Refinement of statemachines using hierarchical states, choice points and joins

Colin Snook¹ and Marina Waldén²

¹University of Southampton, Southampton, SO17 1BJ, UK
²Åbo Akademi University, Lemminkäinen 14A, 20520 Turku, Finland.

Abstract. While refinement gives a formal underpinning to the development of dependable control systems, such models are difficult to communicate and reason about in a non-formal sense, particularly for validation by non-specialist industrial partners. Here we present a visualisation of event B refinement using a specialisation of UML statemachines. The specialisation will be incorporated into the UML-B notation that is being re-developed within the Rodin project to improve integration with the Event B platform, which is also being developed by the Rodin project.

1 Introduction

Formal software construction techniques are beneficial when developing complex distributed control systems. Such systems often demand high integrity to achieve safety requirements. The use of formal analysis tools can increase confidence in the correctness of the system. These tools are, however, not always easy to use, or well accepted, in an industrial environment. This barrier can be overcome by a graphical presentation of the formal models. In many cases, rules of the formal development can be built into the graphical notation and supported directly within the drawing tool.

We use the Event B [AbrMu01] formalism as our formal framework for developing distributed control systems. Event B is a method with tool support for applying distributed systems in the B Method [Abrial96] that is based on Action Systems [BaKu83, BaSe96] and related to B Action Systems [WaSe98]. Hence, we can benefit from the useful formalism for reasoning about distributed systems given by Action Systems and from the tool support in B. Development within Event B is performed in a stepwise manner from abstract specification to concrete implementation using superposition refinement. The correctness of each step is proved in order to achieve a reliable system. The tool assists the development process by generating the proof obligations needed. These proof obligations can then be proved with the automatic or the interactive prover of the tool.

An approach of integrating formal specification and verification in B, with the UML [BJR98] has, for some years, been developed at Southampton. The UML-B

* This work is part of the EU funded research project: IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).
[SOB04] is a profile of UML that defines a formal modelling notation combining UML and B. It is supported by the U2B tool [SnBu04], which translates UML-B models into B, for subsequent formal verification. The translation of UML-B statemachines is similar to that proposed by Sekerinski [Sek98]. A closer integration of UML-B and new Event B tools is being developed in the Rodin project [Rodin]. In this paper we describe part of this ongoing work to extend support for the refinement of event models expressed as UML-B statemachines. This work is based on previous work to develop control systems using UML and B [SnTsWa03]. We show how we use hierarchical states to model the addition of more detail to the state space and corresponding events. We utilise fork and join pseudostates to visualise the required event refinement relationships and choice pseudostates to visualise how events have been decomposed into subevents.

To illustrate these techniques we use a simplified version of a case study that is derived from previous work [Matisse]. We use a part of a microplate liquid handling workstation, Fillwell [PE01, BJW03], for discovering new drugs as our example. The Fillwell workstation consists of a dispense head, a gantry and a processing table with microplates. The dispense head dispenses liquid into and aspirates liquid from the plates on the processing table. The gantry moves the dispense head from one plate to another on the table. Both the dispensing, aspirating and moving have to be performed with very high precision.

2 Modelling Event B systems in UML

We depict the functional requirements of the system in class diagrams. For each class we give the attributes and their types, as well as the methods. Hence, the classes model the static properties of the components. The behaviour of each component of the system is specified with a statemachine diagram. The specification of the eventual system is then gradually captured and made more and more precise throughout a series of these diagrams. In the most abstract class diagram we consider the state value for the state machine and the command that the component is required to obey. The methods cover the functionality of the system and an abstract representation of the types of errors as well as the possibility to fix these errors. Here, we discuss the methods we use to refine statemachines.

The first abstract statemachine diagram of a control system is shown in Figure 1. The diagram models the evolving, failing and recovering of the controller. When the system is functioning successfully, evolve, and is given proper commands ($cmd = true$), it remains in state ok. The transition evolve is an abstraction of the behaviour of the system. If the system fails, it will go to state susp via transition fail. The controller recovers, recover, to state ok, if the fault is recoverable. If a non-recoverable fault occurs, terminate, the controller terminates (i.e. deadlocks) in state flt.
2.1 Creating a B-model from UML

In B, the variables of a system are given in the **VARIABLES**-clause. The types and the invariant properties of the variables are given in the **INVARIANT**-clause and their initial value in the **INITIALISATION**-clause. The events that alter the variables are given in the **EVENTS**-clause.

