Supporting Reuse in Event B Development: Modularisation Approach

Alexei Iliasov[1], Elena Troubitsyna[2], Linas Laibinis[2], Alexander Romanovsky[1], Kimmo Varpaaniemi[3], Dubravka Ilic[3], and Timo Latvala[3]

[1] Newcastle University, UK
[2] Åbo Akademi University, Finland
[3] Space Systems Finland
{alexei.iliasov, alexander.romanovsky}@ncl.ac.uk
{linas.laibinis, elena.troubitsyna}@abo.fi
Dubravka.Ilic@ssf.fi, Timo.Latvala@ssf.fi, Kimmo.Varpaaniemi@ssf.fi

Abstract. Recently, Space Systems Finland has undertaken formal Event B development of a part of on-board software for the BepiColombo space mission. As a result, lack of modularization mechanisms in Event B has been identified as a serious obstacle to scalability. One of the main benefits of modularization is that it allows us to decompose system models into components that can be independently developed. It also helps to manage complexity of models that in the industrial setting are usually very large and difficult to comprehend. On the other hand, modularization enables reuse of formally developed components in the formal product line development. In this paper we propose a conservative extension of Event B formalism to support modularization. We demonstrate how our approach can support reuse in the formal development in the space domain.

1 Introduction

In the Deploy project[8], Space Systems Finland has performed a pilot Event B development[11] of a part of on-board software for the BepiColombo space mission [5]. The developed system is responsible for controlling and monitoring instruments that produce valuable scientific data that are critical for the success of the mission. The undertaken development aimed at identifying the strengths and weaknesses of Event B method and its supporting tool – the RODIN platform[14]. The experience demonstrated that the refinement approach provides a suitable design technique. It allows us to structure complex and numerous requirements and promotes disciplined development via abstraction and proofs. However, it has also became obvious that the lack of modularization makes Event B unscalable for formal development of industrial systems. In this paper we propose a conservative extension of Event B language that supports a simple modularization idea.

The idea of modules is very well known and is supported by most of the formal frameworks. Usually they define a module interface via pre- and postconditions. However, in our case introducing preconditioned operations in Event B was unacceptable due to two main reasons. Firstly, preconditioned operations would not be supported by the RODIN platform and building a new tool of similar strength would require significant time and financial investments. Secondly, introduction of a preconditioned operation would seriously complicate the proof obligations required to verify correctness and hence would lower the degree of automation in the development. Therefore, our approach is strictly driven by the pragmatic needs and oriented towards automation.

In this paper we briefly describe the on-board software that have been modelled and present the experience gained by Space Systems Finland. Then we describe our proposal for introducing modularization in Event B and demonstrate how the system can be redeveloped in a modular fashion.

We believe that by enabling modular development in Event B we not only improve scalability of formal modelling but also potentially increase productivity. Indeed, formally developed components can be reused in other developments and hence amplify the effect of formal modelling.

2 Challenges and Experiences in Formal Development of Onboard Software

2.1 Example of Onboard Software

Spacecraft-embedded software – onboard software – is responsible for managing various spacecraft operations. For instance, the controlling software is critical to the mere survivability of a mission, while scientific software is responsible for correct and effective handling of high volume of data generated by extensive scientific experiments. Therefore, failure of onboard software can have major repercussions. Yet, onboard software must withstand extreme conditions of the space environment and operate with hardware, which has limited capabilities compared to personal computers. It is clear that these factors make the design, implementation and verification of onboard software very challenging.

Space Systems Finland is one of software providers for the European Space Agency (ESA) mission BepiColombo. The main goal of the mission is exploration of the planet Mercury. The mission comprises various scientific studies, e.g., analysis of its internal structure and a surface, investigation of the geological evolution of the planet etc. To achieve the defined scientific goals, one of the mission orbiters – Mercury Planetary Orbiter – will carry remote sensing and radioscience instrumentation. Space Systems Finland is responsible for developing software for an important part of the orbiter – the data processing unit. The company has undertaken formal development[11] of it in the Event B framework with the support of the RODIN platform[14].

The data processing unit (DPU) is used to control two scientific instruments: Solar Intensity X-ray and particle Spectrometer (SIXS) that records the radiation from the Sun at the position of the spacecraft, and Mercury Imaging X-ray Spectrometer (MIXS) that records fluorescent X-rays from the planet surface. In turn, both instruments contain two separate sensor units: X-ray spectrometer (SIXS-X) and particle spectrometer (SIXS-P) for SIXS, and telescope (MIXS-T) and collimator (MIXS-C) for MIXS.

The DPU unit is communicating with the BepiColombo spacecraft via SpaceWire interfaces, which are used to receive telecommands from the spacecraft and transmit science and housekeeping telemetry data back to the spacecraft.

The system under construction consists of three main software components: the Core Software (CSW), the SIXS instrument application software (SIXS ASW) and the MIXS instrument application software (MIXS ASW). CSW is the common interface software for the MIXS ASW and SIXS ASW. It controls and monitors the operating states of SIXS and MIXS instruments, as well as handles telecommand/telemetry communication with the BepiColombo platform.

