Towards Incremental Validation of Railway Systems*

Reiner Hähnle[1,2] and Radu Muschevici[1]

[1] Department of Computer Science
Technische Universität Darmstadt, Darmstadt, Germany
{haehnle|radu}@cs.tu-darmstadt.de
[2] Department of Computer Science
Università degli Studi di Torino, Torino, Italy

Abstract. We propose to formally model requirements and interoperability constraints among components of a railway system to enable automated, incremental analysis and validation mechanisms. The goal is to provide the basis for a technology that can drastically reduce the time and cost for certification by making it possible to trace changes from requirements via design to implementation.

1 Introduction

Railway systems have stringent safety, reliability and availability requirements. Verification and certification are central aspects of system development in the railway domain.

The ongoing *digitalization* of all technical systems affects the railway domain in a manner similar to other systems: an increasing number of functionalities that used to be realized by dedicated hardware are now implemented in software on standard platforms; in addition, these software systems become connected, raising completely new demands regarding usability and security. As a consequence, technical systems become affected by changes in the technological infrastructure, legal regulations and user behavior. While this global trend affects most technical domains, the railway domain is particularly sensitive because of its high quality and safety requirements, which necessitate certification.

It has been pointed out by railway technology suppliers [28] that the expected increase in effort for frequent (re-)certification is about to become a major problem for the railway industry. The reason is that, while railway systems evolve incrementally throughout their lifetime, their validation typically involves re-verifying the whole system. That is, whenever a part of the system changes, the entire system is again subjected to full verification to renew its authorisation

* The research reported here was partially supported by project FormBar "Formalisierung von betrieblichen und anderen Regelwerken", part of AG Signalling/DB RailLab within the Innovation Alliance between Deutschen Bahn AG and TU Darmstadt. Parts of this paper were written during a research stay of the first author at and supported by University of Torino. All support is gratefully acknowledged.

for placing it in service. An *incremental* verification approach would substantially decrease the cost and effort of renewing this authorisation.

In this paper, we propose an incremental certification approach based on a formal modeling language. It consists of the following components:

- *Modeling* of the railway domain, its operative constraints, and its safety regulations in a formal, executable language that allows to represent system variability in an explicit, traceable manner.
- The model is sufficiently precise to permit *code generation* for the intended implementation platform. The code generator needs to be certified only once and for all.
- System *updates* are modeled as new system variants that change or extend some of the existing features. Updates are *traced* from requirements via a feature model to the executable model. The difference between the given model and the updated model is captured in a structured manner in terms of so-called "deltas".
- A structured description of a system update in a formalized language is the basis for advanced *formal methods tools* support: behavioral regression analysis, automated test generation, resource analysis, correctness checks such as deadlock freedom, are available to speed up certification and decrease workload.

The viability of the process sketched above hinges on the availability of a suitable modeling language and tool suite. Such a language is presented in the following section.

2 The Abstract Behavioral Specification (ABS) Language

The Abstract Behavioral Specification (ABS) language [24] is a formal, executable modeling language that has been developed in a series of EU projects since 2009. It comes with a comprehensive tool suite [9,35] that is integrated into a web-based IDE and also available as an Eclipse plugin. ABS was used successfully in industrial projects [7]. In particular, it is currently used in a joint project[3] between Deutsche Bahn Netz AG and TU Darmstadt to formally model and analyze the operation of trains.

In contrast to general purpose formal specification languages such as B [1] or Event-B [2] that are based on set theory, ABS can be seen as a modern programming language based on an ADT—functional—OO—actor paradigm. The core of ABS appears somewhat like a light version of the Scala language but with a less complex syntax. ABS is type-safe, free of data races, and has a formal semantics [3]. It has been carefully designed to admit scalable static analyses of various kinds. As ABS is not based on a mathematical notation but on mainstream programming language idioms, it is easy to learn for anyone with programming knowledge. In this paper, we introduce ABS by example. We refer to the ABS language specification [4] and a tutorial [20] for details.