The first step in our formal development is to create an abstract Event B system from the abstract class diagrams and statemachines. The tool U2B [SnBu00] supports this translation. Attributes in class diagrams correspond to variables in B. Such variables are ‘lifted’ and modelled as functions so that the separate values for each instance can be represented. However, for simplicity, we do not show this lifting in the examples that follow. States in statemachines (including the top-level region representing the whole statemachine) may be represented by a variable that gives the currently selected substate. The state also represents an enumeration element within the type of its parent state (if any). Transitions in statemachines correspond to guarded branches within events of the class.

With Event B we model parallel and distributed systems, where events are selected for execution in a non-deterministic manner. The events are given in the form \( \text{Oper} = \text{WHEN} \ P \ \text{THEN} \ S \ \text{END} \), where \( P \) is a predicate on the variables (also called a guard) and \( S \) is a substitution statement. When \( P \) holds the event \( \text{Oper} \) is said to be enabled. Only enabled events are considered for execution. When there are no enabled events the system terminates. We note that control systems are designed only to terminate when an unrecoverable failure occurs. The events are considered to be atomic, and hence, only their input-output behaviour is of interest.

The statemachine of the controller in Figure 1 can be translated to the Event B system below. Here, the variable \( \text{cmd} \) models the command of the controller. The state is initialised to value \( \text{ok} \). New commands are provided by the environment (user) via event \( \text{new\_command} \), which can occur at any time. Event \( \text{evolve} \) is enabled when a new command is received. The command is deactivated (\( \text{cmd} := \text{false} \)) during execution, \( \text{evolve} \). Instead of executing the new command the system may fail, which is modelled by event \( \text{fail} \). When the system can recover, \( \text{recover} \), from the failure, it returns to state \( \text{ok} \). If the failure is unrecoverable the system terminates, \( \text{terminate} \) and is said to be deadlocked.
3 Refining models

An important feature provided by the Event B formalism is the possibility to stepwise refine specifications. The refinement is a process transforming a specification $A$ into a system $C$ when $A$ is abstract and non-deterministic and $C$ is more concrete, more deterministic and preserves the functionality of $A$. We use a particular refinement method, where we add new functionality, i.e., new variables and substitutions on these, to a specification in a way that preserves the old behaviour. This type of refinement is referred to as superposition refinement [BaKu83, Katz93]. When dealing with complex control systems it is especially convenient to stepwise introduce details about the system to the specification and not to have to handle all the concrete implementation issues at once. In the refinement process we identify the attributes suggested in the requirements specification. We add the computation concerning the new variable to the existing events by strengthening their guards and adding new substitutions on the variables. New events that only assign the new variables may also be introduced.

In our example we use this refinement process to add details about the controller’s behaviour. As the system development proceeds we add more elaborated information about faults and conditions of failure occurrence [PTWBEJ01, Tro00].

3.1 Proving correctness of the refinement

In order to gain confidence in the refinement process we need to prove the correctness of each refinement step. With the mechanical tool Atelier B that supports the B Method and Event B (previously via Evt2B translation) we can prove formally that the refinement is sound. For this a number of proof obligations first need to be generated. The proof obligations needed for proving that a concrete system $C$ is a correct refinement of a more abstract system $A$ are as follows [BaSe96, WaSe98]:

```plaintext
MODEL Controller
VARIABLES
    cmd, state
INVARIANT
    cmd ∈ BOOL ∧ state ∈ {ok, susp, flt}
INITIALISATION
    cmd := BOOL | | state := ok
EVENTS
    new_command(c) == PRE c ∈ BOOL THEN cmd := c END;
    evolve == WHEN state = ok ∧ cmd=true THEN cmd := false END;
    fail == WHEN state = ok ∧ cmd=true THEN state := susp END;
    recover == WHEN state = susp THEN state := ok END;
    terminate == WHEN state = susp THEN state := flt END
END
```
1. The initialisation in $C$ should be a refinement of the initialisation in $A$, and it should establish the invariant of $C$.

2. Each old event in $C$ should refine the corresponding event in $A$, and it should preserve the invariant of $C$. (If the event is renamed in $C$, the fact that it is intended to be a refinement of the old event must be stated explicitly).

