Modularization of Refinement Steps for Agile Formal Methods

Fabian Benduhn[1], Thomas Thüm[2], Ina Schaefer[2], and Gunter Saake[1]

[1] University of Magdeburg, Germany
[2] TU Braunschweig, Germany

Abstract. The combination of agile methods and formal methods has been recognized as a promising field of research. However, many formal methods rely on a refinement-based development process which poses problems for their integration into agile processes. We consider redundancies within refinement hierarchies as a challenge for the practical application of stepwise refinement and propose superimposition-based modularization of refinement steps as a potential solution. While traditionally, each model in a refinement hierarchy must be developed and maintained separately, our concept allows developers to specify the refinement steps that transform a model into a refined one. We have developed tool support for the language AsmetaL and evaluated our approach by means of a case study. The results indicate a reduction of complexity for the development artifacts in terms of their overall size by 48.6% for the ground model and four refinements. Furthermore, the case study shows that superimposition-based refinement eases the development of alternative refinements for exploratory development and to cope with changing requirements. Thus, we consider this work as a step towards agile formal methods that are tailored to support iterative development, facilitating their incorporation into agile development processes.

Keywords: Formal Methods, Agile Methods, Refinement, Modularity, Superimposition, Abstract State Machines

1 Introduction

Despite the potential benefits of applying formal methods to increase the quality of software and a growing number of success stories, facilitating their industrial adoption has been recognized as an important research challenge [46]. Traditional formal methods and techniques have mostly been developed assuming a waterfall-like development process in which all requirements are known from the beginning and do not change during the development process [36]. For decades, research has focused on developing techniques to prevent errors when transforming a set of well-known requirements into an implementation that faithfully fulfills them [19]. However, researchers and practitioners increasingly recognize the need to adapt and develop formal methods to be incorporated into agile development processes in which requirements are expected to change frequently [13, 24].

In this paper, we contribute to agile formal methods by investigating concepts to ease the integration of refinement-based formal methods into agile processes.

Stepwise refinement is an essential concept in formal methods and has been integrated into many popular methods such as Event-B, ASM, or Z [1, 17, 42]. The idea of stepwise refinement is that the developer starts by specifying a high-level model of the system that is derived from the requirements and easy to understand, but still accurate regarding all relevant system properties [16]. Such a high-level model can already be subject to verification and validation, which helps to prevent errors early in the development process [16]. Once the developer is satisfied with the initial model, it is refined by adding more details or additional functionality. This refinement process continues until a satisfying level of abstraction has been reached, eventually leading to executable code.

While the general idea of model-based refinement (i.e., to postpone design decisions as long as possible during the development process) seems to be compatible with agile processes such as iterative development, its practical application poses several challenges. Researchers have identified the development of reusable modules for model-based refinement as a challenge for their integration into agile processes [23]. In particular, iterative development becomes difficult because of the inherent redundancies between the different representations of the system in the refinement hierarchy. When requirements change, the model of the system needs to be adjusted on several levels of refinement. Every modification potentially needs to be performed on all succeeding levels, each typically maintained as a separate development artifact. This overhead is especially a problem for agile processes, in which changes are expected to occur frequently and must be synchronized between all levels of refinement.

We propose to apply superimposition-based modularization to refinement steps with the goal to avoid redundancies within refinement hierarchies and to ease the replacement and removal of design decisions, making it easier to cope with changing requirements. We exemplify superimposition-based refinement using the Abstract State Machine (ASM) method which includes a very general notion of refinement, subsuming other more restricted refinement concepts used in other formal methods [16]. That is, we do not aim to define a specific mathematical notion of refinement for ASMs - this has to be done by the engineer for each project - but to investigate how to describe the required development artifacts of a given refinement hierarchy. As such, we expect superimposition-based refinement to be applicable for other methods than ASM as well.

We have developed tool support based on FeatureHouse, a tool for compositional development of software based on language-independent superimposition [5]. We extended it to support modular refinement steps using the language AsmetaL [28, 27]. To evaluate our approach, we have performed a case study based on the Landing Gear System [14, 8]. In detail, we make the following contributions:

- We propose to apply **superimposition-based refinement**, allowing developers to specify modular refinement steps that can be automatically com-

posed to derive a model of the system on the desired levels of abstraction, to facilitate flexibility.

– We exemplify the concept with an **extension of AsmetaL** that supports superimposition-based refinement.
– We developed **tool support** for our extension of AsmetaL. It is integrated into Eclipse and allows the direct application of the Asmeta toolset to perform various analyses to the model on each level of refinement.
– We provide **first empirical evidence** of the feasibility of our approach by means of its application to the landing gear case study which indicates a large reduction of system size due to removed redundancies in the models.

2   Modularization of Refinement Steps

We explain the basic concept of model-based refinement in formal methods and discuss some of its challenges for the application to agile development in Section 2.1. In Section 2.2, we propose to modularize refinement steps based on superimposition to reduce redundancies within development artifacts.

