Model-Driven Re-engineering of a Pressure Sensing System: An Experience Report

Atif Mashkoor[1,2](✉), Felix Kossak[1], Miklós Biró[1], and Alexander Egyed[2]

[1] Software Competence Center Hagenberg GmbH, Hagenberg, Austria
{atif.mashkoor,felix.kossak,miklos.biro}@scch.at
[2] Johannes Kepler University, Linz, Austria
{atif.mashkoor,alexander.egyed}@jku.at

Abstract. This article presents our experience in re-engineering a pressure sensing system – a subsystem often found in safety-critical medical devices – using the B formal method. We evaluate strengths and limitations of the B method and its supporting platform Atelier B in this context. We find that the current state-of-the-art of model-oriented formal methods and associated tool-sets, especially in automatic code generation, requires further improvement to be amenable to a wider deployment to industrial applications for model-driven engineering purposes.

1 Introduction

One of the ways to promote the use of formal methods for model-driven engineering of industrial applications is to demonstrate the ability of formal methods to automatically generate executable source code from "correct by construction" software models [10]. However, automatic generation of code from a formal specification such that it requires no further human intervention or post-processing before deployment is a weak link in the development chain [3].

Dataflow-oriented frameworks, such as Simulink[1] or Safety-Critical Application Development Environment (SCADE)[2], are already popular for their model-driven engineering capabilities in safety-critical domains such as avionics and automotive systems [27]. The appeal of these frameworks stems from their graphical notation and simulation capabilities. However, as compared to model-oriented formal methods [12], these frameworks lack sophisticated verification techniques which guarantee *correctness* [10] and also suffer from scalability issues [28]. Model-oriented formal methods lend themselves better to abstraction

The writing of this article is supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

[1] http://www.mathworks.com/products/simulink.
[2] http://www.estereltechnologies.com/products/scade-suite.

A. Pierantonio and S. Trujillo (Eds.): ECMFA 2018, LNCS 10890, pp. 264–278, 2018.
https://doi.org/10.1007/978-3-319-92997-2_17

and reduction techniques, the key ingredients in modeling and proving correctness properties, and are supported by a variety of model checkers and theorem provers.

In practice, however, many projects are not about developing new software systems from scratch but improving on existing software systems or porting them to new platforms. Thus, it would also be desirable to be able to use formal methods in maintenance and re-engineering projects. Thereby it should also be possible to transform only parts or individual modules. Doing so would improve the quality of software systems and this would also increase the potential for automation such as automatic code generation.

In this article, we report about our experience with the development of a control software for a Pressure Sensing System (PSS) – a subsystem of a hemodialysis machine [21]. A typical pressure sensing system reads sensor data, transforms the data into meaningful information, saves results, checks whether different values are within certain ranges (which depend on certain modes of running the machine), and raises different types of alarm if defined thresholds are violated.

Based on our experience that stems from the application of formal methods on several industrial and academic projects, for example, hemodialysis machines [23,24], aircraft landing gear system [16], machine control systems [25], and stereoacuity measurement system [5], we decided to use the B method [2] for the task. The B method enjoys extensive tool support, covers all the necessary development phases (e.g., support for code generation), and the developers of the PSS system already had experiences with it. Related model-oriented formal methods either do not cover all phases of development, e.g., there is no automatic code generator available for Alloy [14], Temporal Logic of Actions (TLA+) [18] and Z [29], or have limited code generation capabilities, e.g., Abstract State Machines (ASMs) [8], Event-B [4] and the Vienna Development Method (VDM) [15]. A detailed comparison of various model-oriented formal methods concerning their modeling and code generation capabilities is available in [17].

The main objective of the development was the automatic generation of C language code from a formal requirements model that was, in turn, developed through a re-engineering process. Due to space limitation (and also a nondisclosure agreement with the case study provider), we do not include artifacts, such as model and code, in the paper. We also deliberately omit a detailed discussion on the "traditional" use of formal methods, e.g., requirements modeling, property verification, assessment of code complexity, and proof statistics. For such a discussion, interested readers may consult the work by Mashkoor [22] that contains a detailed account of our effort of model-driven engineering of various components of a hemodialysis machine including a discussion on verification and validation. Here, we rather focus on the modeling and code generation experience with the B method.

