ABZ Languages and Tools in Industrial-Scale Application

Janet Barnes, Jonathan Hammond, Angela Wallenburg(✉), and Thomas Wilson

Altran UK Limited, 22 St Lawrence Street, Bath BA1 1AN, UK
{janet.barnes,jonathan.hammond,angela.wallenburg,
thomas.wilson}@altran.com

Abstract. We give an early view of an ongoing evaluation of ABZ-style languages and their accompanying tools. The target is specifications of safety- and security-critical (software-rich) systems. Our perspective is that of long-term users of formal methods in all parts of the development life cycle. The evaluation's scope is the *production* of specifications. We list requirements for producing specifications, including semantic needs and the resulting requirements on language expressiveness, as well as requirements on tool support for writing, structuring, exploring, and validating specifications. We define criteria for industrial suitability – in our experience – of ABZ languages. We believe that specification structuring is a major discriminating factor for industrial scale-up. So we present an (informal) classification of such mechanisms and illustrate their use by reference to the largest formal specification written by Altran. Our lack of industrial-scale experience in some languages means we are still learning the best mechanisms to use in some cases. We welcome input on this. Finally we discuss remaining work.

1 Background

The SECT-AIR project [6] is a consortium of UK universities and companies with the aim of delivering a step-change improvement in the affordability of aerospace software. The project is looking at how advanced techniques and tooling, including formal methods, can increase productivity across the whole lifecycle.

Altran UK has nearly 30 years of industrial formal methods experience across the lifecycle, including specification in various languages (e.g. VDM, Z, CCS and CSP) and software formal verification and static analysis (e.g. SPARK [5]).

2 Scope: *Production* of Specifications

One SECT-AIR project workpackage is to reduce barriers to the use of formal specification. Although Altran successfully uses formal methods across the lifecycle, we have more challenges with high-level[1] formal specifications. So as part

[1] By high-level we mean expressed in customer/domain, not software, terms and concepts.

© Springer International Publishing AG, part of Springer Nature 2018
M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 3–15, 2018.
https://doi.org/10.1007/978-3-319-91271-4_1