In general, the behaviour of the system consists of receiving telecommands (TC) from the BepiColombo platform and producing corresponding telemetry data (TM). The received TCs are stored in a memory buffer. CSW is responsible for validation of syntactical

and semantical integrity of each received TC. In particular, it checks that each TC adheres to the PUS standard[**?**] describing telemetry and telecommand packet utilization. If validation fails then the corresponding TM is generated. Otherwise a TC is placed in the pool of TCs waiting for execution. Each TC has a "recipient" – the component that will actually execute TC.

There are several types of TCs. They might change the operation mode of the component, request to produce housekeeping report or generate scientific date etc. The component that executes TC always acknowledges TC execution by generating the corresponding TM. Besides an acknowledgment TM, a reaction on a TC might also include a TM containing progress reports, housekeeping reports or periodically generated scientific data.

Above we have given a very brief, high-level overview of system functionality. The actual detailed requirements for the DPU unit are rather complex and large (the real requirements document contains about several hundreds of pages), so we omit their detailed description here. Next we outline the steps of the formal development aimed at modelling the functional behaviour of the system.

2.2 Experiences in Formal Modelling

The formal development of the DPU unit started from an abstract specification that models the general control flow, abstractly representing a sequence of TC handling and TM generation steps. The first refinement step introduces explicit stages of TC and TM processing. Depending on the stage, a TC or a TM is assigned a specific status. For example, the TC status can be *Unchecked* (before validation), *Accepted* or *Reject* (after validation), *Waiting for Execution* (before execution), *Succesful Execution* or *Execution Failed* (after execution), and *Removable* (TC processing is finished).

The second refinement step elaborates on the structure of TCs and TM, introducing the notion of TC and TM types. We introduce a number of concrete types of TCs and TM, though many types are still modelled abstractly. The third refinement step focuses on introducing software processes, representing software components in the model. The representation of TC and TM is extended to explicitly model the target component that should execute a TC or the source component that produced a TM.

The fourth refinement step introduces the notion of the component operating modes and mode transitions. For instance, the Core Software CSW can be in *Operational*, *Standby*, and *Safe* modes. The fifth refinement step focuses on modelling generation of reports – the dedicated TMs confirming validation and execution of the corresponding TCs.

Certain types of TCs require not only reporting TMs but also TMs informing about progress of the TC execution, an operating mode change, or failure detection. Such progress reporting is introduced in the sixth refinement step. Furthermore, this refinement step introduces some details modelling the behaviour of one of the components – SIXS-X. The seventh refinement step models the behaviour of the other instruments in the similar way. Besides it also elaborates on component-specific TM generation and internal component behaviour.

We verified correctness of the entire refinement chain by proofs in the RODIN platfrom. The resultant specification has 20 variables, 61 events, 38 invariants. Additionally, the static data structures (15 sets, 88 constants) are defined by formulating 207 axioms and 20 theorems. The text of the specification (apart from definition of the data structures) has more than 40 pages.

The formal modelling of data processing unit described above has highlighted the following problems in Event B development

– It is not clear how to reuse the conducted development in the similar projects;

– Lack of modularization support hinders independent development of several subsystems;

– Without decomposition(modularization), a specification of even a relatively simple realistic system becomes very large and difficult to comprehend.

Therefore, there is a clear need to support modularization mechanisms in formal Event B development. Next we discuss our proposal for alleviating these problems.

Complexity of onboard software is constantly increasing, thus software for a space mission is usually partitioned into components that developed by different providers. In the space sector, cooperation between the providers and quality assurance is facilitated by two general mechanisms – the standards and, more recently, the reference architecture. Some of the available standards regulate the development process in general. Others define the interfaces (the format of data and data flows) between components, e.g., the PUS standard mentioned above. To facilitate the development in the sector, the reference architecture aims at providing a proven template solution for an architecture for the space domain. It lists typical functions of a space mission and interfaces between the functional blocks. A simplified version of a reference architecture is given in Fig. 1.

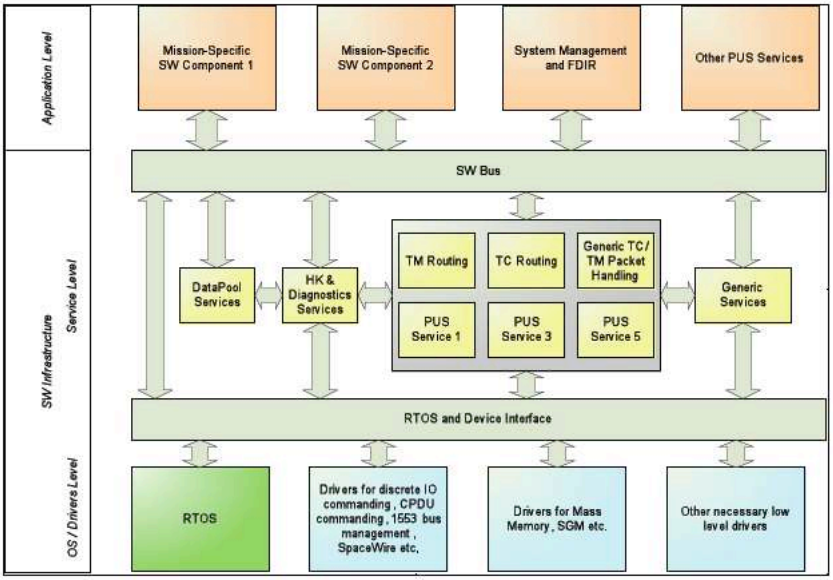


Fig. 1. Reference architecture

The reference architecture provides us with a suitable basis for identifying generic components. Since the communication between the components is regulated by the standards, modules can be abstractly defined by their interfaces. The development of components should ensure that the implementation preserves the given interface. Hence we can formally specify a system on architectural level, formally define the conditions imposed on

