Softw Syst Model
DOI 10.1007/s10270-014-0442-0

REGULAR PAPER

Expressing aspectual interactions in design: evaluating three AOM approaches in the slot machine domain

Johan Fabry · Arturo Zambrano · Silvia Gordillo

Received: 27 January 2014 / Revised: 18 November 2014 / Accepted: 26 November 2014

Abstract In the context of an industrial project, we evaluated the implementation of the software of a casino slot machine. This software has a significant amount of crosscutting concerns that depend on and interact with each other as well as with the modular concerns. We therefore wished to express our design using an appropriate aspect-oriented modeling approach. We therefore evaluated three candidate methodologies: Theme/UML, WEAVR, and RAM to establish their suitability. Remarkably, only the last of the three has shown to allow an adequate expression of the interactions, albeit not fully explicit. The first two fall short because half of the interaction types cannot be expressed at all while the other half need to be expressed using a work-around that hides the intention of the design. Neither does RAM allow a fully explicit expression of interactions, but it would be the most adequate approach for the slot machine case.

Communicated by Dr. Ana Moreira.

This article extends the article “Expressing Aspectual Interactions in Design: Experiences in the Slot Machine Domain” [14] that appeared in the Proceedings of the ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (MODELS 2011) Johan Fabry is partially funded by FONDECYT project 1130253.

J. Fabry (✉)
PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Santiago, Chile
e-mail: jfabry@dcc.uchile.cl

A. Zambrano · S. Gordillo
LIFIA, Facultad de Informática, Universidad Nacional de La Plata, La Plata, Argentina

A. Zambrano
Departamento de Ciencia y Tecnología, Universidad Nacional de Quilmes, Bernal, Argentina

Keywords Aspect-oriented modeling · Aspect interactions · Case study

1 Introduction

A slot machine (SM) is a casino gambling device that has five *reels* which spin when a *play* button is pressed. An SM includes some means for entering money, which is mapped to *credits*. The player bets an amount of credits on each play, the SM randomly selects the displayed symbol for each reel, and pays the corresponding prize, if any. Credits can be extracted (called a *cash-out*) by different mechanisms such as coins, tickets, or electronic transfers.

In the context of an industrial project, we were required to re-implement the software for a particular SM. Previous experience had taught us that, beyond the main functionality sketched above, there are a significant amount of crosscutting concerns present in such applications. For example, counters need to be maintained to be able to audit the SM and the SM needs to be accessible over the network. Moreover, these concerns depend on and interact with each other as well as with the modularized concerns. We therefore opted to use Aspect-Oriented Software Development in this implementation, taking special care of dependencies and interactions between the different aspects and modules. In a previous step, we analyzed the different concerns that define the behavior of SMs, with a specific focus on concern interactions at the requirements level. We described a number of problems with the evaluated methodologies and proposed extensions to the methodologies that we subsequently validated [36,37].

The second step in our development process is modeling the software using an adequate approach for aspect-oriented modeling (AOM). However, to the best of our knowledge there has been no work published that evaluates

AOM approaches in an industrial setting, with a focus on interactions between the different concerns. We therefore undertook an evaluation of three mature AOM approaches to establish their applicability in our context, and we report our evaluation in this article.

As in our previous work, the focus of the evaluation remains on the ability of each approach to deal with aspect interactions, allowing them to be explicitly included in the design. We require such explicit inclusion in the design to be able to better perform development, maintenance, and evolution, especially given that there are about 600 feature requirements in the system, which change frequently.

Somewhat surprisingly, regarding our focus on the ability of AOM approaches to explicitly deal with aspect interactions none of the approaches fully meet our criteria. We can consider only one of the three to be reasonably satisfactory. Note that, as the focus of our study does not include other desirable properties for design models, we consider discussions of other properties outside of the scope of this text.