2.1   Refinement in Formal Methods

In model-based refinement, the developer starts with an abstract model that is refined stepwise to executable code or a sufficiently detailed model [16]. For the sake of clarity, we explicitly distinguish between refinements and refinement steps; A refinement step describes the changes that are applied to transform the initial model or one of its refinements into a more concrete refinement. The result of a refinement-based development process is a sequence of refinements.

In Figure 1, we show a sequence of refinements for the Landing Gear System that we use as a running example. The Landing Gear System has been proposed by Boniol et al. as a benchmark for formal methods and behavioral verification [14]. It describes an airplane landing gear system consisting of three landing sets. The system controls opening and closing mechanisms of the landing sets and includes features such as sensors, cylinders, and a health monitoring system. For a more complete description of the system we refer to the literature [14].

The refinement sequence of the Landing Gear system, presented in Figure 1, has been adapted from Arcaini et al. who exemplified its stepwise development from an abstract model to Java code [8]. Each refinement step can be applied to the model on a previous level of refinement. The initial model, here simply called *Landing Gear*, only includes behavior of a single landing set and its most basic elements, namely doors and gears, and describes their interaction. The first refinement step *Cylinders* adds the behavior of cylinders that extend and retract during the landing sequence. We depict the resulting refinement of the system model in terms of the involved refinement steps: {Landing Gear, Cylinders} describes the first refinement of the Landing Gear model. Similarly, the next refinement steps add details about the behavior of sensors to the landing set, the two additional landing sets, and a health monitoring system, respectively.
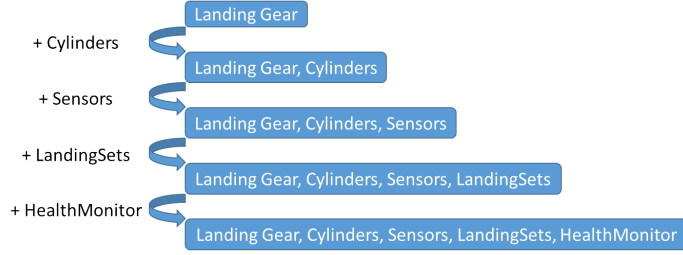
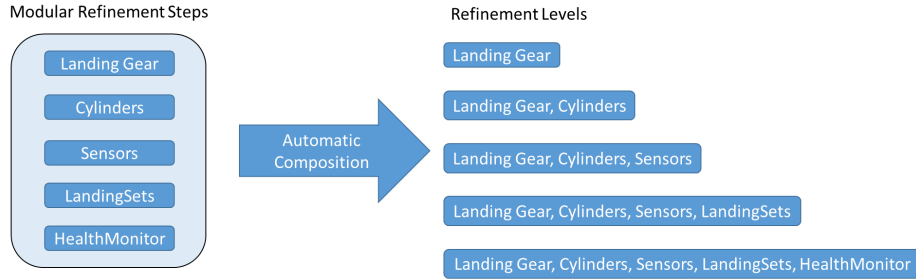Fig. 1: Sequence of Refinements for the Landing Gear System

Fig. 2: Concept of modular refinement applied to the Landing Gear System

A sequence of refinements typically leads to development artifacts containing a high degree of redundancy. The reason is that typically each refinement is merely an extended version of the previous one and only differs in a specific aspect. In practice, refinements are typically created manually by duplicating the initial model and adapting it sequentially by applying refinement steps. This procedure is known as clone-and-own in the context of software variability and has been studied on the level of code [38, 21].

Assuming a sequential development process in which all requirements are known a priori and are not subject to change, the clone-and-own approach would be feasible. However, it is not suitable for agile practices such as iterative development. In particular, the redundancies within the refinement sequence lead to practical problems, especially for the maintenance of development artifacts. In the Landing Gear System, changes to the cylinder sub-system may involve additional adjustments in three succeeding refinement levels. In general, each modification on a given level of refinement may affect lower levels as well. The high degree of redundancy makes it difficult to react flexibly to changing requirements.

2.2   Superimposition-Based Modularization

We have seen that redundancies between development artifacts in traditional refinement hierarchies pose several challenges for the integration of formal methods into iterative processes. To avoid those redundancies, we propose superimposition-based modularization of refinement steps, as illustrated in Figure 2. One of the key ideas is to specify the refinement steps, as partial model representing the