3. Each new event in $C$ (that does not have a corresponding event in $A$) should only concern the new variables, and preserve the invariant.

4. The new events in $C$ should terminate, if they are executed in isolation. No livelock should occur.

5. Whenever an event in $A$ is enabled, either the corresponding event in $C$ or one of the new events in $C$ should be enabled. Relative deadlockfreeness.

6. Whenever an error occurs in $C$ (an error detection event in $C$ is enabled), an error should also be possible in $A$ (an error detection event in $A$ should be enabled).

With the error detection events in (6) we mean the events leading to state $sus$. Hence, according to (6) an abstract representation of an error type is partitioned into distinct concrete errors during the refinement process.

The proof obligations (1)–(3) above are automatically generated by Atelier B. Proof obligation (4) requires a variant that is decreased by the new events. Proof obligations (4) – (6) can be generated after introducing some additional constructs discussed in [WaSe98, BuWa96].

The proof obligations can be discharged with the help of the auto-prover and interactive prover in Atelier B. The proof obligations that cannot be proved automatically can be proved interactively with the interactive prover. By discharging all these proof obligations for each refinement step in the control system development we have proved the correctness of the system with respect to its specification. Since we incorporate failure management in the development process, we ensure that the controller maintains safety not only in absence of faults, but also by being able to withstand faulty behaviour.

4 Graphical interface for refinements

In order to get a graphical interface to the formal development process, the development is performed via UML artefacts. New features are introduced in a stepwise manner into the class diagrams and statemachines. As a starting point we ‘clone’ the current model to obtain a copy to be refined. The refinement process is further facilitated by only allowing events, which guarantee that the refinement rules for Event B (and B Action Systems) are applied [AbMu01, WaSe98]. The new features are modelled with new attributes. The transitions (events) in the corresponding statemachines are modified to take into account these new attributes. The more concrete behaviour of the system can be modelled with hierarchical states by adding sub-states and more complex transitions in the statemachines. Furthermore, new events may be added between the sub-states. When refining the class and statemachines in UML we consider two kinds of refinement: data refinement and event refinement.
4.1 Data refinement

When refining a system we model the behaviour of the system in a more detailed manner. Before adding more detailed behaviour we need to reveal more detailed state space. This can be done using hierarchical states to introduce sub-states within a state. For example, in Figure 1 we model abstractly with state `ok` that the controller is working, while in Figure 2 state `ok` is split into sub-states to model that when the controller is working it can be pending (`idle`), ready (`rdy`) or running (`run`).

![Figure 2. The controller refined by adding hierarchical substates.](image)

A new state variable, `ok_state`, of type `{idle, rdy, run}` will be generated in B to model the substates. Note that `ok_state` retains its value when `state` is not equal to `ok` even though it then has no meaning in terms of the current state of the system. This corresponds to the UML notion of states having a memory (history) that can be returned to. (We have shown this step separately for illustration, normally we would combine it with the next step where new events make use of the new states).

The hierarchy of states is usually given in a refined statemachine diagram by elaborating a state with substates. However, after several consequent splittings into substates, it could be preferable to flatten the hierarchy (i.e. remove old superstates) in order to reduce the complexity of the diagram.

4.2 Event refinement - adding new transitions to use refined states

The more detailed behaviour that is revealed by the hierarchical states is reflected by refining the old events and adding new transitions (new events) between the substates. The old events may be renamed during the refinement to better describe the scenario. The guards of the old events may be strengthened and assignments concerning the new feature added in line with the Proof Obligations (1) and (2) in Section 3. The new events are only allowed to assign the new variables, but may refer to the old variables as stated in Proof Obligation (3).
Figure 3. The controller refined by adding new transitions

In our controller example the abstract event, evolve, is replaced by a sequence of events config; start; stop. Since it is stop that leaves the system in the equivalent state to evolve (ready to start the sequence again), stop refines evolve. This is specified using a fork pseudostate as shown in Figure 3 indicating the old, refined event in angled brackets. The two events leading up to the refining event, config and start, are new events. Note that we do not want the sequence of events to start unless it can complete so we put the guard of evolve on the first event, config. This is aimed at fulfilling Proof Obligation (5). The guard of stop must not be weaker than that of evolve (otherwise stop would not be a refinement since it would permit new traces). This is the case in our refinement because the only possible path of transitions to the state run starts with config. To reinforce this (and assist the proof) we attach the invariant cmd=true to the states rdy and run. Note that the U2B translator automatically adds the premises ok_state=rdy and ok_state=run.