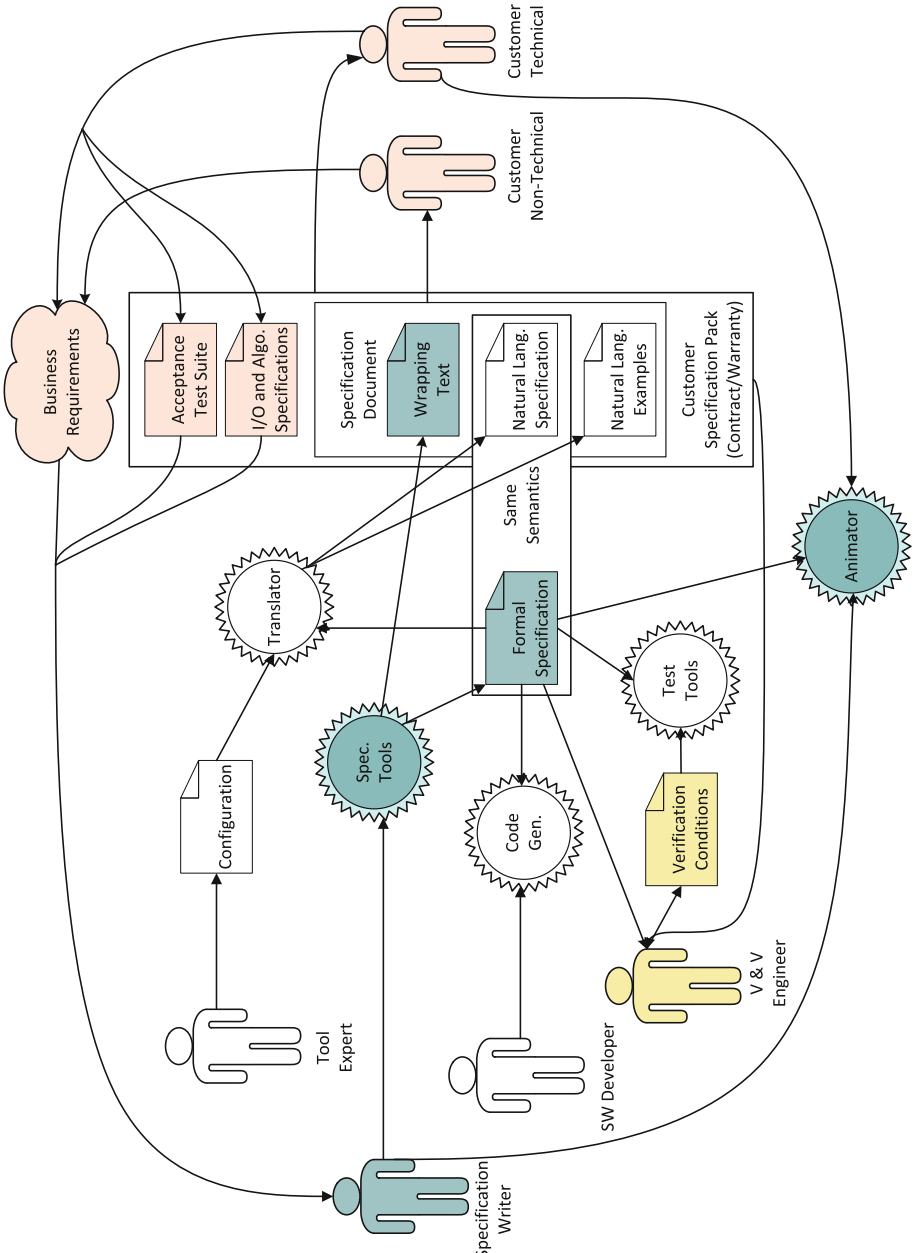

Fig. 1. *Specifications sit at the centre of several key data flows.* The colouring highlights artefacts produced by, and tools driven by, stakeholders, rather than more autonomously. The arrows show the main work flows. Note: the figure shows a wider scope than this paper has space to discuss (e.g. auto-generation of natural language text). The focus of this paper is on the *Spec. Tools* used by the *Specification Writer(s)* to produce the *Formal Specification*, an activity that is beneficial almost regardless of how the formal specification is used later. (Color figure online)

of the SECT-AIR workpackage, Altran is over-hauling its specification solution to improve communication and automation. This solution is to cover the full gamut of specification-related activities, including: writing (editing), exploring (reading, animating, navigating), verifying (e.g. static semantics), validating etc.

Figure 1 summarises the specification solution data flows, showing: (1) stakeholders, (2) artefacts, and (3) tools. The formal specification is at the centre.

An initial downselection of potential solutions has chosen both reactive control and ABZ languages as key parts of meeting SECT-AIR needs. The rest of this paper: makes observations about our industrial formal specification experience (Sect. 3); defines the requirements for our ideal specification solution (Sect. 4); describes our criteria for evaluating possible solutions (Sect. 5); summarises our evaluation process (Sect. 6); discusses ABZ language structuring mechanisms (Sect. 7); finishes with concluding remarks and future work.

3 Observations on Industrial Formal Specification

This section describes our observations based on informal interviewing and discussion with specification writers about their experience of producing specifications over decades at Altran UK, formerly Praxis. (For a wider, including non-engineering, perspective see Sect. 4.) In this section we focus on the notation that we have most commonly used: Z.

3.1 Cultural Setting: Formal Methods Usage Is Normal

Altran UK enjoys a long tradition of substantial industrial use of formal methods. It is regarded as a normal part of multiple stages in the life cycle. Formal methods are used daily by many engineers, not just by a small group of experts.

3.2 Reading Versus Writing Formal Specifications

Altran trains both specification and implementation (coders and verifiers) engineers, using our customised 4 day-long Z courses. Over 100 have been trained to *read* Z and in our experience the majority learn to read the core Z language fairly well. Where appropriate we have also trained customers to read Z.

A much smaller number are needed, and thus trained, to *write* Z. However, only about a dozen have become productive writers, fewer than we'd ideally like.

3.3 Some Observations About Specification Writers

Our most productive specification writers exhibit the following:

1. Particularly skilled in balancing an overview perspective with attention to detail. Unfortunately, this seems to be a rare skill. Most people seem to be good at one or other, but cannot swiftly change between the two.

