the original model. Section 6 outlines how we intend to develop our approach in response to the findings reported here. Section 7 is the conclusion.

2 Background

The Unified Modelling Language (UML) [3] is a semi-formal diagrammatical modelling notation which has been adopted relatively widely throughout industry. UML includes several diagram notations which can be used for different aspects of a system and for different stages of the development process. The UML diagram notations that we are concerned with are 'Class Diagrams' which can be used to show the relationships between different kinds of entities and 'State Diagrams' which give a state oriented view of the behaviour of the classes. To a lesser extent we are also interested in 'Use Case Diagrams' and 'Sequence Diagrams' which are used together to illustrate the requirements of a system. UML is often interpreted flexibly and to some extent this is a strength, but it is sometimes criticised for having imprecise semantics which may lead to confusion and ambiguity. Some variants of UML have been developed which are constrained and more precisely defined. UML has no notion of refinement.

Event-B [4, 5] is a state-oriented formal modelling language that was developed for modelling at the systems level. State is represented by typed variables

and spontaneous transitions (guarded events) occur to alter the state. A central concept of Event-B is refinement, where more detailed state is added and this reveals more detailed events. The Rodin platform has been developed as a formal modelling environment for Event-B and includes a static checker and a prover. The Rodin platform [6, 7] is designed to be extensible and many plug-in's are available which extend the Event-B language or provide additional tools for model development, verification and validation.

We base our approach upon UML-B [8, 9], Event-B and the Rodin platform. UML-B is a visual front-end for the Event-B notation and includes a state machine diagram editor. Tool support for UML-B is provided by a plug-in to the Rodin platform. State machines may be refined by adding nested state machines [10] and can be animated via a plug-in [11] that utilises the ProB [12] model checker and animator. The state machine refinement supported by UML-B allows the model to be progressively developed in stages. This improves understanding, and hence validity, as features can be laid down in small steps while the Rodin provers ensure consistency. Invariants can be added to states (i.e. the state being active becomes an implicit antecedent for the property) providing another mechanism to clearly state our understanding of the model with further consistency checking via the Rodin provers. One use for invariants is to express safety properties that we believe the model satisfies so that we can prove that this is indeed the case. Animation of the state machine diagrams allows us to test that they behave as we expected.

3 UML model of Interlocking

For the purpose of investigating and demonstrating how to apply our methods in this domain, we were given a small UML model of an interlocking system, called micro2010 [13], which had been constructed in a style that is typical of the full scale models that are expected to be produced during or after the INESS project. The model is constructed using the Artisan UML tool [14] in a variant of the UML called xUML [15] which can be executed in a simulation environment called Cassandra [16]. xUML has a precise but easy to read action and constraint language which is interpreted by the Cassandra simulation tool. The advantage of this model is that it has an operational semantics as defined by the Cassandra simulation.

The micro2010 UML model is based on a structure of classes as shown in Fig 2. The class *HAL* represents the control of a physical device in the system. Hence the class structure making up the modelled interlocking system controller mirrors the components in the physical track layout. There are subclasses for *signal*, *track* and *point*. The class, *route*, represents a concept used in railway interlocking systems where a collection of track layout components constitute a path through the layout that a train may wish to follow. The safety of the system is based on a route being requested and subsequently allocated provided that all the components in that route are free and not allocated for use in any other route. The route class therefore has several associations with the logical