As basis for our selection, we used surveys on AOM [8, 35], complemented by a study of more recent literature. The chosen approaches are Theme/UML [10], WEAVR [11, 13], and RAM [22]. Of the mature approaches that are accepted in the community, these are the approaches that claim to have some support for interactions, the key feature that we wanted to evaluate. Furthermore, all methodologies have specific advantages. Theme/UML integrates with Theme/Doc: an aspect-oriented requirements methodology for requirements specification [37]. WEAVR is arguably the best-known industrial application of AOM, and the only methodology that we are aware of that is used in industry to develop complex applications. RAM is a multi-view approach providing a radically different approach to AOM, promising scalability and specific forms of resolution of interactions.

We now give an overview of the requirements we have for the design document, before giving a high-level overview of the design and the different interactions that need to be specified. Section 4 then proceeds with an evaluation of Theme/UML, and Sect. 5 follows up with an evaluation of WEAVR. In Sect. 6, we discuss the design of the slot machine in RAM. A discussion comparing the merits of the different approaches and broadening the scope to other design methodologies is given in Sect. 7. We present related work in Sect. 8 and conclusions and future work in Sect. 9.

2 Requirements for the design

In the design phase, our goal is to refine the requirement specification documents into a model of the software artifacts that will form the final system. This model, written down in a design document, will be passed to the developers for implementation. Hence, it should be sufficiently complete to allow for the implementation to be produced relatively independently. As we are performing Aspect-Oriented Software Development, the choice of an AOM approach for creating this document is given. The expectation is to be able to produce the complete design documents, i.e., not having to resort to a significant additional documents with an ad hoc notation to complement for omissions in the methodology. In the latter case, the advantages of using a standard AOM are small and we would consider rolling our own AOM. Note that, as previously mentioned, we found that none of the methodologies has shown itself to be completely sufficient, and hence in Sect. 7, we discuss possible ways to complement the design documents.

In addition to the goal of using an existing methodology, we have three, related, expectations of the design document: maintenance support, explicit interactions, and scalability.

In subsequent maintenance or evolution phases, the changes made in the requirements will trigger subsequent changes in the design, and the developers will modify the implementation accordingly. Such later modifications must not break the system because they violate constraints of the original design or go against the original design decisions. If the change is significant enough to warrant modifying the design constraints or assumptions, the original intentions should be maintained as much as possible. Hence, the design document must be clear on which are the critical design decisions that were made and what assumptions were taken. Furthermore, it is known that the presence of aspects in a software system that is being evolved can be problematic [21]. Such issues should be mitigated by the information that is explicitly available in the design document. When evolving the software, the implementers must be able to use the document as a guide, seeing what assumptions taken by the aspects no longer hold, or what new code now also falls within the realm of an aspect.

As we have stated above, our experience is that there is a significant amount of non-trivial interactions between the different aspects of the system. This is also confirmed by the results of the requirements analysis we have performed previously [36,37]. Even though aspects are intended to provide advanced modularity and decoupling, they do not exist in isolation. As any module in software, their presence impacts other modules and their functionality may depend on other modules. Documented design decisions should therefore include not only which modules will be aspects and where they crosscut, but also how they *interact* with each other. This information must be made explicit so that critical information is correctly passed to the implementation phase and is present when maintaining or evolving the software.

2.1 Scalability is key

The SM application requirements documents establish approximately 600 requirements [37]. This results in a crucial need for scalability of the design phase. We consider it unrealistic to produce a design document that goes into great detail for all of these requirements, as such a heavyweight approach will not scale.

A second motivation for the need for scalability is that the different requirements specifications [15,16,29] regularly change and moreover are under control of different legal institutions. As a result, frequent changes to deal with new (legal) issues are not synchronized between the different documents. A heavyweight design document that needs to be updated on each change of a regulation document as well as consequent changes in other documents will cause an unacceptable overhead in maintenance and evolution.

As a partial solution to handle these scalability issues, we expect the AOM approach to provide for some means of abstraction over similar patterns in the design. For example, there are different (informal) types of errors that can occur in the SM, and each type requires a different action to be undertaken. The two extreme cases of errors are the following: Minor errors, such as the ticket printer running out of paper requires a message to be sent to the casino server without interrupting play. Major errors, such as a player tilting the machine (to attempt to influence the outcome of a play) require the machine to lock up immediately and to call an attendant by lighting the lamp at the top of the machine while sounding an alarm. There should be a way such that for a class of error only one model is created, instead of a model for each specific error condition.