[3] `http://www.formbar.raillab.de`

3 Modelling the Railway Domain in ABS

The operative rules of the German Railways for running trains [13] contain a very thorough domain analysis with highly precise definitions of all relevant elements of the railway domain: points, track, signal, train, station, etc. Thanks to the object-oriented modeling capabilities of ABS, these can be rendered in a rather natural OO model almost one-to-one. The ABS snippet in Fig. 1, which models a tiny part of *Fahrdienstvorschrift Richtlinie 408*[4] [13], is nearly self-explanatory.

```
1 module DomainElements;
2 data PointsPosition = Left | Right;
3
4 interface Points {
5   Unit change(PointPosition setting);
6 }
7
8 class Points implements Points {
9   PointsPosition position = Left; // default setting
10  Unit change(PointsPosition setting) { this.position = setting; }
11 }
```

Fig. 1. ABS implementation of railway points

As can be seen, classes and interfaces may have the same name. ABS enforces a strict "programming-to-interfaces" discipline where only interfaces define types, so that no ambiguity arises.

ABS is a concurrent language supporting both shared-memory as well as distributed (actor-based) concurrency. Inter-object communication is realized with messages (i.e., method calls). This is a good fit to the train domain where all safety-relevant events are triggered by explicit communication that happens at precisely defined points.

In contrast to thread-based concurrency with interleaving semantics, as found in languages such as Java or C, ABS is based on *cooperative scheduling*. This means that an ABS process cannot be preempted unless its designer explicitly states it. This is a good match with the kind of concurrency encountered in the railway domain: trains, controllers, track elements are modeled naturally as different actors that do not communicate by shared resources, but with explicit messaging events. In addition, there is at most one train at each position of a track; trains can enter block sections only at explicit locations marked by signaling elements, etc.

[4] Richtlinie 408 is the official document that regulates in detail how trains are to be operated in Germany. Similar documents exist in other countries.

We explain the ABS concurrency model with an example, shown in Fig. 2, that gives a fragment of a model of train runs. It is encapsulated in an interface `TrainRun`, of which we give an implementation in an identically named class.

```
1 class TrainRun(Int trainNr, Route route) implements TrainRun {
2
3   BlockPost position; // end of current block section
4
5   Unit run() {
6     BlockPost position = route.getStart();
7     Fut<Unit> entered = position!enterTrain(this);
8     BlockPost destination = route.getDestination();
9     await entered?;
10    while (position != destination) {
11      BlockPost next = route.getNext();
12      ...
13      position = next;
14    }
15  }
16 }
```

Fig. 2. An ABS model of train runs

Each train run is parametrized with a unique identification number and the route it is supposed to take. How routes are modeled is not of interest here and is encapsulated in the `Route` interface. If the implementation of routes is ever changed, then the rest of the model is not affected. For simplicity of presentation, we also do not model elapsed time, schedules, delays, and many other elements.

At each time a train run has a position, by definition the block post (signaling element) that marks the end of the block section it occupies. The method `run` models how a train is run along a given route.

In line 6, a synchronous (i.e., sequential and blocking) method call obtains the initial position of the train. However, the train might still be elsewhere (e.g., in the previous block section), so we must ensure that it has arrived at the current position before it is dispatched further. We do not know exactly how long this takes. Therefore, it is appropriate to model the action with an asynchronous call to `enterTrain`. This call encapsulates a number of complex actions including updating several tables, logging events, etc. In ABS, this asynchronous call *does not* interrupt the execution of `run`, which directly proceeds with line 8. The type of the result variable `entered` is annotated as a *future* value that is not immediately available. If the call to `enterTrain` is executed on a different processor, then it can run concurrently to the `run` process.