According to the refinement rules (Proof Obligation (4)) these new events should not take over the execution. This can be guaranteed by disallowing the new transitions from forming a loop in the state machine diagram. This could be checked automatically via graph theory. For proof in B, a variant could be generated by numbering the states in the diagram with the minimum path length to a refining transition, as well as taking the state variables into account. Each new transition between the substates should decrease the variant, i.e., lead to a new state with a lower number or decrease the variant based on the state variables. For relative deadlock freeness, it is a necessary but insufficient condition for there to be a path of transitions from each new transition to one that refines an old transition.

As features are added to the system the failure management should also be refined. If a fault occurs at a substate it should be possible to return to that substate after recovery. Note that we are not introducing new failure situations, but only splitting up the current non-deterministic failures (according to Proof Obligation (6)).
In Figure 4, the failure management of the controller is refined. This is taken into account by splitting the suspended state \( \text{susp} \) into substates corresponding to the substates of state \( \text{ok} \). From each substate of \( \text{ok} \) a failure event (\( \text{config}_\text{fail} \), \( \text{start}_\text{fail} \) and \( \text{stop}_\text{fail} \)) takes the controller to the corresponding faulty state. Also the recovery event \( \text{recover} \) should be refined so that the controller returns after recovery to the state where the failure was detected. Notice that the recovery event utilises the history that is retained by the substates. That is, the substate that was current when \( \text{ok} \) was left, is reinstated. Since the recovery events may contain different actions depending on the \( \text{susp} \) substates, we refine recover to separate recovery events from each of these substates.

When states are split into substates to give a more precise description of the system, it is also justified to split the event/transition between the split states. When adding these new events to the statemachine diagram we use the symbols \textit{join} and \textit{fork}. An example of using joins and forks is shown in Figure 4 where event \( \text{fail} \) is refined by the events \( \text{config}_\text{fail} \), \( \text{start}_\text{fail} \) and \( \text{stop}_\text{fail} \). In this way we specify the event refinement relationship between new and old events. The refined controller in Figure 4 can be translated to the refined B machine \textit{Controller}$_\text{Ref}$ below.
During the refinement steps described above, it is usual to also add more details about the operation of the system by adding guards that describe when each transition event occurs and hence reduce the non-determinism between alternative transition events such as operation or failure. At the same time, assignment substitutions are added to the transition events to modify the variables controlling these guards. For clarity we have omitted such detail.

4.3 Event refinement – separating existing transitions

During the refinement process new features in the form of new variables are added to the system in each refinement step. The events are refined to take the new features into account. This is performed by strengthening the guards of the events. New assignments may also be added to the bodies of the events concerning the new features. Furthermore, several events may refine one old event and an event may refine several old events (Proof Obligation (2)).
In statemachine diagrams the refinement of an event/transition by splitting it into several events with stronger guards, can be illustrated by adding choice points on the transitions. When adding conditions concerning the new features to the guards we also refine the failure management. We divide an abstract failure into more specific failures on the new features according to Proof Obligation (6). Here we have the refinement case that many events refine one event. This is shown in the example in Figures 5 and 6 where the abstract system is given in Figure 5 and its refinement in Figure 6.

We illustrate the event refinement on transitions with an example of the Fillwell system where a dispense head dispenses and aspirates liquid into an accessory [PE01, BJW03]. Here we focus on the event to aspirate liquid shown in the statemachine diagram in Figure 5. The system is prepared to aspirate when it is in state prep. Furthermore, the guard \( gd \) (representing the command to aspirate) has to hold for the event to be enabled. This either results in a success event aspLiq that takes the system to state aspl or in a failure event aspLiqFail that suspends the system. At this stage no details are given as to how this choice is made.