3 Design overview

This work is one phase of a complete development effort that considers the re-implementation of the SM software. Due to our experience with the existing software, we know that there are significant issues with crosscutting concerns and their interactions. Based on our knowledge of aspect-oriented software development, it seemed that this approach to modularize the different concerns would be the most effective. We therefore chose to use AOSD and not other forms of advanced modularity for this development effort. The use of other forms of advanced modularity may also be adequate; however, a comparison of other forms and subsequent analysis is outside of the scope of this text.

In the first step of the development effort, we performed a requirement analysis that has been reported in previous work [36]. This analysis was based on the formal and legal requirements for the SM as well as our experience of the previous implementation. It established the separation of the software in different concerns and how they crosscut and interact, at a requirements level. For more details on the requirements analysis, we refer to our report [36].

Considering the results of the requirements analysis phase we performed, we now give an outline of how we envision the design of the SM software. Based on our knowledge of the SM domain and the results of requirements analysis, we consider this design outline to be the most adequate for the software that is envisioned. Other designs are of course feasible but a discussion of trade-offs taken is outside of the scope of this text, given that here we focus on the expression of aspectual interactions. The purpose of the proposed design in this section is to provide us with a concrete basis for evaluation of the AOM methodology in this regard. More specifically, the methodology must allow us to expand and refine the overview into a complete design document that treats interactions in an acceptable manner.

3.1 Aspects in the design

A class diagram that shows the outline of the design is given in Fig. 1. It uses an ad hoc extension of UML to indicate crosscutting, showing that we model the following crosscutting concerns as aspects (using the “Aspect” stereotype): Metering, Demo, Program Resumption, Error Conditions, S Communications Protocol, G2S Protocol. We give an overview of these aspects next.

Metering The Metering aspect crosscuts Game and other base entities in order to keep meters data up to date. Meters are essentially a set of counters that keep information about past plays, e.g., the total amount bet. This information is used, among other things, to create reports.

Demo For legal certification the SM must have a ‘Demo’ mode, where all possible outcomes for a play can be simulated. The Demo concern therefore needs to control the outcome produced by the Game class. It furthermore crosscuts Metering to avoid polluting accounting meters when it is active.

Program Resumption is a persistence and recovery requirement. The system should recover the last state after a power outage. Information to be saved includes the status of the current play and the values of the meters.

Error Conditions detected by the game, such as tilt, out of paper, among others, are detected by the Error Condition detection aspect. Once an error condition is detected, some actions need to be performed, e.g., in case of a tilt illuminating the tower lamp and sounding an alarm to call the casino attendant.