In the following, we first briefly describe the B method in Sect. 2. Then we present the case study including its description, overall aim and objectives in Sect. 3. In Sect. 4, we highlight the undertaken re-engineering process. In Sects. 5

and 6, we report the experiences and challenges we met in the course of modeling and code generation with the B method respectively. Section 7 discusses some related work. The paper is concluded in Sect. 8 with a discussion on deployed methods and tools.

2 B Method

B is a refinement- and state-based formal method. A B model describes data structures and operations thereupon. A state is defined by particular values of the variables of the data structure, and operations describe state transitions. A so-called *machine* captures a part of the data structure, constraints on the data structure, and operations on the values of the data structure. Requirements are basically described either by constraints (this includes, e.g., safety requirements) or by operations. In a way, a machine resembles programming code for a software module, but it is more abstract, is not tailored to a specific platform, captures requirements more directly, and is suitable for logical analysis.

Modeling in B starts with one or more *abstract machines* which are supposed to capture the most basic requirement(s) in a concise way. These machines are then refined by adding constraints and detailing the actions of operations, according to additional requirements. This is done in separate files which are called "refinements." This way, the consecutive development of the formal specification is well documented. Verification includes proving that each refinement preserves the specified properties of the abstract machine or of the previous refinement.

A special case of refinement is called "implementation." This does not add further requirements to the model, but transforms the final specification model into a form which is suitable for code generation.

The three different machine types – abstract machines, refinements, and implementations – must obey different constraints regarding the language in which they can be expressed. The language for implementations even has its own name, B0 ("B zero").

Tool support for modeling and analyzing in B is available in the form of the Atelier B platform[3]. Atelier B includes an editor, syntax and type checkers, a proof obligation generator, automatic provers and an interactive proving environment, as well as code generators for different target languages. A standalone model checker and animator, ProB [20], is available for B and can be used together with Atelier B.

3 Pressure Sensing System Case Study

3.1 Case Study Description

Pressure sensors are important ingredients of modern diagnostic and therapeutic devices such as dialysis machines, respiratory devices, drug-delivery systems and

[3] http://www.atelierb.eu.

patient monitors. Pressure sensors provide either gauge or differential pressure that is used for various purposes, e.g., extracting and measuring volumetric flow rates, and total fluid volume transferred. This allows to monitor drug administration and to detect anomalies, in which case alarms can be raised. Consequently, pressure sensors enhance the capabilities of medical devices by providing physicians with the ability to measure blood pressure, administer precise quantities of drugs or oxygen, and track patient compliance.

A PSS is safety-critical with respect to human health and life. For instance, in dialysis machines, PSS are vital to ensure that the dialyzed fluid is pumped back into the human body with the right pressure. If the pressure is not within a certain range, an alarm must be raised and the flow must be disconnected from the patient. Both admissible range and type of alarm are thereby dependent on operation mode and other circumstances. Raising an alarm is part of the tasks of PSS software, whose code may attain several thousands LoC (much of which is dedicated to interfaces and data structures).

The PSS is not large but still a nontrivial system. Its implementation uses relatively few programming constructs; in particular, no loops and no recursion are used. Thus we could test only a roughly estimated half of B's major language constructs, and even less of common programming constructs. Yet this is not uncommon for hardware control software, and the restriction of constructs used facilitates safe modeling and programming – or code generation – as well as verification. At the same time, complex data structures with many fields are used in the interface, and many and often nested case distinctions have to be made, in particular for determining whether different values are within permissible limits and how to react if not, depending on various factors, including the operation mode. The interface also requires many type casts to be performed, amongst others. With an order of magnitude of a thousand lines of code, all this makes the software liable to error and thus formal analysis of the code, and even more correct construction by design through the use of formal methods, are bound to improve correctness and thereby safety. We used this opportunity to conduct a pilot project for evaluating the feasibility of methods and tools for eventual deployment.