![Figure 5. The abstract event to aspirate liquid.](image)

The transitions in the statemachine diagram in Figure 5 translate to the following events in B.

```
EVENTS
...  
aspLiq ==
  WHEN state = prep \( \land \) gd THEN state := aspl END;
aspLiqFail ==
  WHEN state = prep \( \land \) gd THEN state := susp_prep END;
...
```

In the first refinement step of the event to aspirate liquid we introduce choice points to visualise event refinement by strengthening of guards and to show common parts of the guards as in Figure 6. The failure event aspLiqFail is split up into four different failures; noAccFail modelling failure when there is no accessory to aspirate into, noLiqFail modelling failure when there is no liquid in the accessory and aspLiqFail1 representing remaining undetermined failures. The guards \( na \) and \( nl \) represent each failure condition. We have chosen to prioritise failures via a sequence of choice

```
EVENTS
...  
aspLiq ==
  WHEN state = prep \( \land \) gd THEN state := aspl END;
aspLiqFail ==
  WHEN state = prep \( \land \) gd THEN state := susp_prep END;
aspLiqFail1 ==
  WHEN state = prep \( \land \) gd \( \land \) noAccFail THEN state := susp_prep END;
aspLiqFail2 ==
  WHEN state = prep \( \land \) gd \( \land \) noLiqFail THEN state := susp_prep END;
aspLiqFail3 ==
  WHEN state = prep \( \land \) gd \( \land \) aspLiqFail1 THEN state := susp_prep END;
aspLiqFail4 ==
  WHEN state = prep \( \land \) gd \( \land \) aspLiqFail2 THEN state := susp_prep END;
...
```
points. If both na and nl are true, the noAccFail failure will occur since noLiqFail is guarded by ¬na. Note that we do not yet know whether the undetermined failures take priority over the ones we have specified. Hence we do not add the negated failure guards to aspLiqFail1. The symbol join is used to show that the new failure events refine the abstract event aspLiqFail. The guard for event aspLiq includes the conjunction of the negation of all the specific failures.

The second refinement step in Figure 7 shows further decomposition of the undetermined failures in aspLiqFail1. For the accessory failure we introduce event wrongAcc that suspends the system due to the wrong type of accessory being available. We also add a new feature that we can detect a failure if there is not enough liquid to aspirate lowLiq. In this refinement we have chosen not to prioritise between
noLiqFail and lowLiqFail, either may be chosen if both failures are present. Similarly, no priority is defined between noAcc and wrongAcc. This non-determinism may be used when there is functional equivalence between the kinds of failures. The undetermined failure is again still represented in aspLiqFail2.

The transitions in the statemachine diagram in Figure 7 of the second refinement step of the operation to aspirate liquid can be translated to the following B code.

```plaintext
EVENTS
...
aspLiq ==
  WHEN state = prep ∧ gd ∧ ¬(na ∨ wa) ∧ ¬(nl ∨ ll)
  THEN state := aspl END;
aspLiqFail2 [refines aspLiqFail1] ==
  WHEN state = prep ∧ gd THEN state := susp_prep END;
noAccFail ==
  WHEN state = prep ∧ gd ∧ na
  THEN state := susp_prep END;
wrongAccFail [refines aspLiqFail1] ==
  WHEN state = prep ∧ gd ∧ wa
  THEN state := susp_prep END;
nolLiqFail ==
  WHEN state = prep ∧ gd ∧ ¬(na ∨ wa) ∧ nl
  THEN state := susp_prep END;
lowLiqFail [refines aspLiqFail1] ==
  WHEN state = prep ∧ gd ∧ ¬(na ∨ wa) ∧ ll
  THEN state := susp_prep END;
...
```
In subsequent refinements we only show the latest event/transition refinement step in the statemachine diagram. For example, in Figure 7, we show that AspLiqFail2 refines AspLiqFail1 but omit that AspLiqFail1 refines AspLiqFail. This agrees with the corresponding B specification where only the last event refined needs to be indicated. When adding a new feature we usually also add assignment substitutions to the events and give more detailed recovery functions. Here we have focused on the guards of the events for clarity.

5 Conclusions

We have illustrated a specialisation of the UML statemachine notation that enables it to be used to visualise event models and their refinements including specification of the refinement relationship between events of the abstract and concrete models. We utilise hierarchical substates to provide the additional state space needed for defining new event transitions and pseudostates for adding secondary information to the model such as the prioritisation of newly determined events. We have found such techniques to be very useful in communicating models with colleagues and especially with industrial partners who have less experience with formal notations. We envisage improving the existing tool support for refinement in UML-B as part of the Rodin project.

References


[Rodin] Rigorous Open Development Environment for Complex Systems (RODIN) - IST 511599 - http://rodin.cs.ncl.ac.uk/