2. 85% have a PhD, although this is not an Altran job requirement. The PhDs are varied but all in STEM subjects.
3. *Not* an academic expert in ABZ notations and theories, although one exception has a PhD in CSP.

The above sample of writers is too small to draw significant conclusions from. Furthermore, while we are confident that (1) above is a core required skill, we are not sure what to conclude from (2) and (3). Regardless, our observations help illustrate the challenge to find sufficient productive specification writers. We want a specification solution that more people can use to write productively.

3.4 Specification Style and Structure

The largest formal specification produced by Altran is for an air traffic management (ATM) system, called iFACTS [3,17], developed for NATS (UK National Air Traffic Services).

In style, the iFACTS specification – written in Z – bears much resemblance to the detailed specification (called formal design) of the Tokeneer system (also in Z) [4], which is openly available [2]. Each of these specifications:

- Has a high-integrity aspect (safety for iFACTS and security for Tokeneer)
- Specifies software that has a user interface (UI) and receives inputs from users and the wider system environment (other complex ATM software systems for iFACTS and sensors for Tokeneer)
- Is model-based with: data model, initial state, operations and partial operations.

However, iFACTS' main specification is over 3,000 pages of Z and English and is an order of magnitude larger than the Tokeneer specification. So we use the iFACTS specification to illustrate the structuring discussions in this paper.

Why Is the iFACTS Specification so Large? The level of abstraction is a big factor. However, with the exception of some core algorithm details, it is a high-level specification written in ATM domain terms. The domain is a complex one, due to the variety and richness of its data and to the substantial operational rules governing users' responsibilities and thus what iFACTS does. Furthermore, the UI's role in safe and efficient ATM, led to the modelling of individual UI functions, such as buttons and menu items, as distinct Z operations.

Our view is that the specification's content is not implementation detail to be left for software architects and developers (see also refinement Sect. 7.7).

iFACTS Capability. To help explain the iFACTS specification structure, and illustrate it with examples, a basic understanding of what iFACTS does is useful.

iFACTS provides air traffic controllers with advanced support tools based on predictions of the trajectories flights will take up to 18 min into the future. These predictions are used to:

- Detect potential conflicts (with respect to physical separation) between pairs of flights up to 15 min into the future
- Monitor aircraft and if they deviate from a controller's instructions, alert and identify any potential conflicts that result
- Enable a controller to assess in advance the consequences of different instructions they could issue but have not yet issued

The introduction of iFACTS has in the customer's view [17]: "*... revolutionised our operation, freeing up capacity and improving safety, while at the same time reducing delays and cutting carbon emissions.*".

Data Model Packages. The data model is divided into packages (as per the UML term). Each package describes a group of related data items, with the aim to have high cohesion of items within each package and low coupling between packages.

There are a couple of dozen iFACTS packages in the following groupings:

- *UI*: each major UI element (such as each tool) is a separate package.
- *Core algorithms*: each different type of algorithmic processing iFACTS performs (trajectory prediction, deviation monitoring and conflict detection) has its own package.
- *Live domain data*: iFACTS receives and uses a wide range of data that changes in real-time. Each type of data has its own package, for example:
  - Radar data
  - Flight plans
  - Clearances (the ATM term for instructions issued by controllers)
  - Weather forecasts
- *Configured domain data*: such as airspace definitions and aircraft performance models that are fixed at run-time.

The packages form a hierarchy (strictly a partial ordering) where data items in a package may depend on data items in packages beneath it in the hierarchy. The groupings are listed above in top-down order, as elements of the UI present data generated by the core algorithms which in turn use live domain data etc.

The data items in a package are primarily described as classes and relevant associations. UML class diagrams are used to give an overview, with Z schema types, relations and invariants specifying the detail. There is a wide range of package sizes. For example, the radar data package class diagram has two classes, whereas the trajectory prediction package class diagrams total over thirty classes.

Operations and Partial Operations. An *operation* specifies the state change for the entire data model that results from a single external stimulus (such as a user input). A *partial operation* specifies the state change of a single package. An operation is specified as a particular combination (conjunction) of partial operations (plus Ξ-schema predicates for unchanged packages).