Before the loop is entered, where the train is advanced along the route, we must ensure that it has arrived at the current position, This is achieved by the

await statement in line 9 which suspends execution of `run` until its guard becomes true. An expression of the form `f?`, where `f` is a future, becomes true once the call associated with it has completed. Observe that we are not interested in the return value of the call, only in the fact that it has finished.[5]

Cooperative scheduling with explicit **await**s avoids data races. In particular, it is sufficient that a method reestablishes the class invariant before suspension and upon termination—these are the only opportunities for other methods to modify the state of a process. As a consequence, it was possible to design a *modular* proof system for functional verification of ABS models [15], where global invariants are decomposed into locally sequential properties. This is the prerequisite to formally prove safety properties of ABS models for an *unbounded* number of objects and *unbounded* data [16].

It is possible in ABS to model the passing of time, making it suitable for real-time simulation. For example, if we want to model that a train spends a certain amount of time within each block section, we could insert a statement of the form `duration(t,t')` at the beginning of the while loop before line 11. It has the effect to block the current process for at least `t` and at most `t'` time units. If we want the processor not to be idle during this period, then we could write **await** `duration(t,t')` to allow suspension of control. Time parameters such as `t` might be computed by complex expressions. Assume that expression `e` computes the minimum time it takes a train to pass through the current block section. Then we could set `t = e;` before the duration statement. The current system time can be queried with the built-in function `now()` in ABS. This makes it possible, for example, to model in a natural manner that a train waits at least until its scheduled departure time, etc. A global clock, which tends to clutter models and render simulation expensive, is not required.

To finish with the example, the while loop (without giving details) steps through the `route` until the destination is reached. Each loop body contains various asynchronous calls that implement the protocol governing how a train passes from one block section to the next.

As mentioned above, the concurrency model of ABS is a good match for the kind of concurrency encountered in railway systems: in either, actions are asynchronous (e.g., time passes between the signal going green and the train starting to move) and scheduling points are explicit (e.g., signals). It is tedious and complex to model concurrency in general purpose specification languages. This motivated, for example, CSP||B [30] that combines stateful and event-based modeling and is used to formalize railway interlocking [23]. But while CSP||B is a combination of two different formalisms, in ABS data modelling, imperative-OO stateful programming, and asynchronous events are tightly integrated into a uniform high-level language that nonetheless stays formally analyzable.

[5] Of course, it is also possible to retrieve the value of a finished call by accessing a future with `f.get`.

4 Variability Modelling with ABS

The ABS language can model system updates as new system variants that change or extend existing variants. Following a *feature-oriented software development (FOSD)* approach [8], variants are described as sets of *features*. Each feature has its corresponding ABS implementation. A feature's implementation is specified in terms of the code modifications (i.e., additions and removals) that need to be performed to a variant of the system that does not include that feature in order to add it. As the modeler needs to specify differences between system variants, this style of programming is called *delta-oriented programming* [29]. The ABS code modules that encapsulate feature implementations are called *delta modules*, or *deltas* for short. Deltas can modify a given ABS program extensively and comprehensively by adding, removing or modifying program elements, including interfaces, classes, algebraic data types and functions.

The ABS code in Fig. 3 shows a delta that extends points with a configurable maximal traversal speed. Historically, points were designed to be safely traversed at low speeds of around 40–100 km/h. Contemporary point designs embrace the demands of high speed railways and allow trains to pass them at higher speeds of 200 km/h or above.

```
1  delta ConfigurableSpeedPoints(Int maxPointsSpeed);
2  uses DomainElements;
3
4  modifies interface Points {
5      adds Unit setMaxSpeed(Int s);
6  }
7
8  modifies class Points {
9      adds Int  maxSpeed = maxPointsSpeed;
10     adds Unit setMaxSpeed(Int s) { maxSpeed = s; }
11 }
```

Fig. 3. An ABS delta implementing the feature "points with a configurable speed"

The delta `ConfigurableSpeedPoints` modifies the `Points` class by adding a private[6] field `speed` and a setter method to change its value. It also adds the `setSpeed` method to the points' public interface; otherwise the method would be private to its class (because of ABS' "programming-to interfaces" discipline). The points' default maximal speed value is assigned based on the delta's `maxPointsSpeed` parameter. To understand how this delta parameter is configured we need to introduce the feature modelling facility of ABS.