3.2 Aim and Objectives of the Case Study

The overall aim of the case study was to re-engineer the control software for a PSS – a subsystem of a hemodialysis machine – piece-wise so that the subsystem can be transformed into a form which allows for proving certain correctness properties, e.g., by verification (by automatic theorem proving), validation (by simulation), and automatic test case generation. More generally, the quality, certifiability, and maintainability of existing software should be gradually improved in this way. The primary objectives of the case study were that

- C code should be automatically generated from a formal specification,
- it should be possible to directly integrate the resulting C code in the existing software (without further human intervention or post-processing), and

– the generated code should perform exactly the same externally visible actions as the original one. (This objective should be validatable through inspection of the generated code by developers of the original C code.)

Thus, the project crucially involved re-engineering. This is certainly not an exceptional situation as we have ourselves experienced other cases that required re-engineering of often *completely* undocumented legacy code before. There exists also much safety-critical software for which only *informal* specifications are available.

3.3 System Interface

The interface of the system's most important procedure has the form,

```
void do_sensor(const SHARED_DATA *data_access)
```

where `SHARED_DATA` is a structure containing two other structures, one for data which must not be changed (read-only) and another for data which may be changed (write access) by the software; values of the writable structure are also read within the module, e.g., old values for comparison with the new values (before the old values are updated). In total, the two structures contain several dozen data fields, most of which are writable. It should be noted that the respective procedure parameter, `data_access`, serves for both input and output; there is *no formal* output parameter, although output is actually produced (as a "side effect.") Access control for the shared data is not implemented within the system in question.

4 Re-engineering Process

In order to re-engineer the PSS, we first converted its code into an abstract model in the language of a formal method (and the respective tool platform). Then, we tried to automatically generate code out of this model such that the result can again be integrated into the whole targeted software system.

There was no complete requirements specification available for the respective PSS except for a document that briefly explained a large number of parameters and modes of operation and gives a brief, pseudo-code outline of the required actions. The detailed specification had to be extracted from the provided C code (approx. 1K LoC).

To this end, we abstracted from the C code by means of the ASM method. This method is more suitable for reverse engineering as it allows for n-to-m refinement, where the number of algorithmic steps in the refinement (m) may actually be smaller than the original number of steps (n). B, in contrast, allows only for 1-to-1 refinement (through the definition of originally abstract submachines). On the other hand, tool support for B is much better than for ASMs, especially for verification and code generation, so we chose to use both methods to exploit their relative strengths where appropriate.

One of the important cornerstones of the specification process is the representation of requirements at various abstraction levels using the notion of refinement. By following this technique, requirements are easy to specify, analyze and implement. In this style of specification writing, requirements are incrementally added to the model until the model is detailed enough to be effectively implemented. If the refinement model of the method is flexible enough, such as that of the ASM method, it is possible to reverse this process for the sake of increasing abstraction.

It is not possible to directly translate an ASM model to a B model with an off-the-shelf tool or method. However, both B and the ASM method are state-based methods and corresponding models are similar enough to allow for manual translation with a high degree of confidence, provided that the theoretically more expressive ASM method is exploited only to the point where it remains compatible with the expressive capabilities of B.

Once the informal requirements were formally specified, the next step was to make sure that the requirements conformed to verification standards, i.e., requirements are consistent and verifiable. During this process, it was determined that a specification conformed to some precisely expressed properties that the model is intended to fulfill such as well-definedness, invariant preservation and other safety conditions. For verification of the model, we used two well-established approaches of theorem proving and model checking. The former helped us to reason about defined properties using a rigorous mathematical approach. The latter helped us to verify dynamic properties of the model by exploring its whole state space. While theorem proving is helpful in ensuring