The iFACTS specification has approximately 165 operations. Each represents a distinct, discrete piece of functionality that iFACTS provides. About 120 operations can be user initiated and about 50 by either the receipt of data from an external system or due to the expiry of a timer. Note therefore that a handful of the 165 operations have multiple stimuli. For example, a cancel operation may occur either due to user action or because a timer has expired. For iFACTS the number of partial operations varies significantly between packages. For example, the radar data package has just four partial operations, whereas the clearances and some of the UI packages have over thirty each.

4 Specification Solution Requirements

Drawing on our above experience and wider business context (including input from managers as well as engineers), Altran's ideal specification solution would:

R1. *Have a sufficiently low adoption hurdle.* Particularly for specification users, e.g. non-technical readers, for whom minimal/no training should be needed.
R2. *Be amenable to translation to a format suitable for sign-off.* Specifications must be accessible to customers.
R3. *Be amenable to tool assisted validation.* Both readers and writers need ways to check their understanding is correct.
R4. *Have a mathematical underpinning.* Vital to enable meaningful analysis.
R5. *Be expressive enough to capture high-level concepts easily.* Drives ease of understanding and productivity.
R6. *Be scalable to very large systems.* Applies to both tool support and readability.
R7. *Execute as a test oracle or facilitate code generation.* A key business driver to avoid separately duplicating semantics in all of specification, code and test.
R8. *Have a minimum number of languages and tools for the required domains.* We work across domains and cannot afford to maintain competency in a wide range of languages/tools.

There is deliberately no requirement for *formal* proof support. Although we have had notable success with (informal) specification proof [14], we do not believe the cost vs benefit is worthwhile in most cases. That would change if significant automation is possible, for the scale of systems we build.

5 Evaluation Criteria

To evaluate whether languages and tools achieve the above requirements, we have derived the evaluation criteria below. Tracing is given back to the requirements. The weighting reflects the relative importance we place on the criteria.

Evaluation criteria	Weight	Rationale
Format suitable for sign-off (R2)	10	Top requirement to reduce risk with client and enable more use of spec. as basis for contract/warranty
Be amenable to tool assisted validation (R3)	5	Examples: type checking, animation, example generation, consistency checking
Be scalable to very large systems (R6)	8	Both for tools and readability. Often overlooked, key for exploitation
Execute as a test oracle or facilitate code generation (R7)	10	Main cost driver
Writers' satisfaction w.r.t. semantics, expressiveness, abstraction mechanisms (R5)	8	Writing formal specifications pays off by the activity itself, providing writing is sufficiently productive
Have a sufficiently low adoption hurdle (R1)	5	More important for organisations with less experience in applying formal methods
Have a mathematical underpinning (R4)	8	A must for disambiguation, risk reduction, and reliable tools
Be applicable to reactive control systems (R8)	8	Control software projects, often embedded, for example: safety control, engine control, brake control, power control, fly-by-wire, alarm handling, etc.
Be applicable to state based systems (R8)	8	Data and history-rich projects, for example: systems for database and configuration validity, tracking, and book-keeping
Evidence of successful adoption (R1, R6)	8	Is there evidence of repeat and/or widespread (industrial) use (beyond case studies)? Technologies with significant tool building efforts but without repeated use score low. This helps balance any "hyped" technologies. New technologies also score low. Note: this does not exclude new technologies as alternatives, but the reduced score is intended to reflect risk
Tools cost, development and maintenance (R6, R8)	6	Cost matters but has lower weight than most criteria due to expected benefits being worth the investment

These criteria come from both our own experience (e.g. [10]) and business drivers, as well as others' published experiences, such as [16].

6 Evaluation Process

6.1 DAR Process

We are using a Decision Analysis and Resolution (DAR) process to choose our specification solution. DAR is part of Capability Maturity Model Integration (CMMI) [8]. Its purpose is to analyse and document possible decisions using a structured process with evaluation against established criteria.

6.2 First Pass: Creating and Choosing from a Short-List

Based on the requirements (Sect. 4) and an assessment of available language classes, a short-list of 5 possible solutions was created. Grid analysis combined with balanced pairwise comparison led to a preferred solution: a combination of natural language (English) with significant auto-generation from a formal language – either a reactive control or ABZ language, depending on domain. The possible language choices, for more detailed evaluation, are:

1. Reactive control language: Lustre, SCADE, ArgoSim.
2. ABZ language: Event-B, Z, TLA+, VDM

The rest of this paper focuses on the ABZ languages.