[6] ABS enforces strong encapsulation of object states. All fields are strictly private and the only way to access them from other objects is by getter and setter methods.

A *feature* is a familiar but informal notion in software engineering—it is quite common to describe a software system in terms of the features that it offers. FOSD uses features in a more rigorous sense: features are derived from software requirements and their relationships are formally captured in a *feature model* [25]. ABS feature models define features as names (labels) and also allow feature attributes, which are name-value pairs that make features parametrizable.

A feature model describes a system at a high level of abstraction: its specification relates to the requirements engineering phase in a software development process. Delta modeling, that is, the declaration of deltas and their relationships, is part of the design and implementation phase.

The crucial point is that feature model and delta model are *connected*: a feature is implemented using a series of one or more deltas while a delta supplies the (partial) implementation for one or multiple features. Hence, there is a many-to-many relation between features and deltas.[7] This flexibility makes it possible, on the one hand, to avoid pollution of the feature model with implementation aspects and helps to realize a modularization among deltas, on the other.

Importantly, the connection between features and deltas in ABS is *explicit*: we can trace a feature to its delta-based implementation, and we can trace deltas back to the features that they implement. Traceability between feature and delta model allows *automatic generation* of system variants: upon selecting a set of features, the corresponding software product can be generated automatically by tracing the corresponding delta modules and composing them. This is explained in the next section.

5 Modeling Railway Systems Evolution with ABS

To specify variability in the railway domain, we introduce a feature model. This model reflects an evolution scenario: the system model described in Section 3 evolves over time to reflect actual developments in the railway domain, such as the introduction of points traversable at higher speeds. Instead of changing the point's implementation (see Fig. 1) directly, we associate the capability of traversing points at variable maximum speeds with a new *feature* `ConfigurableSpeedPoints`. Then we specify the difference in terms of implementations between historical points and modern points by providing the delta shown in Fig. 3.

The advantage of this approach from a validation/certification standpoint is that the old system continues to exist in its *original form* and therefore does not need to be re-validated. Merely the changes applied to the old system, which are encapsulated in the new features and deltas, need to be re-validated/certified. This is the basis for an *incremental* approach to system validation. By implementing system evolution in this way, we effectively turn our railway system specification into a software product line (SPL) [27]—a *set* of system specifications that are developed together from a collection of common software assets.

[7] Due to the many-to-many association of features and deltas, it is generally not feasible to rely on feature models to abstractly describe delta composition – other modeling tools, such as an extended UML [32] are preferable for this purpose.

```
root RailwayDomain {
  group allof {
    opt ConfigurableSpeedPoints {
      Int max;
    }
  }
}
```

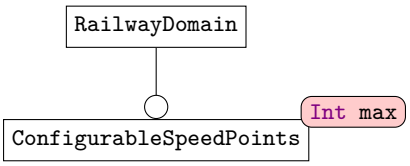

Fig. 4. A simple feature model for the railway domain model

Fig. 4 illustrates the fragment of an ABS feature model associated with the evolution scenario described above. On the left is a textual description and on the right an equivalent, diagrammatic one.

A feature model consists of a hierarchy of features, with the most general feature that describes the overall concept (here: `RailwayDomain`) at the top. Below the top feature, an arbitrary number of sub-features can be declared; these are specified as a **group**. Sub-features can in turn have their own sub-features and so on. In our example, the only sub-feature of `RailwayDomain` is `ConfigurableSpeedPoints`. Moreover, this feature is declared **opt**ional to reflect that it does not have to be part of all variants of the railway domain system (in the graphical representation optional sub-features are marked with a hollow circle). The feature has an attribute `max` that represents the maximum traversal speed of points in the domain. By designing system variants that assign specific concrete values to this attribute, the points in our system will have different capabilities.