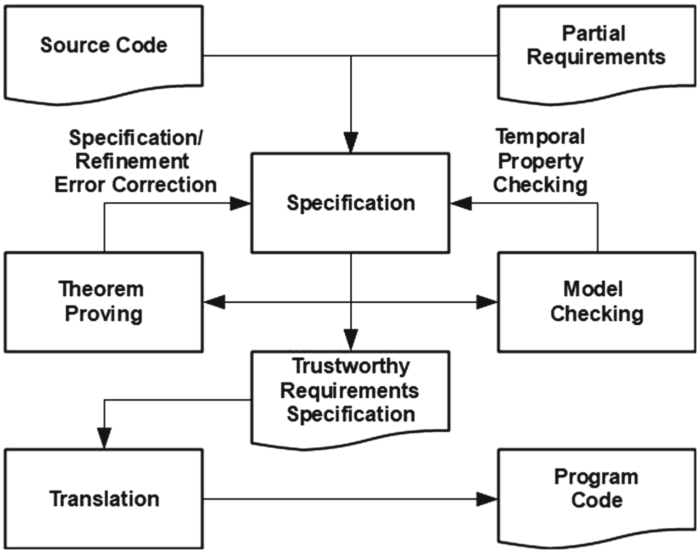
Source Code
Partial Requirements
Specification/ Refinement Error Correction
Temporal Property Checking
Specification
Theorem Proving
Model Checking
Trustworthy Requirements Specification
Translation
Program Code

Fig. 1. Model-driven re-engineering process

safety constraints of the system, model checking is effective in verifying temporal constraints of the system such as liveness and fairness properties and also help in validating the specification against requirements.

The last step of the formal development process is the translation of the requirements specification into program code. This last refinement step is, in fact, already very detailed and close to the implementation stage. The whole re-engineering process is depicted in Fig. 1.

As the main objective of the work was code generation, the rest of the paper will focus only on our experiences and challenges with the B method. However, for a detailed comparison between the ASM and B methods, please see [17].

5 Modeling Experience

In the following, we present our modeling experience with the B method. We first describe what we particularly liked about the method, and later, what was limiting about it.

5.1 Strengths of the B Method

Composition

Support for composition and decomposition in a modeling method is important for any domain of application when it comes to "larger-than-toy" systems [9]. Without decomposition, large complex models cannot be effectively over-viewed and handled. Decomposition is also of great value while proving the correctness of a large system. Composition is also important for model reuse.

Composition and decomposition are supported by the B method by allowing to call operations of other machines and to access, e.g., data structures from other machines. This works basically like calling procedures in procedural programming languages, but B additionally provides a few options regarding the visibility and accessibility of elements of other machines. Every machine has its own file.

Refinement

Refinement is a way of specifying the requirements of a complex system through a series of models for the same system with increasing depth of detail or, for reverse engineering, with increasing abstractness. Refinement is the central element in the B method. B defines a very powerful and well-supported refinement process. B supports a one-to-one notion of refinement. This is rather strict but eventually results in a higher degree of automatically discharged proofs. B allows to refine a model up to the level of detail required for implementation, or actually right down to programming code.

In practice, refinement relies very much on defining the actions of operations which can initially be the empty action, "`skip`." That is, in the operations of one machine one can call operations of other machines which may initially be left abstract. For instance, in this project, one machine may call another machine called "`update_shared_data`," for which the algorithmic definition remains "`skip`," i.e., abstract, because those details are not relevant for the specification or for the current level of abstraction.

Nondeterminism

Support for nondeterminism in a modeling method is very useful for keeping models abstract. B supports nondeterminism by allowing for nondeterministic choice of values for variables out of a given set (corresponding to Hilbert's ϵ operator) as well as by operators "`ANY`" (unbounded choice of value) and "`CHOICE`" (nondeterministic choice of alternative substitutions). In this fashion, new concepts can be added to specifications abstractly in the earlier refinements and can be concretized in the later ones. For example, in this project, we initialize a variable, potentially of a complex data type, with an unspecified value from this data type – as in "`l_d_sensor_input :: SENSOR_DATA`." Thereby no assumption about a concrete value is made other than that it is of type `SENSOR_DATA`.

Correctness Assurance

The possibility to express and prove properties, such as consistency, safety and temporal constraints (e.g., termination, deadlock freeness, fairness, and liveness), are integral to reason about the correctness of a safety-critical system. B enables the expression of typical safety properties through invariants. An invariant is a property that the specification is assumed to meet and maintain. Following is a sample requirement from the project:

> If the system is in the preparation mode or if the system is in the therapy mode and if the critical fluid temperature exceeds the maximum temperature of 41 °C, then the software shall disconnect the supply of the critical fluid within 60 s and execute an alarm signal.