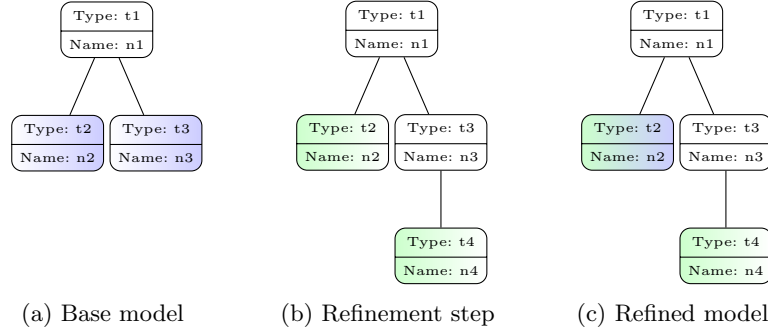(a) Base model   (b) Refinement step   (c) Refined model

Fig. 3: Superimposition-Based Refinement. The colors indicate that node $n2$ is merged using language-specific rules.

delta between the abstract and the refined model. Based on the concept of superimposition, the refinements for each level can be automatically derived from the modular refinement steps (e.g., for analysis purposes). Developers do not need to maintain the models on each level of refinement directly. Instead, only the modular refinement steps have to be maintained manually, allowing the developer to reduce the degree of redundancy in development artifacts. Thus, if we perform a change to a modular refinement step (e.g., Sensors) the change automatically applies to subsequent refinements.

We propose to modularize refinement steps using hierarchical superimposition as proposed by Apel et al. [5]. As depicted in Figure 3, the base model and each refinement step are considered as syntax trees, whose nodes represent syntactical elements of the model. When superimposing two trees, their nodes are merged recursively based on their names, types, and relative positions. Nodes are merged, if they have the same name and type and if their parents have been merged. Nodes that cannot be matched this way are added to the tree at the current position, as is the case for node $n4$ in the example. Corresponding nonterminal nodes are merged recursively, by merging their children. When merging terminal nodes (node $n2$ in the example), specific composition rules are to be defined. We propose such composition rules for AsmetaL in Section 3.2.

By applying superimposition-based composition, the developer only needs to specify the parts of the model that are changed during a refinement step, the model can be generated automatically for each level of refinement. Thus, redundancies within the refinement hierarchy can be avoided to a large extent. As each refinement typically represents a design decision, the modularization facilitates flexibility by allowing developers to replace or modify functionality to reflect changing requirements by merely replacing a module. Thus, it may also become easier to adapt the system or create different variants of the system and respond to changing requirements quickly.

```
1 enum domain HandleStatus={UP|DOWN}
2 enum domain DoorStatus={CLOSED|OPENING|OPEN|CLOSING}
3 enum domain GearStatus={RETRACTED|EXTENDING|EXTENDED|RETRACTING}
4 derived doors: DoorStatus
5 function doors = switch cylindersDoors
6                    case CYLINDER_EXTENDED: OPEN
7                    case CYLINDER_RETRACTED: CLOSED
8                  endswitch
9
10  rule r_closeDoor = switch doors
11                      case OPEN: doors := CLOSING
12                      case CLOSING: doors := CLOSED
13                      case OPENING: doors := CLOSING
14                    endswitch
15
16 function doors = OPEN
```

Fig. 4: Domain, function and rule definition in AsmetaL

3 Modularization of ASM Refinement Steps

We exemplify superimposition-based refinement for the ASM method, and in particular for the language AsmetaL for which we provide composition rules. We introduce ASMs and the language AsmetaL in Section 3.1 and composition rules for AsmetaL in Section 3.2.

3.1 Abstract State Machines and the Language AsmetaL

Abstract State Machines (ASMs) have been proposed by Gurevich as a means to describe algorithms on arbitrary levels of abstraction, and made popular by Börger as the underlying formalism of the ASM method [30, 15]. Besides the ASM formalism, the ASM method, comprises the idea to describe a system on any desired level of abstraction (ground model), and refine it stepwise. For a detailed description of the ASM method we refer to the literature [17].

In this work, we exemplify the proposed concepts using the ASM-language AsmetaL [28] and our running example. The ASM model for the Landing Gear System as used for illustration has been proposed by Arcaini et al. [8]. In AsmetaL, a model include domains, functions, and rules.

Domains represent a mathematical specification for named complex structures. Domains are thereby a combination of either simple predefined types such as integers or other domains. The type of the combination is defined by a set of keywords, such as *enum*, *Set*, or *Map*. Line 1-3 of Figure 4 shows an example of domain definitions from the Landing Gear System.

Functions in ASMs, define the state of the system by their values at a given point of execution. We mainly distinguish between controlled functions, whose

value is controlled by the system, and monitored functions, whose values are given by the environment. An exemplary declaration and definition of functions from the Landing Gear System is shown in Figure 4, Lines 4-5. The type of function *doors* is *DoorStatus* (i.e., a door can be either closing, closed, opening, or open). The function definition uses a case term with the obvious semantics as known from switch statements in programming languages. The doors are open when cylinders are extended and closed when the cylinders are retracted. Thus, the value of function doors is determined by the value of other functions, thus it is considered as a derived function.