To specify different system variants, the ABS language provides **product** declarations. A product is simply a set of features and feature attribute assignments that are given a name. Fig. 5 shows the declaration of four products.

```
product Legacy(RailwayDomain);
product LowSpeed(RailwayDomain, ConfigurableSpeedPoints{max=40});
product MedSpeed(RailwayDomain, ConfigurableSpeedPoints{max=130});
product HighSpeed(RailwayDomain, ConfigurableSpeedPoints{max=230});
```

Fig. 5. A set of products (variants) of the railway domain system

The `Legacy` product includes only the `RailwayDomain` feature while the other three also include the `ConfigurableSpeedPoints` feature specifying different values for the maximum traversal speed attribute.

So far we have shown how to declare deltas that specify modifications to an ABS program without deltas. We have also shown how to specify feature models that model system variability abstractly, using features and attributes. We still lack the connection between both, a component that is essential in establishing

```
1 productline RailwayDomain;
2 features RailwayDomain, ConfigurableSpeedPoints;
3 delta ConfigurableSpeedPoints(ConfigurableSpeedPoints.max)
4       when ConfigurableSpeedPoints;
```

Fig. 6. An SPL configuration for the railway domain SPL

the desired traceability between features and delta models. The construct that provides this link in ABS is called *software product line (SPL) configuration*. An SPL configuration specifies the (many-to-many) relation between the features in a feature model and the deltas in a delta model. Based on the set of features in a specific product, the configuration defines which delta modules should be applied to build the corresponding system variant.

Fig. 6 shows the SPL configuration for our running example. In lines 3–4, it is specified that the *delta* `ConfigurableSpeedPoints` should be applied **when**ever the *feature* `ConfigurableSpeedPoints` is selected. As a consequence, we are able to trace the `ConfigurableSpeedPoints` feature to its implementation by the `ConfigurableSpeedPoints` delta and vice versa.[8]

Looking at the product declarations in Fig. 5, we can see that the bottom three products include the feature `ConfigurableSpeedPoints`. Each of these products assigns a specific value to the feature's `max` attribute. Thanks to the SPL configuration, the delta's `maxPointsSpeed` parameter is initialized using that value, which then becomes the default value for the `maxSpeed` field introduced in the `Points` class by the very same delta (cf. Fig. 3).

6 Towards Automated and Incremental Certification

The example in Fig. 6 demonstrates how ABS supports traceable connections between features and their implementation in terms of deltas, where the latter describe code modifications in a precise and structured manner. This capability can be used to make certification after updates more cost efficient.

Updating and renewal of railways equipment is costly, partly due to the necessity to ensure continuing safety and interoperability of systems throughout their lifespan. Renewal or updating can lead to the loss of the "authorisation for placing in service", which subsequently has to be renewed by following a verification procedure.[9] That verification procedure, i.e., the effort required to ensure that the equipment continues to conform to the requirements (safety, performance, interoperability, etc.) is a major cost factor due to its manual nature. The decision whether and to what extent a verification procedure is

[8] For sake of simplicity, our example constitutes a one-to-one relation between a feature and a delta but in general it is many-to-many.

[9] See Directive 2008/57/EC, specifically Annex III which spells out essential requirements relating to safety, reliability, availability, etc.

necessary is taken on a case by case basis, based on manual assessment of the impact of each change.

The model-centric approach to specification of railway systems with ABS sketched in the previous sections has the potential to substantially reduce the cost of updates, with respect to both assessment and verification. This expectation is based on the capabilities of our approach in terms of *traceability*, *code generation*, and *automated analysis*. For each of these, we now describe its detailed role within the overall picture.

Traceability. Subsequent stages of system evolution are specified in terms of a software product line defined on the basis of a feature model. As the definition of SPLs in ABS permits traceability of features through to code deltas, one can precisely pin down the differences between two product variants (i.e., a legacy system and its update) in terms of implementation within an SPL as illustrated by the `ConfigurableSpeedPoints` update in the previous section.

The ability to present the changes between two system variants in a precise and structured manner not only simplifies the assessment of the impact of a change, but, importantly, even narrows down the scope of assessment to the exact *increment* embodied in the change.