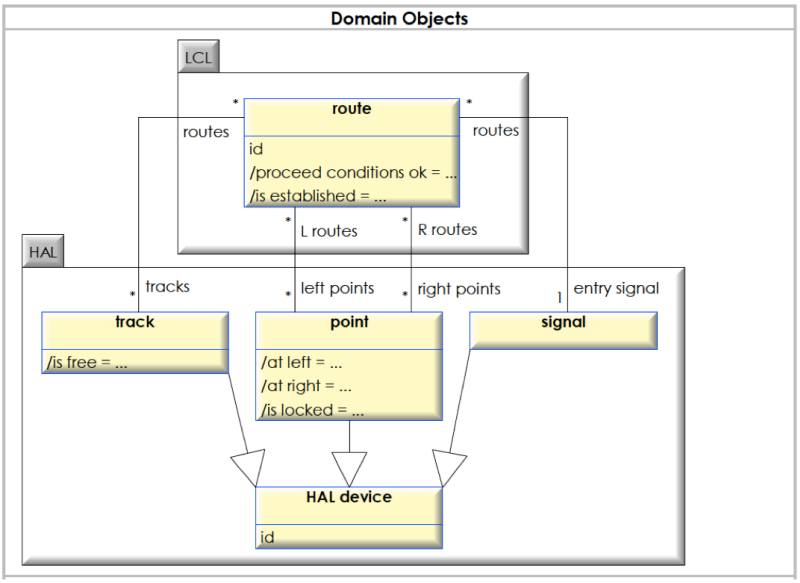

Fig. 2. UML Class Diagram showing structure in the interlocking model

device classes which identify the components involved in that route. Note that, for verification by proof, we do not need to specify the values of these classes and associations, the proof is valid for any valid instantiation, whereas, for model checking or simulation an example configuration is configured or generated by instantiating these classes and giving values to all the associations.

The classes often own derived attributes which represent a boolean condition over properties of the class and its associated classes. Derived attributes (designated by a preceding slash /) do not represent any new state, they are a shorthand way of expressing a boolean condition over variables that are elsewhere in the model. For example, the *point* class has a derived attribute, */is_locked*, which is true if and only if that point belongs to a route that is currently established. There is no actual variable attached to */is_locked*, so it can never be assigned to in an action but it may appear in guards as a shorthand for its definition.

The behaviour of each class is specified by a state machine diagram. Examples of two such state machines are shown in Figs 3 and 4. The state machines are composed in a hierarchical manner with some states containing sub-states. The behaviour of the system is specified in terms of guarded transitions that may fire when the system is in their source state. For convenience, guards are usually specified using a derived attribute (e.g. *moveLeft[not /isLocked]* in Fig 3). There are several different ways by which a transition may be triggered. Transitions stereotyped *ic* are triggered by a send action in another transition (e.g. *moveLeft* in Fig 3). Transitions stereotyped *dv* are triggered by external events in the environment (e.g. *atLeft* in Fig 3). Transitions named with the keyword 'after(t)' are triggered spontaneously after the time t since the source