The safety properties are specified in terms of invariants as follows:

$$\begin{array}{ll} \mathsf{inv1} & \mathsf{systemMode} = \mathsf{Preparation} \land \mathsf{criticalFluidTemperature} > 41 \Rightarrow \\ & \mathsf{systemState} = \{\mathsf{CriticalFluid} \mapsto \mathsf{Disconnected}\} \land \mathsf{disconnectionTime} < 60 \land \mathsf{alarm} = \mathsf{ALM1} \\ \mathsf{inv2} & \mathsf{systemMode} = \mathsf{Therapy} \land \mathsf{criticalFluidTemperature} > 41 \Rightarrow \\ & \mathsf{systemState} = \{\mathsf{CriticalFluid} \mapsto \mathsf{Disconnected}\} \land \mathsf{disconnectionTime} < 60 \land \mathsf{alarm} = \mathsf{ALM2} \end{array}$$

The support environment Atelier B generates POs to make sure that the system specification is well-behaved, i.e., maintains system invariants.

5.2 Limitations of the B Method

No Loops Except in Implementations

A restriction for abstract machines and refinements except for implementations – the last refinement steps before code generation – is that no loops are allowed (in implementations, `WHILE` loops are possible). While we did not need loops in our case study (so far), this restriction can certainly be critical.

Data Types

In a specification and an associated abstract model, usually very few data types are really needed. For this purpose, the data types provided by B (including booleans, integers, arrays, and structures) are largely sufficient. However, in an associated programming code – which also has to take into account efficient use of resources, amongst others –, we often need ultimately more such as a 16-bit integer, a 8-bit unsigned integer, and strings. In the context of the programming language C, *pointers* are also important constructs whose support is not present in typical modeling languages, so as B. In our case, already the modeling of the *interface* turned out to be a problem due to this nonexistent support of pointers.

Subsets

In B0 - the restricted B "dialect" required as a basis for code generation - all enumerated types have to be redefined as integer intervals. Arbitrary sets of integers are not allowed. But this precludes the definition of subsets (sub-types) whose members are not consecutive members of the base type. For instance, when we have a range of (named) colors as the base set, then we cannot create a greater number of arbitrary subsets of that – say, rainbow colors, RGB-colors, reddish colors, etc. – even with the most fancy ordering of the colors in the base set. (n.b., the result – an *interval* – is not a "set" in the mathematical sense any more because of the imposed ordering!) In general, the limit is two subsets.

In our case study, certain data types are used all of which may have the value `NO_DATA`. The definition of such types is not possible in B0 because (a) more than two of such otherwise disjoint types cannot be defined as sub-types of a common supertype (or `SET`), as stated above, and (b) `NO_DATA` can only be defined once and cannot be a member of different sets (which have to be disjoint when defined in `SETS`). (Note that we are only talking about enumeration sets here – with numerical types (e.g., `INT`), this is not possible at all.) A workaround for the "`NO_DATA`" problem is to define different constants for different types – `NO_DATA_X, NO_DATA_Y, ...` –, but this is not compatible with given interfaces.

Note that this problem only surfaces at the level of *implementations*, i.e., the last refinements before code generation; in B proper, arbitrary subsets can be defined (as constants).

Restrictions on Record Handling

Single record fields cannot be directly set via an operation call. Instead, an auxiliary variable has to be used (i.e., such an assignment requires two lines of code instead of one). We suspect that there may be further, similar restrictions for handling record fields; see also code generation problems regarding structures/records in the next subsection.

Identifiers

The language B imposes restrictions on identifiers: scalar parameters of machines must be lowercase only while set parameters must be uppercase only. While this had no direct influence on our case study (although the use of machine parameters might be considered in further development), this indirectly imposes the *convention* to use only uppercase names for sets in general, and only lowercase names for constants and variables, for instance. This, however, may clash with conventions of existing code (and did so in our case study).

No Pragmas

It seems impossible to model compiler directives (pragmas) in B. Pragmas might be seen as too implementation-specific constructs to earn a place in formal modeling, but they are frequently used in legacy code (including the one we dealt with; e.g. `#if 0 ... #endif`) and may be required when just single modules of a larger system are to be modeled. (Note that `#include` and `#define` statements are automatically generated by Atelier B from respective `IMPORTS` or `SEES` sections or from constant definitions, respectively.)