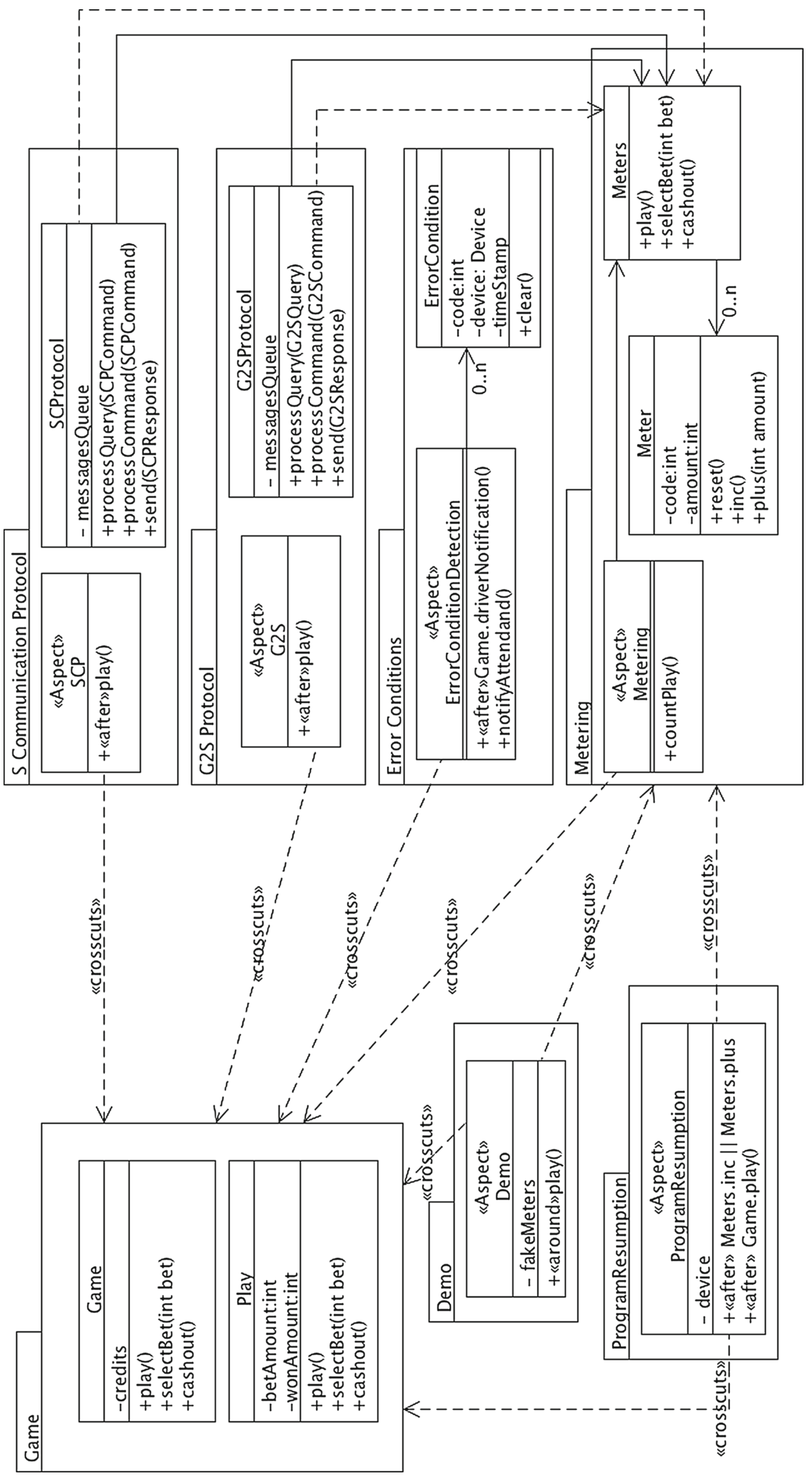

Fig. 1 Overview of the class structure of the design

Communication Protocols The S Communications Protocol[1] (SCP) and G2S Protocol are communication protocols frequently used in the gaming industry. Their corresponding aspects crosscut the Game modules to add behavior such as multiple SMs competing for the same jackpot. Moreover, both protocols need to report metering information and hence crosscut the Meters aspect.

Figure 1 shows that a simple extension of UML already suffices to provide the outlines of the aspectual design. Not surprisingly most, if not all, of the AOM approaches we studied allow us to produce a model similar to this diagram. What is, however, lacking in the above diagram is the information of how the various aspects interact with each other, as well as with the base application. For example, when in Demo mode network communication must be disabled, as queries from the server may only receive values corresponding to normal play conditions. This information should also be present in the design document, but we find no immediately obvious way in which this can be diagrammed, and hence the lack of this information in Fig. 1.

3.2 Interactions between concerns

Our resulting design document not only needs to contain the information of the aspects present in the system, but also how they crosscut. It is also necessary that the interactions which were identified in the requirements analysis phase be present in the design document. To better understand what our needs are for this part of the design document, we now give an overview of the different interactions in the SM, and how we want this to be reflected in the document.

We structure this discussion and the evaluations of the AOM approaches later in the text using the AOSD-Europe technical report on interactions [33]. It classifies interactions in four different types: *dependency*, *conflict*, *mutex*, and *reinforcement*, and the SM software contains an instance of each of these types.

Dependency: Communication Protocols on the Meters Both communication protocols access the meters in order to report their values to the server. Consequently, if for some reason metering is not present, the communication protocols cannot operate. We need to document this dependency to ensure the consistent behavior of the system.