Rules in ASMs are sets of updates that, in its basic form, are controlled by conditional statements called guards. In each step, all rules of an ASM are executed simultaneously and define the update set for the next state transition. Figure 4, Line 10 shows the definition of rule r_closeDoor which handles the opening and closing of the doors. In AsmetaL, the main rule marks the entry point of the ASM's execution, from which further rules can be invoked.

3.2   Composition Rules for Refinement Steps in AsmetaL

We propose an extension of AsmetaL that allows to express refinement steps modularly and to derive the desired refinement hierarchies automatically. The composition mechanism is based on superimposition as explained in Section 2.2. Each refinement step contains a syntactically correct, yet partial, ASM. However, only those parts that are subject to change during a refinement step have to be specified in the corresponding module. The developer can introduce new elements in a refinement step or refine an existing element with the same type and name. For the automated composition of terminal nodes, specific composition rules are required, which we will explain in the following.

Refinement Steps in AsmetaL A refinement step may introduce new functions or refine existing ones. When refining a function, the default behavior is to replace the previous definition of the function. Nevertheless, it is possible to include the content of the function from the previous refinement level by using keyword *@original* that we have adopted from method refinement in feature-oriented programming [5]. Figure 5 shows an example of a function refinement and the result of the composition. It is crucial, that the keyword *@original* does not constitute an absolute reference to a particular previous refinement, facilitating a notion of optional refinements providing more flexibility for agile development. The refinement of rules and domains follows the same principle as the refinement of functions as depicted in Figure 6. As our running example does not contain any refinements of domains, we do not show an example.

Granularity of Refinement To prepare AsmetaL for superimposition-based composition, we had to define which language elements should serve as units of composition by representing them as terminal nodes during superimposition. A

```
1 function flow_rate (valveSize, speed) =              Ground Model
2 valveSize * speed
```

```
1 function flow_rate                        Refinement Step (Valves)
2 (valveSize1, valveSize2, speed) =
3 @original(valveSize1, speed) + @original(valveSize2, speed)
```

```
1 function flow_rate                         Composed Refinement
2 (valveSize1, valveSize2, speed) =
3 (valveSize1 * speed) + (valveSize2 * speed)
```

Fig. 5: Refinement of a function in our extension of AsmetaL

```
1              Ground Model    1          Refinement Step (Valves)
2 rule r_openValve =           2 rule r_openValve
3   valve := open             3   if (pipeFill = empty) then
4                             4     @original()
5                             5 endif
```

```
1 rule r_openValve =                         Composed Refinement
2   if (pipeFill == empty) then
3     valve := open
4   endif
```

Fig. 6: Refinement of a rule in our extension of AsmetaL

natural choice for rules are to consider rule definitions as non-terminals. However, our evaluation with the Landing Gear System showed that it might be useful to consider the possibility to refine specific cases of case rules. The reason is that it appeared as a common pattern to add cases or elements to a given case. Thus, we have introduced the keyword *extendable* that can be used to assign a unique identifier to a case rule. This identifier can be used during refinement by referencing it with the keyword *extend_original*. By means of both keywords, it is now possible to explicitly refine cases rules by adding new cases or modify existing ones as illustrated in Figure 7.

4 Tool Support and Evaluation

In order to evaluate our concepts, we have developed tool support and performed a case study based on the Landing Gear System, which already served as a running example in the previous sections. We give an overview about tool support in Section 4.1 and present results of our case study in Section 4.2.

4.1 Tool Support for Superimposition-Based Refinement in Eclipse

The core of our tool support is an extension of FeatureHouse [5], a command-line tool supporting different types of software composition including super-imposition. We integrated support for the language AsmetaL [28], to enable

```
 1 Ground Model                          1 Refinement Step (Valves)
 2                                        2 /*extend_original(pipe)*/
 3 switch(pipeFill)                       3 switch(pipeFill)
 4 /*extendable(pipe)*/                   4   case filled:
 5   case empty:                          5    par
 6     r_closeValve()                     6      @original
 7   case filled:                         7      warnLight = yellow
 8     r_openValve()                      8    endpar
 9 endswitch                              9   case overflowing:
10                                       10     warnLight = red
11                                       11 endswitch
```

```
 1 switch(pipeFill)                            Composed Refinement
 2   case empty:       r_closeValve()
 3   case filled:        par
 4                         r_openValve()
 5                         warnLight = yellow
 6                       endpar
 7   case overflowing: warnLight = red
 8 endswitch
```

Fig. 7: Refinement of a switch statement in our extension of AsmetaL

superimposition-based composition of refinement rules. It is necessary to decide on a granularity for superimposition by choosing which elements should be considered as terminal nodes during superimposition. For each type of terminal node, we implemented composition rules supporting the keyword *@original()* as explained in Section 3.2. Our extension of FeatureHouse can be used to compose a set of AsmetaL models representing different refinement steps.