the component interfaces and then develop individual components while preserving their interfaces. Furthermore, we can reuse the models of previously developed components by composing them using their interfaces. Such an approach alleviates a problem of verifying large composed specifications.

3 Event B

In this section we introduce our formal framework – The B Method [1]. It is an approach for the industrial development of highly dependable software. The method has been successfully used in the development of several complex real-life applications [10]. Recently the B method has been extended by the Event B framework [2], which enables modelling of event-based (reactive) systems. In fact, this extension has incorporated the action system formalism [3, 4] in the B Method.

The B Method development starts from creating a formal system specification. The basic idea underlying stepwise development in B is to design the system implementation gradually, by a number of correctness preserving steps called *refinements*.

A simple B specification has the following general form:

MACHINE AM
SEES Context
VARIABLES v
INVARIANT Inv
INITIALISATION Init
EVENTS
$E_1 = \ldots$
$\ldots$
$E_N = \ldots$
END

A B specification, called an *abstract machine*, encapsulates a local state (program variables) and provides operations on the state. In the Event B framework, such operations are called *events*. The events can be defined as

WHEN g **THEN** S **END**

or, in case of a parameterised event, as

ANY vl **WHERE** g **THEN** S **END**

where vl is a list of new local variables (parameters), g is a state predicate, and S is a B statement (assignment) describing how the program state is affected by the event. Both ordinary and non-deterministic assignments can be used to specify state change. The non-deterministic assignments are of the form:

$$v :| \mathsf{Post}(v, v')$$

where Post is the postcondition or the next state predicate, relating the variable values before and after the assignment.

The events describe system reactions when the given **WHEN** or **WHERE** conditions are satisfied. The **INVARIANT** clause contains the properties of the system (expressed as predicates on the program state) that should be preserved during system execution. The data structures needed for specification of the system are defined in a separate component called *context*.

4 Introduction to Modules in Event B

Our primary goal is to conservatively extend the Event-B language with a possibility of (atomic) operation calls. Such an extension would naturally lead to the notion of modules – components containing groups of callable operations. Moreover, modules can have their own (external and internal) state and the invariant expressing properties on this state. The important characteristic of modules is that they can be developed separately and then composed with the main system during its formal development. Since we are interested in incorporating modules into Event B modelling, it should be also possible to statically check the correctness of such a composition within the Event B framework.

Let us start with an "ideal" (somewhat extreme) example of a general Event B operation that we would like to be able to express in our formal language.

op $=$
WHEN
Prec(v_1,...,v_N)
THEN
$v_1 :|$ ... op_1_call(parameters$_1$) ...
...
$v_N :|$ ... op_N_call(parameters$_N$) ...
op_{N+1}_call(parameters$_{N+1}$)
...
op_{N+K}_call(parameters$_{N+K}$)
END

Here op_i_call$(...)$ are either function or procedure calls from available modules[4]. A procedure call can be considered as special case of a function call (with the pre-defined return values). Thus from now on we will focus only on modelling function calls in Event-B.

Once an enabled event is chosen for execution in Event B model, all its actions are executed atomically and in parallel. However, the standard semantics of a function call, realised in most programming and formal languages, prescribes the well-defined order of execution steps:

1. Actual parameter expressions are evaluated and passed to a module operation;
2. The operation is executed on the given parameters and the module state. The operation result is returned to the calling operation;
3. The actions of the calling operation are executed, substituting the function calls with the returned results.

Moreover, the atomicity of an event operation with function calls should be preserved – no other event operation of the main system can intervene in between. Our challenge in this paper is to implement this standard functionality within the Event B semantics.

We split our task into two separate issues. First, we show how we can introduce modules and module calls during Event B development using model decomposition. Next, we

[4] Since all actions in the operation body should be executed in parallel, to avoid writing conflicts, we assume here that all function and procedure calls are from different modules

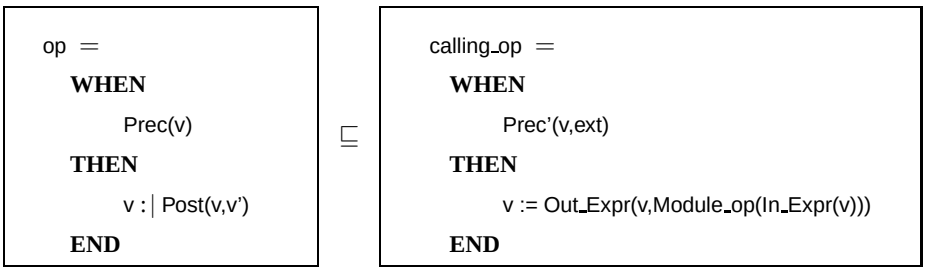
assume availability of pre-defined modules and demonstrate correctness of our specification containing module operation calls. The latter is a special case of verifying model composition.