6.3 Second Pass: More Detailed Evaluations

Each of the above possible languages is now subject to a more detailed evaluation. This includes trying the languages and associated toolsets on both tutorial examples and real specifications (such as Tokeneer [2]). The final language choices will then be made from the results of these more detailed evaluations.

7 Discussion of Structuring Mechanisms

The ABZ languages have a lot in common; most use standard mathematical notation from axiomatic set theory, lambda calculus, and first-order predicate logic. With regards to typing, they differ. Event-B, Z and VDM are all strongly-typed, unlike TLA+. We find automated type-checking very cost-effective.

Most of all, ABZ languages differ in their structuring mechanisms. In our experience the structuring features are the most discriminating for being able to scale-up to large industrial projects (such as iFACTS). Fundamentally this is because the readability, and thus usability, of a large specification is highly dependent on how it is structured. As Jackson says in his essay [13] on abstraction in software development:

> An abstraction can be represented in more than one way. Whether it is comprehensible depends not only on its formal content but also – vitally – on its representation.

We discuss below some ABZ language structuring concepts and relate them to our experience (e.g. iFACTS). We welcome input from the ABZ community.

7.1 Vertical and Horizontal Abstractions

Jackson [13] distinguishes the concepts of *vertical* and *horizontal abstraction*. A *vertical* abstraction introduces a new (higher-level) concept that corresponds to a particular collection of (lower-level/more detailed) phenomena. For example, a circle, with centre and radius, abstracts a specific class of closed curves. A *horizontal* abstraction simply selects those concepts or phenomena that are significant for the purpose in hand. For example, the London Underground map preserves connectivity and ordering but eschews geographical accuracy.

7.2 Aggregations

A very common idiom in object-oriented systems development is *aggregation*, which encapsulates tightly-coupled state. This is used to group sub-components into a whole component. A classical example would be to model a `Car`, as an aggregation of its components e.g. `Wheels`, `Engine`, `Seats`, and so on. Typically encapsulation is achieved by not allowing the outside of the main component to tamper with its subcomponents, for example disallowing the outside of `Car` to manipulate the `Engine` that is owned by that car. Aggregation is therefore a kind of vertical abstraction. The concept of a `Car` is very useful when discussing relationships with other objects, such as owners, passengers, other traffic etc, without needing to refer to its components.

Aggregation (strictly, composition in UML) is commonly used in our formal specifications. For example, UI elements are sometimes defined as a hierarchy of constituent components. More complex data types may also be aggregations. Two iFACTS examples are trajectories and clearances. Encapsulation is partially, and purely stylistically, enforced by operations only specifying state changes by using the defined partial operations on the packages containing the aggregations.

7.3 Generalization and Specialization

Supertypes (generalization) and subtypes (specialization) are frequently used in our data models. Subtypes tend to arise from specific (specialised) needs, but supertypes can have a useful role as a horizontal abstraction, particularly when specifying the relationships (associations) with other concepts. Each subtype's schema type in Z 'inherits' the supertype's schema type via schema inclusion.

For example, in iFACTS there are several specific types of flight predictions that are natural subtypes of a unifying prediction supertype. The supertype can be used to relate to concepts like a conflict (which arises between a pair of predictions) or to define the different predictions that may be needed for the same flight (such as current expected behaviour and potential future behaviour if a controller were to issue a different clearance).

7.4 Partial Operations

It is very natural to specify separately the effects of an external stimulus on different data model packages and then combine those separate pieces to specify the complete effects. In our Z style the pieces are the partial operations. Event-B has a similar concept for sharing, or splitting, an event between machines [1,12].