Our extension of FeatureHouse is integrated into FeatureIDE [45], an Eclipse plug-in integrating numerous tools to develop configurable software. We have extended existing views to handle ASM models, so that they can be used to maintain an overview about the refinement hierarchy. The general development interface can be seen in Figure 8. The Package Explorer, on the left, shows a FeatureIDE project with the Landing Gear System. The folder *refinement_steps*, contains for each refinement step a sub-folder containing a set of AsmetaL models. A configuration describes a sequence of refinement steps and can be created using the Configuration Editor in the top-right window. The set of selectable refinement sequences can be defined in the model.xml file, for which also a graphical editor exists.

The composed models are automatically generated into the *refinement* folder and can be used as input for other tools. Our tool is developed as an Eclipse plug-in, and thus, it easily integrates with the Asmeta toolset, which has been built around the Asmeta Framework and the language AsmetaL [28]. It incorporates several tools including support for simulation, model checking, and static analysis of ASM models. These existing editors, views and analysis tools can be used for the automatically generated ASM models on each level of refinement.
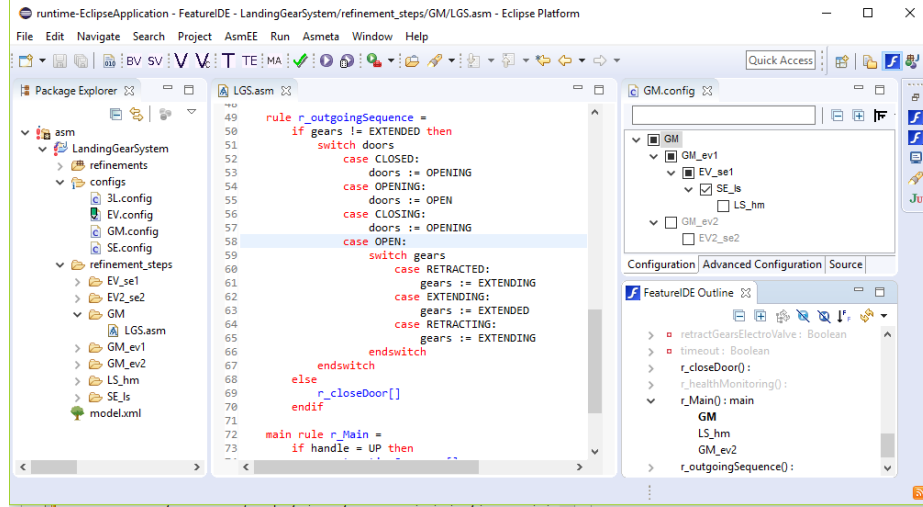
Fig. 8: Integration of our tool support into Eclipse including language-specific editor and views for ASMs in our extension of the language AsmetaL.

4.2   Modularizing Refinement Steps of the Landing Gear System

To evaluate the feasibility of superimposition-based refinement for ASMs, we have used our tool to perform a case study based on the Landing Gear System. Arcaini et al. provide an ASM implementation of the system that has been used as the foundation for our case study [7].

The existing refinement hierarchy by Arcaini et al. describes the AsmetaL model on each level of refinement in detail [8]. We derived the necessary refinement steps, which makes the differences between two subsequent refinement explicit and by modularizing them manually. We took care that the composed models for each level of refinement do not differ semantically from the original models. In addition to syntactical comparisons that were sufficient for large parts of the model, we applied the Asmeta Simulator, Validator, Model Adviser, and Refinement Prover of the Asmeta toolset[3] and compared the results to ensure the correctness of our modularization. After defining the refinement hierarchy and modularizing the refinement steps of the Landing Gear System, we were able to automatically derive the ASM model for each level of refinement.

As our goal is a reduction of redundancies, we compared the size of the model on each level of refinement with the size of the modularized refinement steps. Our results show that it is possible to remove large parts of the redundancy in the development artifacts. In Table 1, we present the size of the AsmetaL models in lines of code, i.e., non-empty lines excluding comments. The second column shows the lines of code of the original refinement step and the third column (the accumulated) size of the necessary refinement steps. In the third row, we present the percentage of the reduction in size.

---

[3] http://asmeta.sourceforge.net/

Refinement	Refinement (acc.) [loc]	Refinement Step (acc.) [loc]	Reduction (acc.) [%]
Ground Model	83 (83)	83 (83)	0.00 (0.00)
Cylinders	170 (253)	154 (237)	9.41 (6.3)
Sensors	187 (440)	131 (368)	29.94 (16.4)
LandingSets	199 (639)	23 (391)	88.44 (38.8)
HealthMonitor	250 (889)	66 (457)	73.60 (48.6)

Table 1: Reduction of system size achieved by modularization of refinement steps in the Landing Gear case study.