4.1 Introducing Modules via Model Decomposition

In this paper we use the J.-R. Abrial's approach on Event-B decomposition[13]. The approach allows to split an Event B specification into several components (sub-models) that can be developed separately. If needed, some of these components can be further decomposed. Most importantly, the approach formally guarantees that the final re-composed system will be a refinement of the original one.

The decomposition is based on partitioning the model operations among the new components. The model variables are distributed as well, either as *internal variables* belonging to some particular components, or as *shared variables* that can be accessed by several components. To make the components self-contained, each of them is complemented by special *external events*, abstractly modelling how the shared variables may be modified by other components. The approach also restricts data refinement of the shared variables to make a decomposed system consistent. Essentially, the shared variables between two components of a decomposed system can be often seen as the input and output channels allowing these components to synchronise their activities.

Let us start with a simple generic example of an Event B operation. We would like to refine it so that it delegates (part of) its functionality to an external operation and then uses the returned result. In other words, the operation refinement should be of the form:

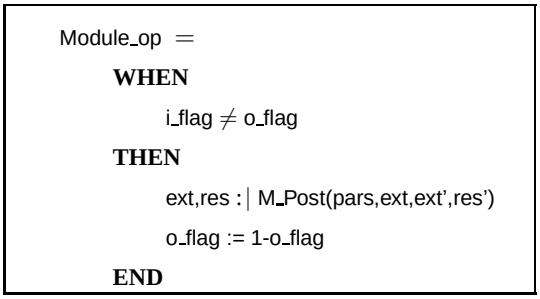


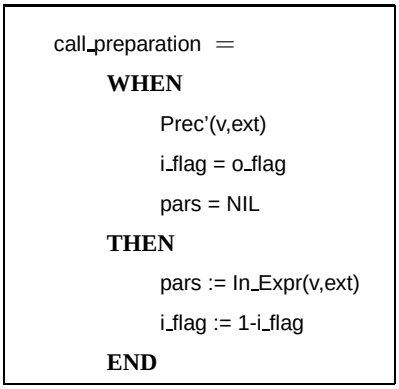
where Post is the postcondition of the original event, $\mathsf{In\_Expr}(v)$ is the actual parameter expression, $\mathsf{Out\_Expr}(...)$ is a state expression incorporating the result of the operation call, and *ext* is the externally visible part of the module state.

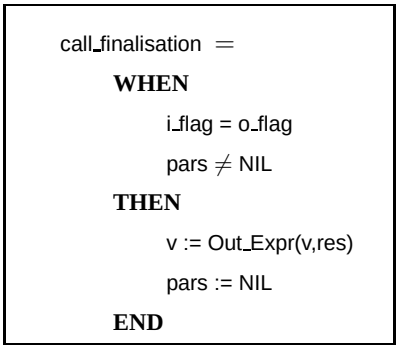
We interpret the refined operation as a syntactic sugaring hiding the actual definition in terms of the current Event B language. The idea is to model a function call by three events, simulating the three-step execution described above. Moreover, these three events should be introduced in such a way that we could decompose the system by distributing the system state and operations between the calling and called components.

The execution of a called module operation is abstractly modelled by Module_op presented below. Note that, in addition to calculating the result res, an operation call can also update the module state ext. The execution of a module operation is wrapped by two events of the calling component: call_preparation, which passes parameters to a module, and call_finalisation, which incorporates the returned results.

The variables i_flag and o_flag (of the type 0..1) are used to enforce the fixed order of execution between the main component and a module: first call_preparation, then Module_op, and finally call_finalisation. In addition, to guarantee atomicity of an operation







call, all the other operations of the calling component should be blocked until call_finalisation finishes. It can be achieved by strengthening their guards by $(\mathsf{i\_flag} = \mathsf{o\_flag}) \wedge (\mathsf{pars} = \mathsf{NIL})$. Essentially, the above solution is a special case of the alternating bit protocol.

This refinement step also achieves partitioning the state and operations between components. The variables res, $\mathsf{o\_flag}$ can be put into the future module component, while $\mathsf{pars}, \mathsf{i\_flag}, \mathsf{v}$ belong to the main specification. Following the Abrial's approach, we can decompose the system by moving Module_op into a separate module, where it can be developed (refined) independently.

To prove operation refinement, we need to show the connection between and the abstract operation Prec and the strengthened precondition Prec', as well as the expected postcondition Post in the main specification and the postcondition M_Post of the module operation. Specifically, the following two theorems should be proved as additional proof obligations:

$$\forall \mathsf{ext}.\mathsf{Prec}'(\mathsf{v},\mathsf{ext}) \wedge \mathsf{M\_Inv}(\mathsf{ext}) \;\Rightarrow\; \mathsf{Prec}(\mathsf{v})$$

$$\forall(\mathsf{v},\mathsf{ext},\mathsf{ext}'\mathsf{res}).\mathsf{M\_Post}(\mathsf{In\_Expr}(\mathsf{pars},\mathsf{ext}),\mathsf{ext},\mathsf{ext}',\mathsf{res}) \wedge \mathsf{M\_Inv}(\mathsf{ext}) \;\Rightarrow$$
$$\mathsf{Post}(\mathsf{v},\mathsf{Out\_Expr}(\mathsf{v},\mathsf{res}))$$

where M_Inv is the module invariant on its external state.