6 Code Generation Experience

In the following, we present our experience with code generation. For the case study, we used the community version of Atelier B including its code generator that is released for public use after every two years[4].

6.1 Strengths of the Atelier B Suite

The provided tool support is very good. Atelier B comes with code generators for different target languages, including C, C++, Java, and Ada. Although the generated code requires some post-processing, it is a good basis for the implementation of a B specification. The generated code is well-structured, well-documented and legible.

6.2 Limitations of the Atelier B Suite

Identifiers

In order to fit into a given interface, identifiers have to match exactly. However, this is not possible with the code generator of Atelier B. This code generator produces identifiers – most importantly, procedure names – as a combination of the machine name and the operation or set or constant name, separated by a double underscore. This is motivated by the need to avoid identifier clashes and

[4] According to ClearSy, they will fix some of the concerns raised in this paper in the upcoming version of the code generator.

related scoping problems (see [1, p. 6]). However, this renders it impossible to get procedure (and custom type) names prescribed by an existing interface.

This is a *crucial* problem in a project setting where only a part of an existing piece of software shall be transformed to code generated from a formal model. Manually rectifying all these names is tedious and error-prone. Automated post-processing would have to be performed separately for each project[5].

Structures and Records

The translation of structures (“`struct`” types) is actually faulty: the result *cannot be compiled* without post-processing (we did manual corrections).

In the B model, we had a single definition file in which all the `struct` types were defined. But in the generated code, any typing declaration with such a structure was individually expanded to the whole type definition with all fields and their types, and everywhere it was given a different name. In the interface of a procedure with two parameters of the same `struct` type, this `struct` was twice expanded and given two different names. The respective type names (`R_1`, `R_2`, ...) even differed between the header files and the corresponding definition files.

To mend this, in post-processing, one has to make the necessary `struct` definitions once and for all in some header file and then change every occurrence of this type to the respective type name. If one has large structures (in our case, with up to 36 fields), this is hard to automate, even using regular expressions, and error-prone. (The regular expression “`struct R_? {*}`” actually matches *any* occurrence of *any* structure type, and the number attached to `R_` does not give any indication as to which particular structure type is actually given in the respective place.)

Miscellaneous Observations

In B0, the language from which code can be generated, we discovered a strange restriction on expressions, i.e., for `IF` conditions: a statement of the form “`IF a < (b + c) THEN ...`” is not possible (though it is in B in general); instead, one has to add an auxiliary statement: “`aux := b + c; IF a < aux THEN ...`”[6].

An issue related with data type restrictions as well as with identifier restrictions is that the generated code requires the Standard Library for C. This may seem reasonable at first sight, but in practice, the Standard Library is not always used in industrial practice. This is a real problem in particular when the task is to obtain code which fits into given interfaces of a larger, existing system. In

[5] According to ClearSy, in the upcoming version of the code generator, custom identifiers (without prefix) would be possible. This would indeed constitute a major improvement and should be regarded as an important *and feasible* requirement for code generators.

[6] According to ClearSy, this construct eases the proving process.

such a case, there is no realistic possibility to change the libraries used or to rename types (or other identifiers). At the moment, further post-processing is required if, e.g., the Standard Library is not used or not supported in the target environment.

7 Related Work

Bert et al. [7], apart from our work, also critically evaluated the performance of the Atelier B platform in this direction and have suggested several improvements. An experience report involving Atelier B by Beneviste [6] briefly mentions code generation for VHDL. However, both of these aforementioned works were not really useful for our case study.

Event-B is a variant of B for higher-level specification, verification and validation of systems and environments where software systems are supposed to operate. We, alternatively, tried to generate code from an Event-B model of the pressure sensing system but failed. It turned out that none of the available code generators for Event-B was usable for our purpose. The one by Wright [30] was custom-built and only supports a part of the Event-B syntax. The most significant shortcoming is that it does not support contexts and therefore cannot be used when constants and sets are used in a model. The one by Fürst et al. [13] is not publicly available, thus we could not use it at all. EB2ALL [26] explicitly requires the manual alteration of code after generation. This is contradictory to one of the objectives of the case study. The Tasking Event-B tool [11], which appears to be the most mature of all, is compatible only until Rodin 2.8 (the current Rodin version at the time of writing this paper is 3.2) and may only work properly on 32-bit machines; however, such a machine was not available for the case study.