The overall size of the refinement sequence has been reduced by decomposition into modular refinement steps by 48.6%. In general, it can be seen that the reduction increases with a growing number of refinement steps. In some cases, such as Cylinders, the reduction is relatively low while other refinement steps benefit from larger savings. These results suggest that a relevant reduction of redundancies is possible, in particular for large refinement hierarchies, but its degree also depends on the particular design of the modularization, such as the choice of granularity for superimposition, and possibly on the nature of the given refinement steps.

Despite being able to derive the original refinement hierarchy from the refinement modules automatically, the modularization of refinement steps allows us to derive even more variants of the system by composing different combinations of refinement steps. We considered the development of two alternative refinements (for Cylinders and Sensors) that have been created during exploratory phases of the original development. With superimposition-based refinement, it was possible to switch between alternative implementations of individual refinement steps. Changes are implicitly propagated to all generated refined models by automatically rebuilding them after each change. This was especially helpful, when considering the impact of changes to subsequent refinements of the original sequence.

We have experienced that it is possible to omit certain refinement steps, allowing the generation of completely new variants of the system. For instance, it would be possible to derive a variant of the Landing Gear System without Sensors but with HealthMonitor. We explored the idea by considering different optional refinements. Our results show that it is generally possible, but may require non-trivial changes to the design of the involved refinement steps or modules to handle particular combinations of refinement steps. For instance, the refinement step *LandingSets* does not depend on a particular refinement, which means that it can be modified arbitrarily (e.g., by choosing a different number of landing sets). In contrast, the refinement step *HealthMonitor* contained syntactical dependencies to the previous refinement step and cannot be freely combined in other ways without major changes.

Furthermore, we observed that each iteration in agile development corresponds to identifying a set of desired refinements in the design space, and developing the necessary refinement steps to generate these refinements. For the

sake of generality, we do not restrict the particular mapping between refinement steps and iterations. One the one hand, it is possible to decide on a set of features for the next iteration, extend the ground model of the previous iteration and adapt all refinements accordingly all the way to the implementation. On the other hand, each iteration could involve the development of a single refinement step only. In this case, the suitability of ASMs to model a system on arbitrary levels of abstraction enables early validation and can be used to get early feedback from the customer. In this case, the refinement sequence might involve an arbitrary combination of refinements.

5   Related Work

Researchers increasingly recognize the need to incorporate formal methods into agile development processes [13, 24]. The use of light-weight formal techniques, such as static verification, in agile development processes has been shown to be applicable in practice [36]. However, researchers have identified the integration of more heavy-weight formal methods typically based on stepwise refinement, such as Event-B, ASM, and Z, as a promising way to develop safety-critical systems [36]. In particular, the need for concepts to facilitate reusability in model-based refinement has been identified as a major challenge [23]. We address this challenge by investigating the application of superimposition as a technique to achieve reuse between refined models. Furthermore, to our knowledge, we are the first to propose the integration of ASM and agile methods.

Formal refinement concepts have been studied intensively [16, 9, 25, 35]. However, the main focus of this line of research are the theoretical underpinnings of refinement rather than on ways to facilitate flexibility. In contrast, we aim to ease development in the presence of refinement hierarchies independent of particular notions of refinement.

There exist other approaches to avoid redundancies in model-based refinement. In particular, the Rodin tool-suite for Event-B allows developers to express a refinement by defining only those parts that differ from the abstract model [2]. However, this merely corresponds to a static reference to previous models which does not facilitate the desired flexibility for agile methods. In contrast, the keyword *@original* in our approach facilitates more flexible extensions by omitting to specify a particular model to which it refers.

Various concepts to modularize features and cross-cutting concerns, such as feature-oriented programming [39, 12], aspect-oriented programming [33] and delta-oriented programming [40], have been proposed [43, 4]. In particular, these approaches build on superimposition (or similar concepts) which has been recognized as a general concept for software composition that can be applied uniformly to all kinds of software artifacts [10, 5].

Historically, superimposition has been proposed as a concept to extend distributed programs with parallel composition [32, 18]. In general, the early work on superimposition focuses on semantic superimposition of particular models, typically with the goal to establish a set of desired properties [26].We adopt

a more general approach from Apel et al., which merely operates on AST-like representations of development artifacts, facilitating a notion of uniform composition for all kind of development artifacts [5]. In this work, we consider this language-independent notion of superimposition.

Superimposition has already been applied to compose method contracts in JML [20, 47, 31, 44], Alloy specifications [6], state machines and markov decision processes [37, 22], and unit tests [34, 3]. We build on this idea by investigating its application to refinement steps, showing that similar benefits such as reduction of redundancies and compositionality can be expected. However, we are the first to apply it to refinement hierarchies. Further, the focus of our work is to leverage the incorporation of refinement-based formal methods into agile processes.