If the state changes in two packages are truly independent then decomposition into separate pieces presents no problem. Of course, in general, such independence is not the case. The data model package hierarchy often means that one package's state change depends on another package's state change lower in the hierarchy. This is true in the iFACTS specification. For example, the state changes triggered by receiving new radar data for a flight include:

1. Updating the stored radar data associated with the flight
2. Generating an updated prediction using the new radar data
3. Checking for any unacceptable deviation from the flight's clearance
4. Potentially generating new predictions if deviations are found
5. Displaying the appropriate results of all the above items in the tools

Each item from 2 onwards depends on the results of at least one earlier item. Specifying such dependencies is straightforward if the lower-level package after state variables are visible in higher-level package partial operations (as in the iFACTS specification). Event decomposition may be another approach, see Sect. 7.7.

7.5 Condition Hierarchies

Another important vertical abstraction in our experience is the ability to specify hierarchies of conditions (predicates). Such conditions may be guards, at either whole operation or partial operation level, or the definition of cases or alternative courses of action within an operation or partial operation. Being able to abstract cases/groups of conditions, by naming combinations of predicates, makes the overall logic much clearer than a long list of potentially complex predicates.

The iFACTS specification makes extensive use of Z schemas as predicates to build up complex logic hierarchically. For example, deviation monitoring schemas check a flight conforms to a clearance's altitude. There are different cases for flying level, climbs and descents. Each case has subcases, such as if a descent needs to be now, or only in time to reach the required altitude at a defined point.

7.6 Abstract Data Types

An effective use of abstract data types (ADTs), in the specification of a train control system, is described in [9]. A key motivation for this ADT use is ease of refinement proof. Event-B contexts are used to give abstract types that embody domain concepts of rail network connectivity and train occupancy. Subsequent successive instantiations of the ADT, by more concrete representations, enables formal refinement to a practical implementation.

In our usage of Z, abstract data types (ADTs) are not a central concept.[2] Some schemas could be viewed as ADTs, but only by convention because encapsulation is not enforced. Typically our specifications have not required ADTs. We wrote a Z specification [15] without ADTs in the same domain as the train control system, but we had no intention to refine as far as implementation.

7.7 Role and Use of Refinement

What Is Being Refined? Refinement is the main exploiter of vertical abstraction in formal development. It is important to consider what is being refined:

[2] We refer here to use of ADTs for elements of a specification. Of course an entire model-based Z specification can be viewed as an ADT, particularly when refining it.

1. The environment of the system being developed (particularly its interface with the system), or
2. The internals of the system (such as its architecture and data formats)

We think this distinction is important due to the different degrees of freedom typically involved. Significant parts of the environment are normally a given, or at least require negotiation with various stakeholders. However, there is often much more freedom to choose system internals. The former type of refinement tends to be dominated by requirements and/or systems engineering skills, whereas the latter relates more to architecture and/or software design.

Consequently, in our experience, different people/teams need to do the different refinements. Environment refinement is part of our specification process, whereas internal refinement is part of system development. Depending on the domain this can result in large specifications, like iFACTS, due to the amount of information to be negotiated and captured to use as a basis for development.

Role of Decomposition. To avoid too much detail at once, it seems to us that Event-B uses refinement to decompose and add detail [1,12]. This includes event decomposition, which allows an atomic event to be decomposed into a series of events. This could be used to specify iFACTS' radar update example (Sect. 7.4), as the listed steps appear to be a natural sequence. However, we see such decomposition as an internal refinement issue, best left to software architects and developers, not specifiers. There may be factors other than logical dependencies, such as performance, that affect how best to decompose the event.

Either way, event decomposition clearly supports internal refinement, such as an architecture of communicating subsystems implementing a system-level event. Event decomposition could also apply to environment refinement. However, when an environment interface is known in advance (e.g. it already exists) it is harder to justify the cost of constructing more abstract models, to be able to refine down to the already known interface, even if insights are gained (such as discussed in [7]) through the abstraction.

Where Is Refinement Beneficial? Our experience is that the major sources of error are things like significant domain misunderstandings and missing stakeholders. (For example, see the e-commerce security case study in [11].) Our impression is that formal refinement methods have limited value in eliminating these.

Conversely, formal refinement clearly assists critical property preservation, such as Tokeneer security requirements. Although we routinely do *informal* (especially internal) refinement, formal refinement has not been found cost-effective on our projects so far. This has been due to (1) the large effort required for formal proof of refinement steps, (2) not enough benefit given (insights or issues found) from the typically small refinement steps that have been required for proof, and (3) the cost of maintaining several descriptions of the same system, which can be seen as redundant in the feature-oriented more than insight-oriented industrial setting.