Conflict: Demo versus Multiple Concerns The aspects of Meters, Communication Protocols, and Program Resumption are present to comply with legal accounting requirements regarding plays performed on a SM. The Demo aspect, also a legal requirement, conflicts with all of the above aspects. This is as the legislation states that a play in Demo mode must not alter the meters nor that its activity is visible over the network. Hence, after a Demo session the Game must recover its original status and any event or state change while in Demo must not be reported by the communication protocols.

In order to cope with this conflicting behavior, the design and implementation must take care of the following when in demo mode:

- The communication protocols should not divulge any values of the machine when queried by the server and should not report errors over the network. Protocols must report the SM as being in an *out of service* state. This might be done by intercepting the response behavior of the protocols and responding with an "out of service" message. Stopping communications completely is not an option due to protocol constraints.
- Meters and GameRecall information must not be polluted with Demo information. A fake set of meters and log should be used. This ensures that actions in demo mode do not alter the meter values of normal operations and, at the same time, allows to audit the correct behavior of the SM. When finishing the demo session, the "original objects" must be restored.
- Program Resumption must not persist the information generated during "demo" plays. The functionality of this aspect must be deactivated. That is, if the machine is restarted in normal mode, changes due to demo mode must be lost.

Mutex: Between both Communication Protocols Both communication protocols provide similar functionality, allowing the server to query information and set some configuration values and state on the SM. For read-only behavior, such as reporting the value of a meter, there is no problem with having them active at the same time as no interference will result. On the other hand, for operations that alter the state of the machine, mutual exclusion may need to be ensured during a single program execution. If not, inconsistencies in the SM may arise. For example, consider setting the time of the SM, an operation performed by the casino server. With both communication protocols enabled, two different servers with different clock values may set the time on the SM to either of both clock values. As a result, the timing of events on the SM is ambiguous. To document, this mutex what we need is the ability to express that certain object interactions may not occur during the programs' execution.

More concretely, we consider the following design decisions that must be documented:

- Configuration commands will be instances of different classes, which will belong to the same hierarchy as needed. For example, SetTime, setting the time, versus

[1] A pseudonym, licensing restrictions prohibit us from using the real name.

SetProgressive, setting the increment value for a progressive jackpot.
- As protocols use almost the same set of commands, these will be shared between the protocols; that is, there will be no separate hierarchies for G2S and SCP commands. This means that a command such as SetTime can be issued either by the G2S or SCP protocol.
- For a given run of the system, it is necessary to assign which command can be received from which protocol. This can be done by configuration, or using an first-come first-served policy. An example of the latter would be that once a command,e.g., SetTime, is received through the G2S protocol the next occurrences of SetTime will only be processed if they come from the G2S monitoring system.
- Complementary, if a configuration command arrives through an improper protocol (SetTime coming from SCP in our previous example), it will be ignored, and this occurrence will be logged for future fixing.

Reinforcement: From Error Conditions to Communication Protocols There is a reinforcement from Error Conditions to Communication Protocols. Not all the Error Conditions specified in the legislation are mandatory; however, when an optional error condition is present in the game, e.g., because a driver allows for these errors to be detected, the communication protocols must report this to the server. This means that during the development of new versions of the Game, when new error conditions are present, the associated behavior in the Communication Protocols should be revisited to ensure that the new information is properly reported. Hence, we need to document that a change in different parts of the application enables optional or extended behavior of a given concern.

In order to cope with *reinforcement*, we need to express the following design decisions:

- Joinpoints where the error conditions are issued must be captured.
- When detected, it is necessary to perform some kind of check that ensure the error condition is notified or it is intentionally left aside for each communication protocol.

3.3 Methodology of evaluation

We investigated the literature to select the approaches for evaluation. The first requirement that we had for the approaches is that they should be mature, as witnessed by an apparently comprehensive feature set or reports of their use for realistic cases. The second requirement was that they be accepted by the community, as reflected in the presence of multiple publications and a significant number of references to these publications. The third and last requirement was for the approaches to claim to have some form of interaction support. Given that interaction support is the specific focus of our evaluation, we did not consider other features of the different approaches for selection.