8 Conclusion

The use of formal methods is "highly recommended" for safety-critical software by international standards and can actually also be economic in the long run for other kinds of software. Yet there are still a couple of hurdles for a more widespread use of formal methods in industrial software development, which have to be addressed one by one. Two such issues are the extra effort invested in formal modeling and the potentially unsafe transition from the specification to the implementation. Both issues can be tackled by means of automated code generation from formal models, i.e., from those models which are used in formal specification and analysis.

We have put the current state-of-the-art tool support for code generation to test with a case study drawn from real industrial software development. We have chosen B for this purpose because it is well-suited for ordinary software development, is well-known also in parts of industry, supposed to be mature, has commercial tool support including code generation, and there was already expertise available within the team.

The results of our practical investigations are mixed. First and foremost: yes, generating code from B models, using B's standard tool Atelier B, *does* work. However, there are several restrictions which we have encountered, ranging in severity from inconvenient to critical. Some of these restrictions not only concern code generation, but also modeling in B.

We believe that the necessary improvements are feasible with reasonable effort. For the scenario which the formal methods community typically envisages, i.e., where one starts with a completely new development from requirements engineering via formal specification and analysis to design and coding (etc.), code generation already works quite well when we take aside type restrictions, the problems with records which we encountered, as well as efficiency issues.

But for a scenario which involves re-engineering and improving a single module out of a large system by means of (e.g.,) the B method, there are still major shortcomings, as we have described in detail in this paper. Note, however, that other application examples are likely to identify more issues; to name just a few examples, we did so far not touch upon loops, recursive function calls, float (real) types, or strings. Some of the necessary improvements will probably require more customization of the tool. This concerns enhanced support of (custom) types in particular. It should be noted, though, that the tool is still being improved and certain issues may even have been solved by the time this paper is actually published, or will be solved in the near future.

According to ClearSy [19], almost all safety-critical products using B have their own code generator because of the constraints imposed by the platform running it. However, this further confirms our impression that off-the-shelf code generation is still an open issue (which probably holds for other formal methods as well).

A factor for the successful application of a particular method which cannot be neglected is the existence of a dynamic community wherein experiences can be shared and issues can be discussed. The B method has rich tool support and a sizable community in this regard. On the other hand, more popular methods with larger and more diverse communities, such as Event-B and TLA+, are either not universally applicable, or do not support code generation, or both. This suggests that a new impetus is needed for general-purpose formal methods which support the whole software development cycle, from specification and analysis to code generation and test suite generation to maintenance and versioning/product family development.

Still, it must be noted that, in principle, most important technologies and tools are there and do work. What is desirable now is further improvement and consolidation of these methods and tools.

We end with final remarks for practitioners. When starting to use formal methods in software development, the goals of their introduction must be clear and expectations need to be realistic. Furthermore, it is important to select a suitable method for the chosen goals and with respect to the given organizational environment and similar parameters. It should also be considered to introduce formal methods *incrementally* so as to minimize impact on development time and costs at each step and thus lowering the economic and psychological thresholds which cannot be completely avoided at the beginning. This article may help to assess what can be expected from the current state-of-the-art of some important aspects of formal methods. This article also explicitly addresses the scenario of incremental introduction, in particular, using formal methods for selected modules out of a larger system.

Acknowledgment. Thanks to Thierry Lecomte (ClearSy) for providing feedback that helped a lot to improve the quality of the presented work.