Code generation. ABS code is executable and highly precise, which not only permits code generation, but results in *efficiently* executable code. At this time, ABS supports code generation backends to Java 8 [31], Haskell, Erlang, and ProActive [21]. All these code backends are highly optimized for concurrent execution on multi-processor systems.

To test this claim, we created a prototypic ABS model of railway operations based on [13] for a simple track layout that corresponds to a typical medium-sized station with 5–10 trains running in parallel. We chose 5m as its spatial resolution and 50ms as temporal resolution (corresponding to a maximum of 360km/h). We were able to simulate several hours of operation with dozens of train runs within a few seconds on a standard laptop.

In addition to simulation, a major advantage of code generation is that verification and certification of code generators can be performed completely independent from system modeling. As a consequence, certification of code generation needs to be done only once and then can be re-used for all subsequent system updates.

Automated analysis. Verification of a system update requires to ensure that (i) existing system properties are preserved, (ii) the new features work as intended, and (iii) no undesired properties are introduced by the update. This is a highly complex and time consuming, therefore, expensive task. It is here that the model-centric approach to system validation sketched in this paper offers its greatest potential. The reason is that the ABS language was explicitly designed to be analysable by highly automated tools [9] and is delivered with a comprehensive tool suite [35].

Regarding verification of system updates we distinguish between functional ("behavior") and non-functional ("performance") properties. In addition, we look at global correctness properties of ABS code, such as deadlock-freedom, uncaught exceptions, etc.

Functional properties are established by two complementary approaches: symbolic execution and automated test generation. Symbolic execution [22] of the legacy and the updated system is useful to compare control flows, which can then be checked for inaccurate modeling or hidden assumptions. In addition, test cases can be automatically generated from ABS code [6]. With regard to non-functional properties, we rely on automated *resource analysis* of ABS models [5]. Finally, it is important to ensure that the updated system does not introduce unwanted behavior in the form of deadlocks, uncaught exceptions, etc. To this end, it is important to ensure *correctness* of ABS code which is done either by deductive verification [14] or by type-based inference systems [19]. Importantly, these verification methods also support incremental changes [10] and, specifically, deltas [34].

Limitations. Like any modeling framework, also the approach based on SPL and ABS, has the limitation to cover merely those parts of the real world that have been modeled. Not all aspects of the train domain are suitable for modeling in ABS. This is the case, for example, with respect to usability/UIs, optimization, or continously evolving states. Some aspects, such as usability, are inherently difficult to formalize and our approach does not extend to them. Other aspects, such as optimization, are not the focus of (re-)certification and need not to be modeled. Or they could be modeled in a different, more suitable formal framework. For example, the system KeYmaera [18] is able to prove safety properties [26] of a hybrid model of the European Train Control System. These properties, for example, maximum break distance, can then be assumed and trusted in the ABS model.

7 Related Work

Refinement calculi such as Z, B, and Event-B are frequently used in the railway domain [28] because one can refine a declarative specification into a provably correct, executable program. This may involve a high number of refinement steps which typically results in considerable verification effort. Refinement-based approaches support a limited number of target languages (e.g., OCaML in the case of Event-B). While it is possible to model distributed concurrency [11,23], shared memory access, such as in ABS, seems problematic. The largest drawback of refinement-based approaches we see is that they do not support incremental system updates. The term *incremental* in connection with refinement-based models (e.g., [11,12]) refers to refinement steps. General system updates, however, are often not strictly refinements of data or behavior but modifications or extensions.

There is considerable work on formalization of different aspects of railway systems (see [17] for a recent collection). Unsurprisingly, much of it is concerned

with safety-critical aspects such as interlocking systems. In addition to B and Event-B, Petri nets are commonly used [33]. These have the limitation that it is not feasible to model and analyze a large or unbounded number of objects (e.g., train runs).

We are not aware of a formal model that attempts to formalize signaling as well as operational aspects of railways at the same time. We are also not aware of a formal model that supports incremental system updates that are not refinements.