In this work, we combine techniques from formal methods and software composition. Börger and Batory exemplified the modularization of programs, theorems, and correctness proofs from the JBook case study in a uniform compositional way [11]. In our work, we build on their observation that refinements can be modularized in the same way as features in the context of software product lines, but consider the modularization of refinement steps to achieve practical benefits for applying formal refinements. Gondal et al. have proposed a feature-oriented extension of Event-B to investigate to which extent the traditional Event-B composition mechanisms can be used to implement and compose features [29]. However, the focus is on enabling the correct development of several similar variants of a system, but not on the implications of the refinement hierarchies for the development process itself. Schaefer et al. consider modularizing software taxonomies which represent a family of different software variants in a refinement-based fashion [41]. The authors describe a process how a software taxonomy can be transformed into a software product line, but do not target the modularization of the refinement hierarchies themselves as done in this work.

6   Conclusion and Future Work

The introduction of refinement hierarchies to agile development processes poses several challenges. We have identified the inherent redundancies between refinements as particularly problematic for iterative development in which models on multiple levels of refinement may need to be changed frequently to respond to changing requirements. We have proposed superimposition-based modularization as a possible solution and exemplified it using ASMs. We have developed composition-rules and implemented tool support for the language AsmetaL.

To evaluate the concept, we have performed a case study using the well-known Landing Gear System. Our results indicate a significant reduction of redundancies, possibly reducing development and maintenance effort. Furthermore, we show that superimposition-based refinement enables a more flexible refinement hierarchy. While these results are promising, it remains to be seen to which extent developers benefit from this reduction in practice. Further empirical studies regarding comprehensibility and maintainability of the development artifacts would help to better understand the potential advantages.

The modularization of refinement steps may facilitate agile development by allowing developers to modify a design decision by merely replacing the corresponding module. In future work, we want to investigate the potential of modularization of refinement steps to serve as a basis for refinement-based development of software product lines and their efficient verification.

7   Acknowledgments

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York, NY, USA, 1st edn. (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for event-b. In: ICFEM. vol. 4260, pp. 588–605. Springer (2006)
3. Al-Hajjaji, M., Meinicke, J., Krieter, S., Schröter, R., Thüm, T., Leich, T., Saake, G.: Tool Demo: Testing Configurable Systems with FeatureIDE. In: Proc. Int'l Conf. Generative Programming: Concepts & Experiences (GPCE). pp. 173–177. ACM, New York, NY, USA (2016)
4. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer, Berlin, Heidelberg (2013)
5. Apel, S., Kästner, C., Lengauer, C.: Language-Independent and Automated Software Composition: The FeatureHouse Experience. IEEE Trans. Software Engineering (TSE) 39(1), 63–79 (Jan 2013)
6. Apel, S., von Rhein, A., Thüm, T., Kästner, C.: Feature-Interaction Detection Based on Feature-Based Specifications. Computer Networks 57(12), 2399–2409 (Aug 2013)
7. Arcaini, P., Gargantini, A., Riccobene, E.: Modeling and analyzing using asms: The landing gear system case study. In: ABZ 2014: The Landing Gear Case Study, pp. 36–51. Springer (2014)
8. Arcaini, P., Gargantini, A., Riccobene, E.: Rigorous development process of a safety-critical system: from asm models to java code. International Journal on Software Tools for Technology Transfer 19(2), 247–269 (Apr 2017)
9. Banach, R.: Model based refinement and the tools of tomorrow. In: International Conference on Abstract State Machines, B and Z. pp. 42–56. Springer (2008)
10. Batory, D.: A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In: Proc. Generative and Transformational Techniques in Software Engineering. pp. 3–35. Springer, Berlin, Heidelberg (2006)
11. Batory, D., Börger, E.: Modularizing Theorems for Software Product Lines: The Jbook Case Study. J. Universal Computer Science (J.UCS) 14(12), 2059–2082 (2008)

12. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Software Engineering (TSE) 30(6), 355–371 (2004)
13. Black, S., Boca, P.P., Bowen, J.P., Gorman, J., Hinchey, M.: Formal versus agile: Survival of the fittest. Computer 42(9) (2009)
14. Boniol, F., Wiels, V.: The landing gear system case study. In: ABZ 2014: The Landing Gear Case Study, pp. 1–18. Springer (2014)
15. Börger, E.: High level system design and analysis using abstract state machines. In: Applied Formal Methods—FM-Trends 98, pp. 1–43. Springer (1999)
16. Börger, E.: The asm refinement method. Formal Aspects of Computing 15(2), 237–257 (2003)
17. Börger, E., Stark, R.F.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Secaucus, NJ, USA (2003)
18. Bougé, L., Francez, N.: A compositional approach to superimposition. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 240–249. ACM (1988)
19. Clarke, E.M., Wing, J.M.: Formal methods: State of the art and future directions. ACM Computing Surveys (CSUR) 28(4), 626–643 (1996)
20. Clifton, C., Leavens, G.T.: Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In: Proc. Workshop Foundations of Aspect-Oriented Languages (FOAL). pp. 33–44. Iowa State University, Ames, IA, USA (Apr 2002)
21. Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., Czarnecki, K.: An Exploratory Study of Cloning in Industrial Software Product Lines. In: Proc. Europ. Conf. Software Maintenance and Reengineering (CSMR). pp. 25–34. IEEE, Washington, DC, USA (2013)
22. Dubslaff, C., Klüppelholz, S., Baier, C.: Probabilistic Model Checking for Energy Analysis in Software Product Lines. In: Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD). pp. 169–180. ACM, New York, NY, USA (2014)
23. Edmunds, A., Olszewska, M., Waldén, M.: Using the event-b formal method for disciplined agile delivery of safety-critical systems (2015)
24. Eleftherakis, G., Cowling, A.J.: An agile formal development methodology. In: Proceedings of the 1st South-East European Workshop on Formal Methods. pp. 36–47 (2003)
25. Ernst, G., Pfähler, J., Schellhorn, G., Reif, W.: Modular refinement for submachines of asms. In: International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z. pp. 188–203. Springer (2014)
26. Fiadeiro, J., Maibaum, T.: Categorical semantics of parallel program design. Science of Computer Programming 28(2-3), 111–138 (1997)
27. Gargantini, A., Riccobene, E., Scandurra, P.: Deriving a textual notation from a metamodel: an experience on bridging modelware and grammarware. Milestones, Models and Mappings for Model-Driven Architecture p. 33 (2006)
28. Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based language and a simulation engine for abstract state machines. J. UCS 14(12), 1949–1983 (2008)
29. Gondal, A., Poppleton, M., Butler, M.: Composing Event-B Specifications: Case-Study Experience. In: Proc. Int'l Symposium Software Composition (SC). pp. 100–115. Springer, Berlin, Heidelberg (2011)
30. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM Transactions on Computational Logic (TOCL) 1(1), 77–111 (2000)
31. Hähnle, R., Schaefer, I.: A Liskov Principle for Delta-Oriented Programming. In: Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA). pp. 32–46. Springer, Berlin, Heidelberg (2012)

32. Katz, S.: A superimposition control construct for distributed systems. ACM Transactions on Programming Languages and Systems (TOPLAS) 15(2), 337–356 (1993)
33. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Proc. Europ. Conf. Object-Oriented Programming (ECOOP). pp. 220–242. Springer, Berlin, Heidelberg (1997)
34. Kim, C.H.P., Marinov, D., Khurshid, S., Batory, D., Souto, S., Barros, P., D'Amorim, M.: SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In: Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE). pp. 257–267. ACM, New York, NY, USA (Aug 2013)
35. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming. Springer (2012)
36. Larsen, P.G., Fitzgerald, J.S., Wolff, S.: Are formal methods ready for agility? a reality check. In: FM+ AM. pp. 13–25. Citeseer (2010)
37. Li, H., Krishnamurthi, S., Fisler, K.: Modular Verification of Open Features Using Three-Valued Model Checking. Automated Software Engineering 12(3), 349–382 (Jul 2005)
38. Linsbauer, L., Lopez-Herrejon, R.E., Egyed, A.: Variability Extraction and Modeling for Product Variants. Software and System Modeling (2016), to appear
39. Prehofer, C.: Feature-Oriented Programming: A Fresh Look at Objects. In: Proc. Europ. Conf. Object-Oriented Programming (ECOOP). pp. 419–443. Springer, Berlin, Heidelberg (1997)
40. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-Oriented Programming of Software Product Lines. In: Proc. Int'l Software Product Line Conf. (SPLC). pp. 77–91. Springer, Berlin, Heidelberg (2010)
41. Schaefer, I., Seidl, C., Cleophas, L.G., Watson, B.W.: Splicing TABASCO: custom-tailored software product line variants from taxonomy-based toolkits. In: SAICSIT'15. pp. 34:1–34:10 (2015)
42. Spivey, J.M.: Understanding Z: A Specification Language and Its Formal Semantics. Cambridge University Press, New York, NY, USA (1988)
43. Tarr, P., Ossher, H., Harrison, W., Sutton, Jr., S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proc. Int'l Conf. Software Engineering (ICSE). pp. 107–119. ACM, New York, NY, USA (1999)
44. Thüm, T.: Product-Line Specification and Verification with Feature-Oriented Contracts. Ph.D. thesis, University of Magdeburg, Germany (Feb 2015)
45. Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. Science of Computer Programming (SCP) 79(0), 70–85 (Jan 2014)
46. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: Practice and experience. ACM computing surveys (CSUR) 41(4), 19 (2009)
47. Zhao, J., Rinard, M.C.: Pipa: A Behavioral Interface Specification Language for AspectJ. In: Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE). pp. 150–165. Springer, Berlin, Heidelberg (2003)