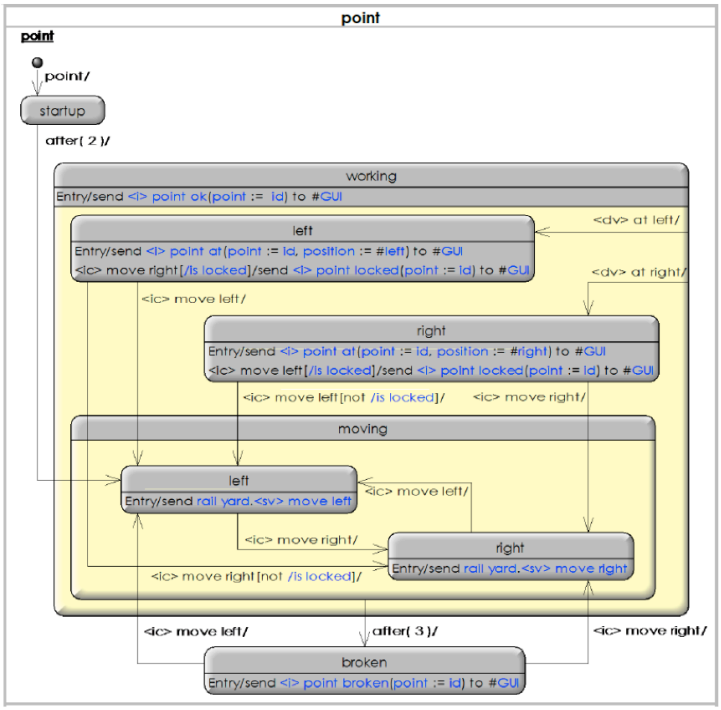

Fig. 3. UML State Machine of Points

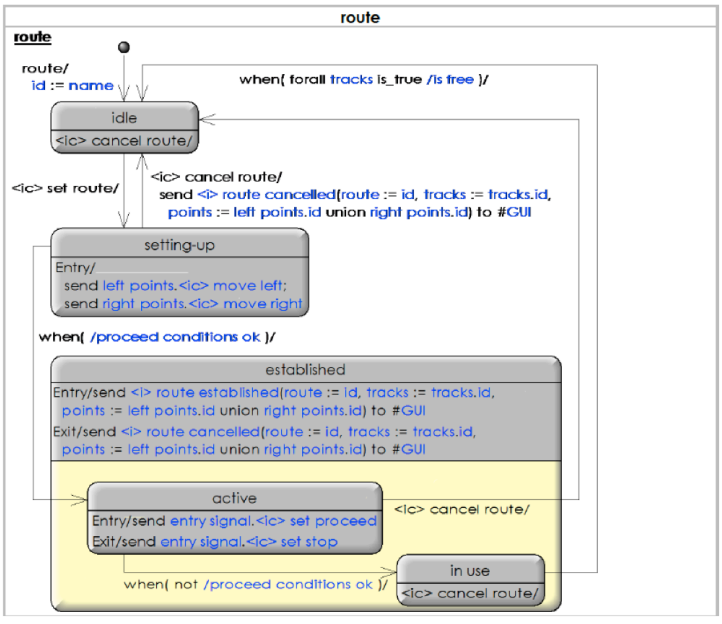

Fig. 4. UML State Machine of Routes

state was entered (e.g. *after(3)* in Fig 3). Transitions named with the keyword, 'when', are triggered spontaneously when their guard condition is true (e.g. *when([/proceed-conditions-ok])* in Fig 4).

4 Translation to UML-B

UML-B provides equivalent modelling diagram notations to those used in the UML model (i.e. Class and State-machine) but with minor syntactic and significant semantic differences. In general UML-B is less flexible because it is strongly founded on the Event-B formalisation, but in most cases there are no significant problems in translating the diagrams. We also introduced some new features into UML-B in order to assist the translation. However, the semantic differences in transitions deserves some attention. Due to their correspondence with Event-B events, the only mechanism for triggering transitions in UML-B is spontaneous triggering when the transition guard is true ('when' transitions). The other transition triggering methods used in the UML model are handled as follows.

Externally triggered transitions are modelled using spontaneous triggering in the same way as 'when' transitions. Since we do not explicitly model the environment, the transition is considered to implicitly represent the response to an event occurring in the environment.

Timed transitions are modelled using spontaneous triggering in the same way as 'when' transitions. Since we do not explicitly model time, the transition is considered to implicitly represent the response to a time limit being reached. Since the properties we wish to verify only concern event ordering, this is sufficient for our purposes provided that there are no cases where two timing events compete from the same system state.

Internally triggered transitions are represented by modelling a message passing system. A base class *buffer_owner* is inherited by all other classes and provides an attribute which is a set of messages received by an object in that class. A transition that needs to be internally triggered waits for an instance of its trigger message to be received by its owning object and removes that message as it fires. In a UML-B refinement, new transitions may only modify variables that have been introduced in that refinement and are not allowed to alter the variables introduced in previous refinement levels. This means that we can not alter the buffer in any subsequent refinement levels after we first introduce the buffer. To avoid this problem we provide an event that non-deterministically modifies the buffer attribute so that transitions introduced in subsequent refinements may refine this behaviour by sending trigger messages to other objects and removing their own trigger messages. Note that although this mechanism does not impose any ordering on triggering within an object and does not allow for multiple triggering of the same transition it is sufficient for the safety properties being verified.