8 Conclusion

In this paper, we sketched an approach for formal modeling, simulation, and analysis of railway systems, their requirements, and their interoperability constraints. It is based on the modeling language ABS that permits precise, executable specifications from which efficient code generation is possible. ABS permits to trace system updates from requirements down to the implementation via delta modeling, thus allowing to analyse functional and non-functional properties that may have changed. We showed that the specification of system updates in terms of software product lines in ABS permits traceability of features down to code. Together with the analysis tool suite of ABS, this forms a feasible technological basis for an incremental verification and certification process.

References

1. Abrial, J.R.: The B Book: Assigning Programs to Meanings. Cambridge University Press (Aug 1996)
2. Abrial, J.: Modeling in Event-B — System and Software Engineering. Cambridge University Press (2010)
3. Deliverable 1.2 of project FP7-231620 (HATS): Full ABS Modeling Framework (Mar 2011), `http://www.hats-project.eu`
4. The ABS Language Specification (2016), `http://abs-models.org/documentation/manual/`
5. Albert, E., Arenas, P., Flores-Montoya, A., Genaim, S., Gómez-Zamalloa, M., Martin-Martin, E., Puebla, G., Román-Díez, G.: SACO: static analyzer for concurrent objects. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 20th Intl. Conf., Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, Grenoble, France. LNCS, vol. 8413, pp. 562–567. Springer (2014)
6. Albert, E., Arenas, P., Gómez-Zamalloa, M., Wong, P.Y.H.: aPET: a test case generation tool for concurrent objects. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Joint Meeting European Software Engg. Conf. and ACM SIGSOFT Symp on Foundations of Software Eng., ESEC/FSE, St Petersburg. pp. 595–598. ACM (2013)
7. Albert, E., de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L., Wong, P.Y.H.: Formal modeling of resource management for cloud architectures: An industrial case study using Real-Time ABS. Journal of Service-Oriented Computing and Applications 8(4), 323–339 (Dec 2014)

8. Apel, S., Kästner, C.: An overview of feature-oriented software development. Journal of Object Technology 8(5), 49–84 (Jul 2009)
9. Bubel, R., Flores Montoya, A., Hähnle, R.: Analysis of executable software models. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E.B., Schaefer, I. (eds.) Executable Software Models: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinoro, Italy. LNCS, vol. 8483, pp. 1–27. Springer (Jun 2014)
10. Bubel, R., Hähnle, R., Pelevina, M.: Fully abstract operation contracts. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation, 6th International Symposium, ISoLA 2014, Corfu, Greece. LNCS, vol. 8803, pp. 120–134. Springer (Oct 2014)
11. Butler, M.: Incremental design of distributed systems with event-b. In: Broy, M., Sitou, W., Hoare, T. (eds.) Engineering Methods and Tools for Software Safety and Security: Marktoberdorf Summer School 2008, chap. 4, pp. 131–160. IOS Press (2009), `http://eprints.soton.ac.uk/266910/`
12. Butler, M.J., Yadav, D.: An incremental development of the Mondex system in Event-B. Formal Aspects of Computing 20(1), 61–77 (2008)
13. Deutsche Bahn Netz AG, Frankfurt, Germany: Fahrdienstvorschrift Richtlinie 408 (Dec 2015), `http://fahrweg.dbnetze.com/fahrweg-de/nutzungsbedingungen/regelwerke/betriebl_technisch/eiu_interne_regeln_ril_408.html`
14. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In: Felty, A., Middeldorp, A. (eds.) Proc. 25th Intl. Conf. on Automated Deduction (CADE), Berlin, Germany. LNCS, vol. 9195, pp. 517–526. Springer (2015)
15. Din, C.C., Owe, O.: Compositional reasoning about active objects with shared futures. Formal Aspects of Computing 27(3), 551–572 (2015)
16. Din, C.C., Tapia Tarifa, S.L., Hähnle, R., Johnsen, E.B.: History-based specification and verification of scalable concurrent and distributed systems. In: Butler, M., Cochon, S., Zaïdi, F. (eds.) Proc. 17th International Conference on Formal Engineering Methods, ICFEM, Paris. LNCS, vol. 9407, pp. 217–233. Springer-Verlag (2015)
17. Fantechi, A., Flammini, F., Gnesi, S.: Formal methods for railway control systems. STTT 16(6), 643–646 (2014)
18. Fulton, N., Mitsch, S., Quesel, J., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) 25th Intl. Conf. on Automated Deduction (CADE), Berlin, Germany. LNCS, vol. 9195, pp. 527–538. Springer (2015)
19. Giachino, E., Laneve, C., Lienhardt, M.: A framework for deadlock detection in `core abs`. Software & Systems Modeling pp. 1–36 (2015)
20. Hähnle, R.: The Abstract Behavioral Specification language: A tutorial introduction. In: Bonsangue, M., de Boer, F., Giachino, E., Hähnle, R. (eds.) International School on Formal Models for Components and Objects: Post Proceedings. LNCS, vol. 7866, pp. 1–37. Springer-Verlag (2013)
21. Henrio, L., Rochas, J.: From modelling to systematic deployment of distributed active objects—extended version. Research Report <hal-01299817>, I3S (Apr 2016)
22. Hentschel, M., Bubel, R., Hähnle, R.: Symbolic execution debugger (SED). In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification, 14th International Conference, RV, Toronto, Canada. LNCS, vol. 8734, pp. 255—262. Springer (2014)
23. James, P., Moller, F., Nga, N.H., Roggenbach, M., Schneider, S.A., Treharne, H.: Techniques for modelling and verifying railway interlockings. STTT 16(6), 685–711 (2014)

24. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F., Bonsangue, M.M. (eds.) Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010). Lecture Notes in Computer Science, vol. 6957, pp. 142–164. Springer-Verlag (2011)
25. Kang, K.C., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-021, Carnegie Mellon University Software Engineering Institute (1990)
26. Platzer, A., Quesel, J.: European Train Control System: A case study in formal verification. In: Breitman, K.K., Cavalcanti, A. (eds.) Formal Methods and Software Engineering, 11th Intl. Conf. on Formal Engineering Methods, ICFEM, Rio de Janeiro, Brazil. LNCS, vol. 5885, pp. 246–265. Springer (2009)
27. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering. Springer (2005)
28. Reichl, K., Fischer, T., Tummeltshammer, P.: Using formal methods for verification and validation in railway. In: Aichernig, B., Furia, C. (eds.) Proc. 10th Intl. Conf. on Tests & Proofs, Graz, Austria. LNCS, Springer-Verlag (2016)
29. Schaefer, I., Bettini, L., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: International Software Product Line Conference. pp. 77–91. SPLC '10, Springer (2010)
30. Schneider, S., Treharne, H.: CSP theorems for communicating B machines. Formal Asp. Comput. 17(4), 390–422 (2005)
31. Serbanescu, V., Azadbakht, K., de Boer, F.S., Nagarajagowda, C., Nobakht, B.: A design pattern for optimizations in data intensive applications using ABS and JAVA 8. Concurrency and Computation: Practice and Experience 28(2), 374–385 (2016)
32. Setyautami, M.R.A., Azurat, A., Hähnle, R., Muschevici, R.: A UML profile for delta-oriented programming to support software product line engineering. In: International Software Product Line Conference. ACM Press (2016)
33. Sun, P., Dutilleul, S.C., Bon, P.: A model pattern of railway interlocking system by Petri nets. In: Intl. Conf. on Models and Technologies for Intelligent Transportation Systems (MT-ITS), Budapest, Hungary. pp. 442–449. IEEE (2015)
34. Thüm, T., Schaefer, I., Hentschel, M., Apel, S.: Family-based deductive verification of software product lines. In: Ostermann, K., Binder, W. (eds.) Generative Programming and Component Engineering, GPCE'12, Dresden, Germany. pp. 11–20. ACM (2012)
35. Wong, P.Y.H., Albert, E., Muschevici, R., Proença, J., Schäfer, J., Schlatte, R.: The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. STTT 14(5), 567–588 (2012)