4.2 System Development via Model Composition

In the previous section we showed how we can delegate a part of functionality of the main specification to a module by means of model decomposition. In practice, however, we are more interested in the opposite – composing our systems using a collection of pre-defined modules.

In our examples above, execution of a module operation was specified as a single event. In general, a module implementation could contain many callable operations, each of them consisting of a group of events. Demonstrating the correctness of a operation call would then become a non-trivial task.

Since Event B is a refinement-based formalism, the problem can be solved by applying the classical rules of program correctness, in particular, the correctness rules for operation calls[7, 9]. Basically, following these rules, it is sufficient to show the relationships between the pre- / postcondition of a operation call and the corresponding pre- / postcondition of a module operation. Specifically, we need to prove that

$$\mathsf{Prec} \wedge \mathsf{M\_Inv} \Rightarrow \mathsf{M\_Prec}$$

$$\mathsf{M\_Post} \wedge \mathsf{M\_Inv} \Rightarrow \mathsf{Post}$$

where Prec, Post and M_Prec, M_Post specify respectively an operation call and an module operation itself.

The pre- and postcondition for a module operation then become a part of the externally visible module description, alongside with the external module variables and invariant. Such an external description is called a *module interface*. An exact structure of a module interface will be presented in the next section.

Let us recall the example from the previous section. However, this time the module interface describing the module external state, invariant, and operation preconditions and postconditions is available. Then it can be shown that the operation calling_op is just a syntactic sugaring for the following (provided that the above conditions on the preconditions and postconditions are proved):

```
calling_op =
    ANY
        ext', result
    WHERE
        Prec'(v,ext)
        M_Post(In_Expr(v),ext,ext',result)
    THEN
        v := Out_Expr(result)
        ext := ext'
    END
```

The required sequence of parameter passing, external operation execution, and returning of its results is now implicitly modelled by new local variables and their initilisation in the operation guard.

In this section we demonstrated that the module interfaces can be very useful verying the correctness of a module operation call. However, the examples considered so far are still pretty simple. In the next section we will discuss the structure and semantics of modules and their interfaces in a general case.

5 Extending Event B with Modules

5.1 Module Interface

Our main objectives are to facilitate model reuse and enable concurrent development of formal models. The interface concept plays a central role in achieving this. The introduction of an operation call can be validated by considering only an interface description of a called operation. Symmetrically, an implementation of an operation does not have to be aware of a possible context of an operation call since the validation is done againts the requirements stated in the interface. In other words, a module interface allows a module user to invoke module operations and observe module external variables without having to inspect module implementation details.

In our approach, a module interface consists of external module variables (w), constants (c), and sets (s), the external module invariant, and a collection of module operations, characterised by their pre- and post-conditions.

MODULE_INTERFACE MI =
  SEES Interface_Context
  VARIABLES w
  INVARIANT M_Inv(c, s, w)
  OPERATIONS
    res ← op_1(par) =
      PRECONDITION M_Pre_1(c, s, par, w)
      POSTCONDITION M_Post_1(c, s, par, w, w', res')
  ...
END

A module interface does not have an initialisation (it is provided by a module implementation) and there are no events. However, an interface still must satisfy certain consistency conditions typical for Event B specifications – operation *feasibility* (i.e., there are some states that would satisfy pre- and postconditions) and preservation of the module invariant:

$$\exists res', w' \cdot M\_Inv(c,s,w) \wedge M\_Pre(c,s,p,w) \wedge M\_Post(c,s,p,w,w',res') \quad (1)$$

$$M\_Inv(c,s,w) \wedge M\_Pre(c,s,p,w) \wedge M\_Post(c,s,p,w,r',w') \Rightarrow M\_Inv(c,s,w') \quad (2)$$

A module development always starts with the design of an interface. Once an interface is formulated and declared final it cannot be altered in any manner. This ensures that an operation call context is recomposable with an operation implementation, provided by the last refinement step of a module body.

5.2 Module Body

A module interface formally defines a collection of module operations. Obviously, it should be complemented by the corresponding module body that provides a suitable implementation for each operation. Since an Event-B specification has a flat structure, there is a problem of relating an interface operation declaration to a set of events implementing the

operation. To show correctness of a module implementation, we need a clear separation between the events implementing different module operations.

The solution we are putting forward is based on an introduction of a simple specification structuring mechanism. The events associated with a particular operation are put together forming an *event group*. Several event groups make up a body a module implementation, one group for each interface operation. The defining property of an event group is the following: once a control is passed to a group, the group runs till termination without interference from other groups. This allow us to formulate correctness conditions by considering only an operation and its associated event group.