As in Event-B, refinement is a key concept within UML-B. Even without introducing a safety property, there are significant proof obligations concerning the internal consistency and well-definedness of the model and these would

be difficult to handle if the model was introduced in a single stage without refinement. Therefore, we need to build the UML-B model in several refinement stages. Since this is a research experiment aimed at finding a method that can be applied to bigger and more general UML models of interlocking systems, we also require that the method of introducing refinements is methodological and does not rely on high expertise in refinement and abstraction. Three potential methods for introducing refinement are considered.

a) Class inheritance has some analogy with refinement. Classes higher up the inheritance structure can be introduced and modelled without knowledge of subclasses. These inherited classes have features that are common to their subclasses which can be modelled in an abstract level before the subclasses are introduced in a subsequent refinement. In this case the micro2010 model contains one inherited class, *HAL* which does not contain any interesting behaviour. Therefore we do not use this technique in this translation.

b) Class associations and behaviour can be examined for dependency and used to determine a priority ordering for introducing classes. It is usually the case that the classes exhibit a hierarchy in the way that they control each other with one class responding to instructions from another (via the internal triggering mechanism). The associations give a clue to this hierarchy because internal triggering occurs between instances that are linked by associations. In the class diagram of the micro2010 model (Fig. 2) observe that the *track*, *signal* and *point* classes are not connected by association whereas the *route* class connects with all three. It is apparent that the *track*, *signal* and *point* classes are independent and can be introduced in any order whereas the *route* class is dependent on the others and must be introduced last. This is confirmed by examining the internal triggering where the *route* class is the only one that sends triggers while the other classes are only responding to them.

c) State machine hierarchy can be used to introduce the full behaviour in stages following the hierarchy of nesting within the class' state machine. Nested state machines only elaborate the behaviour of the parent state machine which can be constructed in a way that is valid without the nested one. We use this method to introduce the *point* class in three refinements and again to introduce the *route* class in two refinements

These 3 methods should be used in the order shown since this will result in an appropriate ordering with respect to class dependencies. We also suggest using the 3 methods in a 'depth first' manner. That is, the highest (most abstract) level classes in the inheritance structure should be introduced first and fully refined using methods b and c before the next level in the inheritance structure is dealt with. Similarly the classes introduced by analysing the associations should be fully refined by method c before the next priority according to associations is dealt with.

There is no facility in UML-B for representing derived attributes. Therefore derived attributes are fully expanded with their definitions wherever they are used and only introduced at a refinement level where all the features required

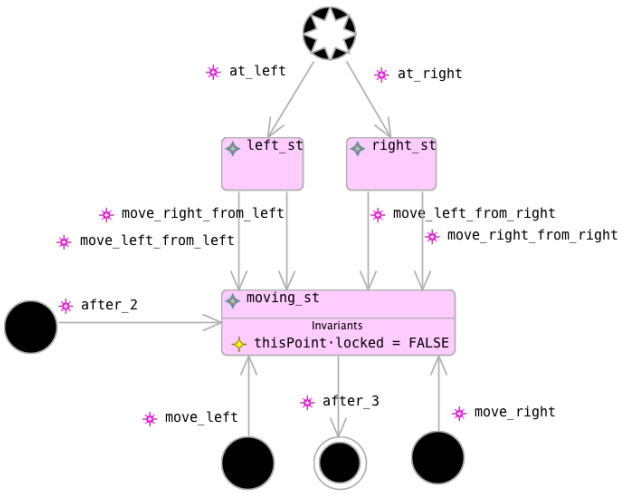

Fig. 5. UML-B State Machine Diagram showing the behaviour of points in the working state

by their definitions have been added to the model. However, our experience of proving (described in chapter 5) has led us to the conclusion that there needs to be more abstraction of features of the model. We are now re-assessing whether these derived attributes represent useful abstractions that should be introduced in early refinements for the purpose of proving important properties.