References

1. Atelier B Translators: User Manual version 4.6. http://tools.clearsy.com/resources/documents. Accessed 28 Feb 2018
2. Abrial, J.R.: The B Book. Cambridge University Press, Cambridge (1996)
3. Abrial, J.R.: Formal methods in industry: achievements, problems, future. In: Proceedings of the 28th International Conference on Software Engineering ICSE 2006, pp. 761–768. ACM, New York (2006)
4. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
5. Arcaini, P., Bonfanti, S., Gargantini, A., Mashkoor, A., Riccobene, E.: Formal validation and verification of a medical software critical component. In: 13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, pp. 80–89. Austin, 21–23 September 2015
6. Benveniste, M.: On using B in the design of secure micro-controllers: an experience report. Electron. Notes Theor. Comput. Sci. **208**, 3–22 (2011)
7. Bert, D., Boulmé, S., Potet, M.-L., Requet, A., Voisin, L.: Adaptable translator of B specifications to embedded C programs. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 94–113. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_7
8. Börger, E., Stark, R.F.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag New York Inc., Secaucus (2003)
9. Clarke, E.M., Wing, J.M.: Formal methods: state of the art and future directions. ACM Comput. Surv. **28**(4), 626–643 (1996)
10. Daskaya, I., Huhn, M., Milius, S.: Formal safety analysis in industrial practice. In: Salaün, G., Schätz, B. (eds.) FMICS 2011. LNCS, vol. 6959, pp. 68–84. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24431-5_7
11. Edmunds, A., Butler, M., Maamria, I., Silva, R., Lovell, C.: Event-B code generation: type extension with theories. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 365–368. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30885-7_33
12. Fitzgerald, J.S., Larsen, P.G.: Triumphs and challenges for model-oriented formal methods: the VDM++ experience. In: Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2006), pp. 1–4, November 2006
13. Fürst, A., Hoang, T.S., Basin, D., Desai, K., Sato, N., Miyazaki, K.: Code generation for Event-B. In: Albert, E., Sekerinski, E. (eds.) IFM 2014. LNCS, vol. 8739, pp. 323–338. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10181-1_20

14. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2006)
15. Jones, C.B.: Systematic Software Development Using VDM, 2nd edn. Prentice-Hall Inc., Upper Saddle River (1990)
16. Kossak, F.: Landing gear system: an ASM-based solution for the ABZ case study. In: Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. CCIS, vol. 433, pp. 142–147. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07512-9_10
17. 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
18. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
19. Lecomte, T.: Atelier B has turned twenty. In: Keynote of the Fifth International Conference on ASMs, Alloy, B, TLA, VDM, and Z (ABZ 2016). Springer, Heidelberg (2016)
20. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. J. Softw. Tools Technol. Transf. **10**(2), 185–203 (2008)
21. Mashkoor, A.: The hemodialysis machine case study. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 329–343. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_29
22. Mashkoor, A.: Model-driven development of high-assurance active medical devices. Softw. Q. J. **24**(3), 571–596 (2016). https://doi.org/10.1007/s11219-015-9288-0
23. Mashkoor, A., Biro, M.: Towards the trustworthy development of active medical devices: a hemodialysis case study. IEEE Embed. Syst. Lett. **8**(1), 14–17 (2016)
24. Mashkoor, A., Biro, M., Dolgos, M., Timar, P.: Refinement-based development of software-controlled safety-critical active medical devices. In: Winkler, D., Biffl, S., Bergsmann, J. (eds.) SWQD 2015. LNBIP, vol. 200, pp. 120–132. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-13251-8_8
25. Mashkoor, A., Hasan, O., Beer, W.: Using probabilistic analysis for the certification of machine control systems. In: Cuzzocrea, A., Kittl, C., Simos, D.E., Weippl, E., Xu, L. (eds.) CD-ARES 2013. LNCS, vol. 8128, pp. 305–320. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40588-4_21
26. Méry, D., Singh, N.K.: Automatic code generation from Event-B models. In: Proceedings of the Second Symposium on Information and Communication Technology SoICT 2011, pp. 179–188. ACM, New York (2011)
27. Miller, S.P., Whalen, M.W., Cofer, D.D.: Software model checking takes off. Commun. ACM **53**(2), 58–64 (2010)
28. Reicherdt, R., Glesner, S.: Formal verification of discrete-time MATLAB/Simulink models using boogie. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 190–204. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10431-7_14
29. Spivey, J.M.: Understanding Z: A Specification Language and Its Formal Semantics. Cambridge University Press, Cambridge (1988)
30. Wright, S.: Automatic generation of C from Event-B. In: Workshop on Integration of Model-based Formal Methods and Tools (2009)