Events groups simply partition events of a machine. A module body defining a collection of groups has the following structure:

MODULE M =
  VARIABLES w
  INVARIANT M_Inv
  GROUP *group_name_1*
    (events)
  GROUP *group_name_2*
    (events)
  ...
END

The name of a group must match the name of an interface operation definition. Each interface operation is associated with one group and vice versa. The termination of an event group corresponds to the termination of an operation call.

Events of a group obey the usual Event-B consistency and refinement conditions with an additional constraint requiring that a refined event inherits a group membership from its abstract counterpart.

The pre- and postconditions of an interface operation define high-level requirements to the behaviour of an event group. At least one event of an event group must be enabled in the state described by the operation precondition.

$$\mathsf{M\_Pre} \Rightarrow \mathsf{G}_1 \vee \mathsf{G}_2 \vee \cdots \vee \mathsf{G}_n \tag{3}$$

Each of the events returning control back from an event group must satisfy the operation postcondition and provide suitable return values.

$$\mathsf{Post}_{\mathsf{ev}}(\mathsf{w}, \mathsf{w}') \wedge \neg(\mathsf{G}_1(\mathsf{w}') \vee \mathsf{G}_2(\mathsf{w}') \vee \cdots \vee \mathsf{G}_n(\mathsf{w}')) \Rightarrow \mathsf{M\_Post}(\mathsf{w}') \tag{4}$$

where $Post_{ev}$ is the event postcondition.

A divergent event group cannot be a proper implementation of an operation. Therefore, In the first model realising a given interface (that is, an abstract module implementation) all the event groups must be terminating. The further refinement steps have to demonstrate the non-divergence of new events, as it is done in a conventional Event-B development.

5.3 Operation Invocation

The syntactic shorthand for an operation invocation is a function call. The interpretation behind such a shorthand is based on the interface attributes of an operation: its pre- and

post-conditions. We have already discussed a simple case when just one invocation happens within an action. However, our approach scales well to several invocations even when there is a complex interlink between call instances such as using the result of one operation as a parameter for another.

The semantics of an operation call is given by the computation of an equivalent statement that would be free from the call. Let us consider the following general case of an event which action relies on an operation call:

$$\mathsf{E} \;=\; \textbf{WHEN}\ \mathsf{G(v, w)}\ \textbf{THEN}\ v\ :|\ \mathsf{Post(v, w, v', op(a))}\ \textbf{END}$$

Here the predicate *Post* is the before-after predicate of the event E. It relates the current model state v to the next state v' and also, indirectly, via the operation call, the current external module state w to the next state w'. The result of the operation call $\mathsf{op(a)}$ ia used in Post to constrain v'. The following rewrite rule replaces the operation call with an equivalent characterisation based on the module interface pre- and postconditions:

$$\begin{array}{l}
\mathsf{E} \;=\; \textbf{ANY}\ \mathsf{res, w'}\ \textbf{WHERE} \\
\qquad \mathsf{M_Inv(w)} \wedge \mathsf{M_Pre(par,w)} \wedge \mathsf{M_Post(par, w, w', res))[a/par]} \\
\quad \textbf{THEN} \\
\qquad \mathsf{v\ :|\ Post'(v, w, v',res)} \\
\qquad \mathsf{w := w'} \\
\quad \textbf{END}
\end{array}$$

where $\mathsf{M\_Inv(w)}$ is the module invariant and $\mathsf{M\_Pre}$ and $\mathsf{M\_Post}$ are the pre- and post-conditions of the operation op. The new postcondition Post' is computed by replacing all the occurrences of op invocations with the local variable *res*, constrained in the event guard to a possible return value of op.

Since there can be more than one such invocation, the rule has to be applied iteratively until there are no operation calls left. The important point is the order in which invocations are eliminated. In a general case, there is a causal link between calls because each subsequent call may observe side effects (updates of module external or internal variables) of all the preceding calls. Another form of a causal link is passing the result of an operation call as a parameter to another call. The collection of causal relationships defines a total order on operation calls of an event. Once this ordering of calls is defined, we apply the above rule iteratively. The result is the following syntactic translation. For some event depending on a set of operation calls $a_1, \ldots, a_n$

$$\mathsf{E} \;=\; \textbf{WHEN}\ \mathsf{G(v, w)}\ \textbf{THEN}\ \mathsf{v\ :|\ Post(v, w, op_1(a_1), \ldots\ , op_n(a_n), v')}$$

the corresponding (free of operation calls) translation is computed as follows:

$$\begin{array}{l}
\mathsf{E} \;= \\
\qquad \textbf{ANY}\ \mathsf{res_1}, \mathsf{w_1'} \\
\qquad \textbf{WHERE}\ \mathsf{G(v, w)} \wedge \mathsf{call(1)[a_1\ /\ par_1][osub(0)]} \\
\qquad\quad \textbf{ANY}\ \mathsf{res_2}, \mathsf{w_2'} \\
\qquad\quad \textbf{WHERE}\ \mathsf{call(2)[a_2}, \mathsf{w_1'/\ par_2}, \mathsf{w_2][osub(1)]} \\
\qquad\qquad \ldots
\end{array}$$

ANY res_n, w'_n
WHERE call(n)[a_n, w'_{n-1}/ par_n, w_n][osub(n-1)]
THEN
$w := w'_n$
$v :| Post(v, par, op_1(a_1), \ldots, op_n(a_n), v')[osub(n)]$
END
END
END

where $[osub(k)]$ is the subsitution $[res_1, \ldots, res_k/op_1(a_1), \ldots op_k(a_n)]$, and $call(k)$ stands for $M\_Inv(w) \wedge M\_Pre_k(w_k, par_k) \wedge M\_Post_k(par_k, w_k, w'_k, res_k)$. Here Pre_k and $Post_k$ are the pre- and post-conditions of the operation op_k. A nested ANY construct is a syntactic sugaring that may be reduced to a single ANY. More details on this may be found in the Rodin deliverable on the Event-B language [13].