The refinement steps of the UML-B version of the railway interlocking model can be summarised as follows: (The application of the methods outlined above is shown for each refinement).

m0 Introduces the message passing mechanism used for internal triggering of transitions. (No method - prior to translation).

m1 Introduces the *signal* class and its state-machine describing its behaviour which sets the signal to *proceed* or *stop* in response to internal triggers. (Method b).

m2 Introduces the *track* class and its state-machine describing its behaviour which sets the track to *free* or *occupied* in response to internal triggers. (Method b).

m3 Introduces the *point* class and its first level state-machine describing its behaviour to go to the *broken* state if a command is not achieved within a time limit and to recover if another internal trigger is received. (Method b).

m4 Elaborates the behaviour of the *point* class when it is in the *working_st* state by adding the nested state-machine shown in Fig. 5. (Method c).

m5 Elaborates the behaviour of the *point* class when it is in the *moving_st* state by adding the nested state-machine shown in Fig. 6. (Method c).

m6 Introduces the *route* class and its first level state-machine shown in Fig. 7. (Method b).

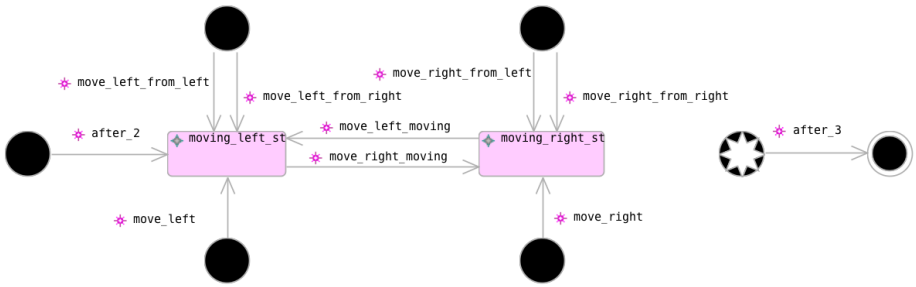

Fig. 6. UML-B State Machine Diagram showing the behaviour of points in the moving state

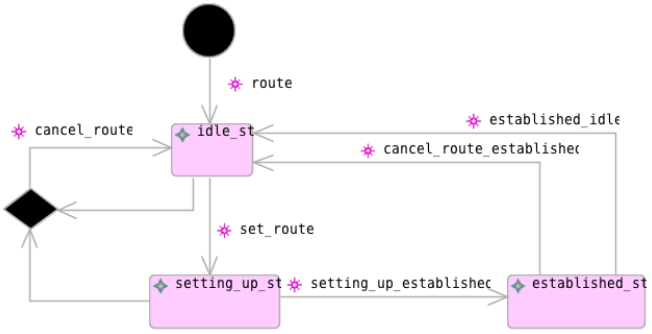

Fig. 7. UML-B State Machine Diagram showing the behaviour of routes

m7 Elaborates the behaviour of the *route* class when it is in the *established_st* state by adding a nested state-machine. (Method c).

m8 Adds the safety invariant to check that points that are in an established route do not move. (No method - add invariant).

In order to reason about the UML-B model and, in particular, to discuss the invariants, we need a textual representation of the value of a state machine. This is provided by the UML-B toolset since it automatically translates the diagrams into an equivalent Event-B model for verification purposes. The tool translates each class into a set with the same name as the class, and each state machine to a variable using the name of the state machine. (As a convention we have named the state machines after their parent with "_sm" appended). The state machine variable is a function from the set representing the class to an enumeration of the states in the state machine. The value of a state machine for a particular instance of a class is therefore available by function application. Associations are constant relations between the class sets and values can be accessed by relational image. Hence we can refer to the left points of a route, r, as $left\_points[\{r\}]$ and a point, p, being in the left position as $point\_sm(\ p) = left\_st$.

5 Proving the Safety Invariant

Once the UML-B model is developed throughout its levels of refinement, we can turn our attention to verifying that it satisfies the safety requirements. Up to this stage, until we introduce some safety requirements, the proof only concerns:

a) well-definedness (e.g. that where a function application is used, the function is defined for that value),

b) typing (i.e. that an assignment doesn't contravene a defined sub-range of a variable's basic type) and

c) simulation (i.e. that the the principles of refinement are observed).

These proof obligations, which are mostly discharged automatically by the Rodin provers[1], ensure that the model is constructed correctly in a consistent manner but do not prove anything about how the model behaves.