We found three approaches that satisfied these three criteria: Theme/UML, WEAVR, and RAM. We evaluated them in this order and in this text report on the evaluations in the same order.

The evaluation proceeded as follows: For each approach, we read published work as well as any web sites on the approach. If the tools of the approach were claimed to be freely available, we tried to obtain, learn, and use these tools. We then endeavored to provide a design document for at least one instance of each interaction kind that is listed in Sect. 3.2. Notably, we did not construct a full design for the entire slot machine, as this would be prohibitively expensive considering the time it would take.

When in doubt of how to use an approach to construct a model, we contacted the primary authors of the respective publications via e-mail, asking for more information. We continued via e-mail or by other means deemed more effective. Conversations continued until we were satisfied with the provided information, while also taking into account the latency and clarity of responses. In general, interaction with the respective authors was not so fluid. There was, however, one case: RAM, where the authors were highly responsive, even permitting direct interaction to clear up questions and doubts by means of a research visit to their laboratory.

4 Evaluation of Theme/UML

Theme/UML is the second half of the Theme approach for aspect-oriented requirements analysis and design. The first half is called Theme/Doc and is a methodology for AO requirements analysis. Theme provides a process for transforming requirements in Theme/Doc into a design in Theme/UML and, moreover, claims to have support for conflict resolution. We therefore chose to evaluate Theme for our development effort. In the requirements engineering phase [37], we have evaluated Theme/Doc and now continue with an evaluation of Theme/UML.

The Theme/UML approach [10] is an extension of UML that provides both a notation and a methodology for modeling AO systems. In Theme/UML, a *theme* refers to a concern. A theme can consist of class diagrams, sequence diagrams, and state diagrams, each of which is extended with the required notation to be able to express aspect-oriented concepts. Each theme is designed separately and, subsequently, the themes are composed each other. This is performed using composition relationships that detail how this is performed.

Theme/UML claims to have tool support in the form of an Eclipse plugin that works on the output of the MagicDraw tool. Unfortunately, MagicDraw is proprietary software, and the Theme/UML Eclipse plugin was outdated at the time we performed our comparison (which was confirmed by the plugin authors). This prevented us from using any tool support. However, the existent scientific publications together with the user manuals of the plugin and the theme approach web page gave us a comprehensive basis on which to perform our evaluation. Note that the support offered by the plugin is to provide help during the composition of themes, which is downstream from our evaluation in the modeling process. This is our objective to clearly document aspect interactions during the design.

Themes are divided into two classes: base and crosscutting themes. Base themes describe a concern of the system that has no crosscutting behavior. Base themes are composed, both structurally and behaviorally, to form the base model. If a given concept appears in multiple themes, the composition can merge the various occurrences into one entity. Crosscutting themes describe behavior that should be triggered as the result of the execution of some behavior in the base model. They are designed similar to base themes and are parameterizable. Parameters provide a point for the attachment of the crosscutting behavior to the base model. By binding them to values of the base themes, the crosscutting themes are composed with the base model. Crosscutting themes are composed one by one with the base themes until the complete design is produced.

In accordance with Fig. 1, we modeled Game as a base theme and Demo, G2S, Meters, and SCP as crosscutting themes. We found it is straightforward to express where to attach the crosscutting behavior, both on the base themes and on other crosscutting themes. However, when considering interactions we find that Theme/UML does not perform as well. We now discuss the obstacles we encountered classified in the four different kinds of interactions [33]: Dependency, Conflict, Mutex, and Reinforcement.

4.1 Dependency

The metering theme maintains track of given events in the game by changing the values of meter objects, as shown in Fig. 2a. Complementary to this, Fig. 2b shows how one of the communication protocols responds to queries sent by the remote server, using the information previously stored in the meters. It is clear that the latter behavior implies the former, i.e., the communication theme depends on the meters theme.