The expansion of operation calls into a plain Event-B notation reduces the problem of operation call verification to conventional set of proof obligations generated for an Event-B event. However, we are not proposing to do such conversion in practice – this would undermine all the benefits provided by a syntactical representation of an operation call. Instead, we rely on the expanded form to derive the proof obligations neccessary to demonstrate event correctness. From practical view, a tool implementing the operation call mechanism would do the operation call expansion as an intermediate step prior to the generation of proof obligations.

6 Modularization of the DPU unit

This section presents an application of our modularization approach in Event B to model one of important DPU subsystems, responsible for TC validation.

6.1 The Validation Module

The arrived telecommands should be validated (i.e., checked for syntactic and semantic correctness of their fields) before they are forwarded to execution. The core software is responsible for syntactic (”early”) checking, while the telecommand target software (which can be either the core software or application software) does more thorough (”late”) semantical checking.

In the Event B specification, the validation stage of telecommand processing corresponds to a group of events, covering different cases depending on the telecommand type, the software component (process) it is targeted to, the current core software mode etc. As a result of validation, the status of the processed telecommand is changed to either Accepted or Rejected. In addition, the additional set variable Exclusive_Rej is updated in the case when the core software rejects the telecommand. The information from Exclusive_Rej is needed by the core software later – in the reporting phase.

One of examples of such validation events is as follows:

Reject_Private_TC_Early $=$
ANY
tc_handler
WHERE
tc $\in$ dom(TC_pool)
TC_status(tc_handler) = TC_Unchecked
TCpool(tc_handler) $\in$ VALID_TCS
Type_of_TC(TCpool(tc_handler)) $\in$ PRIVATE_TC_TYPES
CSW_mode $\neq$ Operational
THEN
TC_status(tc_handler) := TC_Rejected
Exclusive_Rej := Exclusive_Rej $\cup$ {tc_handler}
END

This is an abstract event specifying one such case when the considered TC belongs to private (i.e., mission-specific) TC type and the core software is not in the operational mode (i.e., is on standby or in the safe mode). As a result, the core software rejects the telecommand and marks it as ”exclusively rejected”.

Many implementation details describing the validation process (especially the acceptance of TCs) are still missing and could be added in the later refinement steps. However, we would like to move the whole group of validation cases into a separate module (called Validation) and develop this module further independently. The case analysis and application of concrete validation actions would happen then within the Validation module. Therefore, we can specify the validation phase within a single operation event containing a call to the operation Validate described in this module.

Validate_op $=$
ANY
tc_handler
WHERE
tc $\in$ dom(TC_pool)
TC_status(tc_handler) = TC_Unchecked
THEN
TC_status(tc_handler) := Validate(tc_handler,CSW_mode)
END

The parameters for calling the Validate operation are the TC being processed as well the current core software mode. The returned result is the new status of the processed TC. Please note the absence of the variable Exclusive_Rej in the calling operation. The reason for that is that we turn Exclusive_Rej into an external variable of the new module. The ”external” status would allow other components read the current value of this variable. The variable will be updated internally, when needed to record ”exclusive” rejection. The additional module operation Remove_Exclusive would allow other the calling component

to remove a particular tc_handler from Exclusive_Rej after it served its purpose (i.e., in the reporting phase).

The following excerpt of the Validation module interface contains declaration of the external module variable Exclusive_Rej as well as the interfaces for the operations Validate and Exclusive_Remove.

MODULE_INTERFACE Validation $=$

VARIABLES Exclusive_Rej

INVARIANT

Exclusive_Rej $\subseteq$ TC_ADDRESSES

...

OPERATIONS

res1 $\leftarrow$ Validate(tc_handler,CSW_mode) =

PRECONDITION

tc_handler$\in$dom(TCpool)

CSW_mode $\in$ MODES

TC_status(tc_handler)=Unchecked

POSTCONDITION

res1 $\in \{$TC_Accepted,TC_Rejected$\}$

tc_handler$\in$Exclusive_Rej' $\Rightarrow$ res1=TC_Rejected

TC_pool(tc_handler)$\neq$VALID_TCS $\Rightarrow$ tc_handler$\in$Exclusive_Rej'

Type_of_TC(TC_pool(tc_handler))$\in$ PRIVATE_TC_TYPES $\wedge$

CSW_mode$\neq$Operational $\Rightarrow$ tc_handler$\in$Exclusive_Rej'

...

res2 $\leftarrow$ Exclusive_Remove(tc_handler) =

PRECONDITION

tc_handler $\in$ Exclusive_Rej

TC_status(tc_handler) = Rejected

POSTCONDITION

res2 $\in$ BOOL

(res2 = TRUE) $\Rightarrow$ (Exclusive_Rej' = Exclusive_Rej$\setminus\{$tc_handler$\}$)

(res2 = FALSE) $\Rightarrow$ (Exclusive_Rej' = Exclusive_Rej)

...