A more interesting and challenging use of proof is to introduce some invariant property which we require to hold at any time in the model. These invariant properties are ideal for expressing safety requirements. The micro2010 UML model contains four safety requirements which are stated in natural language in the documentation. We choose one of these safety requirements for the purpose of illustrating our method.

SR1: A point that is locked by an established route shall never move.

[1] Some theorems were added at the first refinement to help the provers discharge POs about the refinement of buffer assignments

The safety requirement is worded with a little redundancy since, by definition, a point is locked by establishing a route to which it belongs. Since we have expanded away all derived attributes, such as */is_locked*, in our UML-B model, we re-state the safety invariant as follows:

> SR1': A point that belongs to the left points or right points of an established route shall never move.

We can now translate this into an invariant using the features of our UML-B model. When translated to plain Event-B this state invariant becomes::

$$\texttt{SR1}: \forall r, p \cdot r \in route \land route\_sm(r) = established\_st \land p \in (left\_points[\{r\}] \cup right\_points[\{r\}]) \Rightarrow (p \in dom(working\_sm) \Rightarrow working\_sm(p) \neq moving\_st)$$

SR1 is read as follows: For all p and r, where r is a route which is in the state *established_st* and p belongs to the union of the *left_points* and *right_points* of r then (if p is in the domain of the state machine, *working_sm* then) p is not in the state *moving_st*. Note that the condition in brackets is not a logical necessity since if it is not true then the point is not even in the super state of *moving_st*, however, it is included so that the prover can discharge a related proof obligation that the function application, *working_sm(p)*, is well formed.

We were unable to prove this safety invariant and suspected the original UML model to be unsafe. Often when this is the case, examination of the proof obligation helps understanding the problem and suggests a correction in the model. However, in this case it is not obvious why the proof obligation is not provable and how the model can be fixed. The difficulty stems from the late arrival of the concept of routes and the consequent lack of abstract representations of the concepts involved in the safety invariant. In fact, the original UML model introduces the abstract concept of locking (albeit for different reasons) which we have discarded due to difficulty in translation. To improve our model we re-worked it from refinement level *m4* where *point* is introduced and at this stage we model the derived attribute *is_locked* as a boolean attribute of the *point* class. This allows us to express, at an earlier stage, the fact that points must not move while locked, even before specifying the real meaning of locked. At this stage we introduce some non-deterministic alteration of the locked attributes which are later refined into the behaviour of routes. We introduce an invariant into the *moving_st* state of the *point* class to express the constraint that the point should not be locked when it is moving. In doing so we prove an abstract equivalent of the safety invariant SR1, namely:

> SR1a: A point that is moving is not locked.

The invariant can be seen in the state *moving_st* of Fig 5. When translated to plain Event-B this state invariant becomes:

$$\texttt{SR1a}: \forall thisPoint \cdot ((thisPoint \in point) \Rightarrow ((point\_sm(thisPoint) = working\_st) \Rightarrow ((working\_sm(thisPoint) = moving\_st) \Rightarrow (locked(thisPoint) = FALSE))))$$

The invariant in the diagram is translated from dot notation to function application to obtain *locked(thisPoint) = FALSE*. The contextual position of the invariant in the sub-state *moving_st* of the state *working_st* of a state machine belonging to the class, *point*, gives rise to the chain of antecedents.

In attempting to prove this we realised that the model has unguarded transitions to *moving_st* which are triggered when a point is requested to move to the position it is already in (transitions *move_left_from_left* and *move_right_from_right*). Perhaps this was not considered by the UML modellers to be a genuine move for the purpose of safety but, if this is the case, it is not possible to (formally) distinguish safety in the given model meaning that we have been given an impossible task. To investigate further we corrected the model by adding guards to prevent these transitions when the point is locked allowing us to prove the abstract safety invariant SR1a.