Currently we are devising approaches to do bigger refinement steps, formally stated, but verified by dynamic means. With this we hope to gain meaningful insights during the process of formalising the refinement, as well as increase the productivity of the V&V process.

8 Conclusions and Future Work

We have discussed how scalability, to the kinds of systems we build, is a key discriminating factor between languages and tools. We are still learning the appropriate language mechanisms in the various ABZ languages to manage the amount and detail of information needed for some of our domains. We welcome the ABZ community's input on the structuring idioms discussed in this paper.

We will complete the evaluations of the possible languages and toolsets (see Sect. 6.3). A comparison table will be produced to show the assessment of each against the evaluation criteria. This will drive the selection for our specification solution. Case studies will then be written using this solution, which will be published – as with all key SECT-AIR project outputs – in appropriate fora.

Acknowledgements. This work was made possible by the ATI SECT-AIR project. SECT-AIR is a UK collaborative Aerospace Technology Institute (ATI) research project with the twin aims of reducing aerospace software costs and timescales. SECT-AIR is co-funded by Innovate UK. We would also like to thank our Altran colleagues and John Colley for their valuable input.

References

1. Abrial, J.-R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: application to event-B. Fundam. Inform. **77**(1–2), 1–28 (2007)
2. Altran. Tokeneer project release (2008). https://www.adacore.com/tokeneer. Accessed 13 Dec 2017
3. Astill, J.: Precision instruments. Airport Focus International, September 2012. http://airportfocusinternational.com/precision-instruments/. Accessed 20 Dec 2017
4. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the Tokeneer enclave protection software. In: 1st IEEE International Symposium on Secure Software Engineering, March 2006
5. Barnes, J.: SPARK: The Proven Approach to High Integrity Software, 3rd edn. Altran, Paris (2012)
6. Bennett, M.: SECT-AIR: a UK initiative to reduce aerospace software cost. Presentation at Verification Futures 2017 Conference, April 2017. http://www.testandverification.com/conferences/verification-futures/vf2017-europe/vf2017-sect-air/. Accessed 3 Jan 2017
7. Butler, M.J.: Mastering system analysis and design through abstraction and refinement. In: Broy, M., Peled, D.A., Kalus, G. (eds.) Engineering Dependable Software Systems. NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 34, pp. 49–78. IOS Press (2013)

8. Chrissis, M.B., Konrad, M., Shrum, S.: CMMI Guidelines for Process Integration and Product Improvement. Addison-Wesley Longman Publishing Co. Inc., Boston (2003)
9. Fürst, A., Hoang, T.S., Basin, D.A., Sato, N., Miyazaki, K.: Large-scale system development using abstract data types and refinement. Sci. Comput. Program. **131**, 59–75 (2016)
10. Hall, A.: What does industry need from formal specification techniques? In: Proceedings of 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques, pp. 2–7 (1998)
11. Hammond, J., Rawlings, R., Hall, A.: Will it work? In: Proceedings of the Fifth IEEE International Symposium on Requirements Engineering, RE 2001, pp. 102-. IEEE Computer Society, Washington, DC (2001)
12. Hoang, T.S., Iliasov, A., Silva, R., Wei, W.: A survey on event-B decomposition. ECEASST **46**, 1–15 (2011)
13. Jackson, M.: Aspects of abstraction in software development. Softw. Syst. Model. **11**(4), 495–511 (2012)
14. King, S., Hammond, J., Chapman, R., Pryor, A.: Is proof more cost-effective than testing? IEEE Trans. Softw. Eng. **26**(8), 675–686 (2000)
15. King, T.: Formalising British rail's signalling rules. In: Naftalin, M., Denvir, T., Bertran, M. (eds.) FME 1994. LNCS, vol. 873, pp. 45–54. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58555-9_86
16. Kossak, F., Mashkoor, A.: How to select the suitable formal method for an industrial application: a survey. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 213–228. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_13
17. Rolfe, M.: How technology is transforming air traffic management. NATS Blog, Jul 2013. https://nats.aero/blog/2013/07/how-technology-is-transforming-air-traffic-management/. Accessed 20 Dec 2017