END

6.2 Module Architecture

The Validation module is just one example of DPU modularization. Below we present the suggested module architecture, structuring the Core software and instruments into several different modules such as Validation, Reporting, Mode Management and so on, each containing callable operations and both external and internal data. The modules TC pool and TM pool are especially interesting, since they essentially implement datatypes (classes) for handling currently processed TCs and TMs.

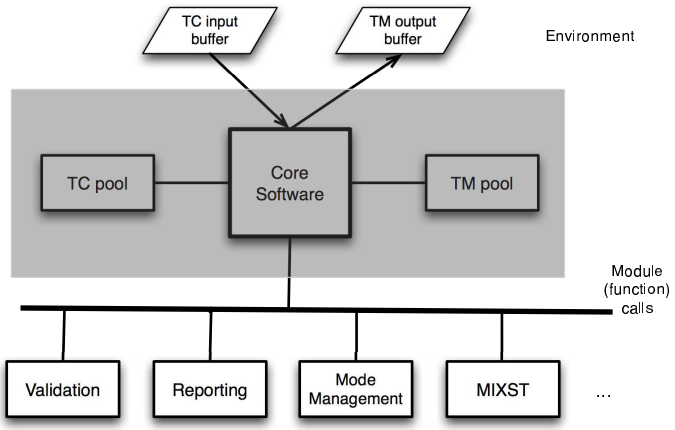


Fig. 2. Module Architecture

7 Conclusions

In this paper we proposed a pragmatic approach to supporting modularization in Event B. This work was motivated by the formal development conducted by Space Systems Finland[11]. We described the system that have been developed, presented the development approach and experience gained from the development. The analysis of the development has shown that the lack of modularization makes the approach unscalable. Yet the top-down development paradigm and automated proof-based verification offer an attractive design platform. Our conservative extension of Event B alleviates scalability problem while preserving all the benefits.

The proposed approach to modularization can be seen as a special case of the ”shared variables” type of decomposition by J.-R.Abrial[13]. Abrial aims at enabling decomposition for distributed systems. Hence his approach is more general and complex. In our case, the systems under construction are sequential, even though their functionality is distributed among several modules. Our goal was to enable parallel development of several independent parts of the system as well as reuse formally developed modules in other developments.

Another proposal for supporting decomposition in Event B aims at ”shared events” style decomposition for distributed systems [6]. Finally, there is also proposal for supporting event fusion in Event B[12]. However, all these works offer more general and hence more difficult to implement alternatives for the modularization.

We believe that our proposal for supporting modularization for Event B can help to keep a positive momentum gained in the recent development and pave a path towards industrial deployment of formal engineering. In our future work we are planning to implement our approach as a plug-in to the RODIN platform.

Acknowledgments

This work is supported by IST FP7 DEPLOY Project.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. J.-R. Abrial. Extending B without Changing it (for Developing Distributed Systems). *Proceedings of 1st Conference on the B Method*, pp.169-191, Springer-Verlag, November 1996, Nantes, France.
3. R. Back. Refinement calculus, Part II: Parallel and reactive programs. *Stepwise Refinement of Distributed Systems, Lecture Notes in Computer Science*, Vol.430, pp.67-93, Springer-Verlag, 1990.
4. R. Back and K. Sere. Superposition refinement of reactive systems. *Formal Aspects of Computing*, 8(3), pp.1-23, 1996.
5. BepiColombo Overview. Online at http://www.esa.int/esaSC/120391_index_0_m.html.
6. M. Butler. Decomposition Structures for Event-B. *Proc. of Integrated Formal Methods'2009*, 2009.
7. D. Gries and G. Levin. Assignment and Procedure Call Proof Rules. *ACM Transactions on Programming Language Systems*, Vol.2, pp.564–579, 1981.
8. Industrial deployment of system engineering methods providing high dependability and productivity (DEPLOY). IST FP7 project, online at http://www.deploy-project.eu/.
9. A. J. Martin. A General Proof Rule for Procedures in Predicate Transformer Semantics. *Acta Informatica*, Vol.20, pp.301–313, 1983.
10. MATISSE. *Handbook for Correct System Construction*. Available at http://www.esil.univ-mrs.fr/ spc/matisse/Handbook/.
11. OBSW formal development in Event B. Online at http://deploy-eprints.ecs.soton.ac.uk/view/type/rodin=5Farchive.html.
12. M. Poppleton. Decomposition Structures for Event-B. *Proc. of ABZ2008: Int. Conference on ASM, B and Z, 16-18 September 2008, London*, 2008.
13. Rigorous Open Development Environment for Complex Systems (RODIN). Deliverable D7, Event B Language, online at http://rodin.cs.ncl.ac.uk/.
14. The RODIN platform. online at http://rodin-b-sharp.sourceforge.net/.