Having established that locked points do not move, to complete the proof of the safety invariant, we need to prove that the points in an established route are always locked. (This is the gluing invariant corresponding to the data refinement that replaces *is_locked* with membership of an established route). That is:

GL1 : $\forall r, p \cdot r \in route \wedge route\_sm(r) = established\_st \wedge p \in (left\_points[\{r\}] \cup right\_points[\{r\}]) \Rightarrow locked(p) = TRUE$

Given the abstract invariant, SR1a and the gluing invariant, GL1, the original safety invariant, SR1, is discharged easily by the Rodin provers. In fact, since it follows directly from SR1a and GL1 we can make SR1 a theorem so that it only needs to be proved once.[2]

However, the gluing invariant still can not be proved for the transitions, *cancel_route* and *established_idle*, that reset locked to false (or to use the concrete representation, release an established route). Upon examination, we realise that the model does not prevent two conflicting routes (that share common points) from being established concurrently (so that a point is locked by two routes at the same time). In this case, the gluing invariant is violated when either one of the routes is cancelled and unlocks its points. The location of the mistake is in another derived attribute of the UML model, *proceed_conditions_ok*, which is defined as 'all *left_points* of the route are *at_left*, all *right_points* of the route are *at_right* and all *tracks* of the route are *free*'. Hence the proceed conditions defined in the source model prevent a route from being established when its points are not in the required positions but not if another route has already established and locked those points in the correct positions. Strictly speaking, the safety requirement we are working on does not specify anything about conflicting routes, it was an assumption we made in our gluing invariant. We could design an alternative, provable, gluing invariant which allows for conflicting routes, but since the micro2010 UML model also contains a use-case that indicates that a 'set route' request should not succeed if one of its points is locked by a route, we

[2] In Event-B, and hence UML-B, to prove an invariant it is necessary to prove that every event results in a state that satisfies the invariant whereas a theorem is deduced from the other invariants and theorems.

deduce that there is a mistake. To correct it we add a guard to the transition, *setting_up_established*, in the route state machine Fig. 7 of refinement level *m6*. The guard ensures that none of the points that are used in the route are already locked:

`Guard4 :` $locked[left\_points[\{self\}] \cup right\_points[\{self\}]] = \{FALSE\}$

The guard uses the contextual instance, *self*, of the class, *route* (i.e. *self* is the route being set up by the transition *setting_up_established*). The guard can be read as, the only value of *locked* for all the *left_points* and *right_points* of *self*, is *FALSE*.

Note that this is the same transition that locks the points used by the route. It is important that this is done in one (atomic) transition to a) ensure that the guard remains true when the locks are set and b) so that the derived attribute *is_locked* is refined by its definition.[3].

After adding this guard we are able to prove the gluing invariant for the two transitions that render a route un-established, i.e. *cancel_route* and *established_idle*. This completes the proof of the safety invariant SR1.

The statistics for proofs are given in the following table. The automatic provers are configurable so that the user can choose which provers to try and specify timeout values. These figures represent a configuration with a meta-prover (called *Relevance Filter*) enabled. With this meta prover enabled, before tackling the safety requirement in m8, only one PO (from m6) requires interactive proof. All of the POs relating to the abstract safety invariant in m4 are discharged automatically. In m8, three PO's required interactive proof. These are the proof of the intermediate state invariant when a route is established and the two (identical) proofs that the transitions that un-establish a route satisfy the gluing invariant.

Refinement	Proofs	Automatic	Interactive
m0	7	7	0
m1	11	11	0
m2	18	18	0
m3	15	15	0
m4	88	88	0
m5	34	34	0
m6	34	33	1
m7	26	26	0
m8	16	13	3

It is interesting to know where the effort lies in carrying out this verification work. The effort in actually carrying out an interactive proof is minor. Now that

[3] Although, for clarity, we have retained *is_locked*, we would like the option to remove it in a later refinement and our proof of SR1 relies on the data refinement where it is replaced by its definition

the model has been constructed and corrected, we could conduct the interactive proofs within an hour at most. Similarly, discovering and correcting errors in the model is not particularly time consuming because the proof obligations quickly lead one to the problem. However, the iterative process of attempting the proofs, running into difficulties and improving the modelling techniques to make the model more amenable to proof, took several weeks of effort.

6 Future Work

As part of our role in the INESS project we will continue to prove other safety requirements in the micro2010 model. We expect to find that similar techniques of introducing more abstraction are necessary and hope this will lead to a general method. Hence we will investigate in more detail how to methodically generate abstract concepts from UML derived attributes in a way that helps us prove safety invariants. We will continue this investigation using more extensive examples in order to test the generality of the method.

We are also interested in developing guidelines for constructing a UML-B model in refinements from a UML model that has none. For this purpose, the micro2010 model has given us some initial ideas but is somewhat limited. We plan to investigate these guidelines using some more extensive examples of interlocking models that are also available as part of the INESS project.

Another approach that we have previously investigated [2] is to start by modelling the most abstract model that is needed in order to demonstrate the safety requirements. It may then be possible to develop the full translated model as a refinement of the safety requirements, hence showing that the model is safe by construction. We would like to investigate this alternative approach in comparison to that presented here.

7 Conclusion

We have investigated a mechanistic approach to translation of a UML model into a series of UML-B refinements and the introduction of safety invariants at a final stage. We found that this approach has limitations because the refinement is based on the structure of the UML model resulting in a superposition approach to refinement (often called horizontal refinement). The safety invariant to be proved becomes too complicated to a) find a proof, and b) to find why it is not provable in the model. We deduce that the translation does not provide enough abstraction of concept (often called vertical refinement) and suggest that derived attributes of the UML source model may provide a clue to introducing this methodically through the translation. When we introduced a vertical refinement using the derived attribute *is_locked* we were able to find and rectify two problems in the model and subsequently proved the safety invariant with relative ease using the abstract version of it and the gluing invariant of the refinement.

References

1. INESS. http://www.iness.eu/ (2010)
2. Snook, C.: Specifying Safety Requirements for a Railway Interlocking System. Dagstuhl Seminar on Refinement based methods for the construction of dependable systems (2009)
3. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual. 2nd edn. Addison-Wesley Object Technology. Addison-Wesley Professional (July 2004)
4. Metayer, C., Abrial, J.R., Voisin, L.: Event-B Language. Rodin deliverable 3.2, EU Project IST-511599 -RODIN (May 2005)
5. Abrial, J.R.: Modelling in Event-B: System and Software Engineering. Cambridge University Press (2010)
6. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. International Journal on Software Tools for Technology Transfer (STTT) **12**(6) (2010) 447–466
7. Abrial, J.R., Butler, M., Hallerstede, S., Voisin, L.: An Open Extensible Tool Environment for Event-B. In: ICFEM 2006, LNCS, Springer (2006) 588–605
8. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. **15**(1) (2006) 92–122
9. Snook, C., Butler, M.: UML-B and Event-B: an integration of languages and tools. In: The IASTED International Conference on Software Engineering - SE2008. (February 2008)
10. Said, M., Butler, M., Snook, C.: Language and Tool Support for Class and State Machine Refinement in UML-B. In Cavalcanti, A., Dams, D., eds.: FM 2009: Formal Methods. Volume 5850 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2009) 579–595
11. Savicks, V., Snook, C., Butler, M.: Animation of UML-B Statemachines. Technical Report (http://eprints.ecs.soton.ac.uk/18261/1/TBFMsmAnim.pdf) and presented at Rodin User and Developer Workshop (2010)
12. Leuschel, M., Butler, M.: ProB: A model checker for B. In Araki, K., Gnesi, S., Mandrioli, D., eds.: FME 2003: Formal Methods. LNCS 2805, Springer-Verlag (2003) 855–874
13. Schacher, M.: Micro interlocking 2010. Know Gravity Inc. http://knowgravity.com (2010)
14. ArtisanStudio. http://www.atego.com/products/artisan-studio/ (2010)
15. Mellor, S., Balcer, M.: Executable UML: A foundation for model-driven architecture. Addison Wesley (2002)
16. Cassandra. http://www.knowgravity.com/eng/value/cassandra.htm (2010)