The Theme/UML methodology, however, states that each theme defines all structure and behavior needed to provide the desired functionality, i.e., in a stand-alone fashion. Furthermore, the designer may choose a subset of all themes to compose a system [10]. In our case, this will lead to errors, as selecting the theme of a communication protocol without adding the theme of meters leads to an inconsistent design of the system.

What we need is a way to express that the meters themes are necessary whenever the communication protocol themes are composed into the system, but we have found no way to specify this in Theme/UML. Hence, we are unable to include the dependency in the design.

4.2 Conflict

Theme/UML provides support for conflict resolution when composing different themes. These composition conflicts arise when the same diagram element in different themes has an attribute with different values. An example of this is an instance variable with different visibility specifications. Conflict resolution then consists of choosing which of the conflicting attributes to use in the composition.

The conflicts we are facing are, however, of a different nature. For example, consider the Demo aspect. As mentioned in Sect. 3.2, when it is active all conflicting aspects must be somehow deactivated. We therefore need to model the predominant nature of this aspect in some way.

Depending on the aspect language used in the implementation, such a conflict management strategy can be realized in different ways:

1. The Demo aspect could intercept and skip the behavior of the conflicting aspects (using around advice without a proceed).
2. When the Demo aspect is deployed, conflicting aspects must be undeployed; that is, if runtime deployment and undeployment of aspects are possible.
3. If load time weaving is available, different configurations of active aspects could be loaded when the SM boots. One of these would have Demo installed, and the conflicting aspects not, a second configuration would be the inverse.

We were obliged to model the conflict management strategy using the first option. This is because, to the best of our knowledge, there is no way to fully express the other options using Theme/UML.

We therefore model conflict management as follows: the Demo theme crosscuts the Game theme, capturing the execution of play() for the Game class. When active, Demo skips the execution of the original play() and instead generates a predetermined outcome (which is the main responsibility of the Demo mode). In order to keep the meters unharmed, parts of the Metering theme behavior are captured and skipped. Considering the communication protocols, their original behavior is altered: Instead of responding to queries, failure responses are returned.

Our model is shown in Fig. 3. We use Theme/UML sequence diagrams, a straightforward extension of UML

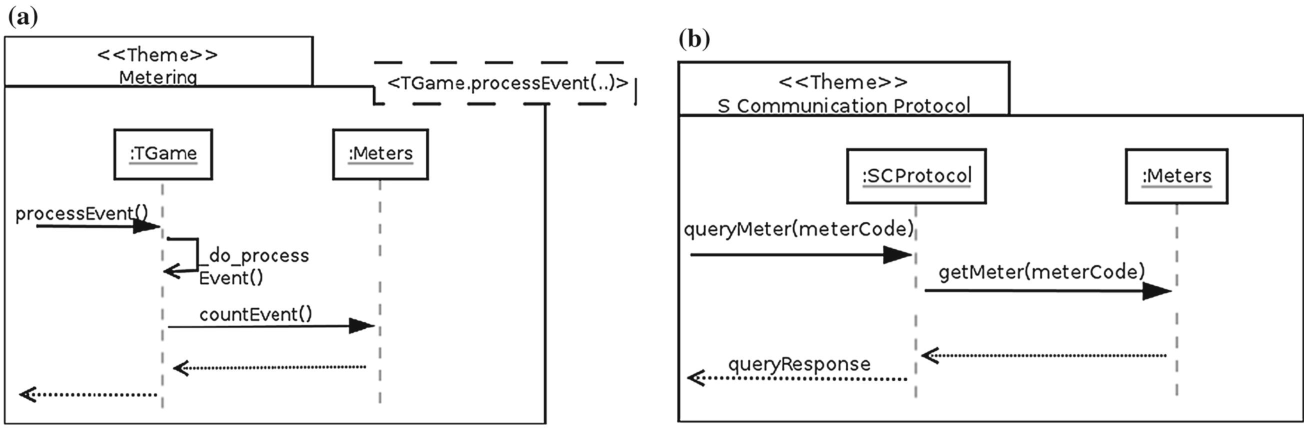

Fig. 2 Information captured by meters is used by the S communication protocol. **a** Metering theme acquiring information regarding events on Game, **b** SCP theme using information acquired by metering behavior

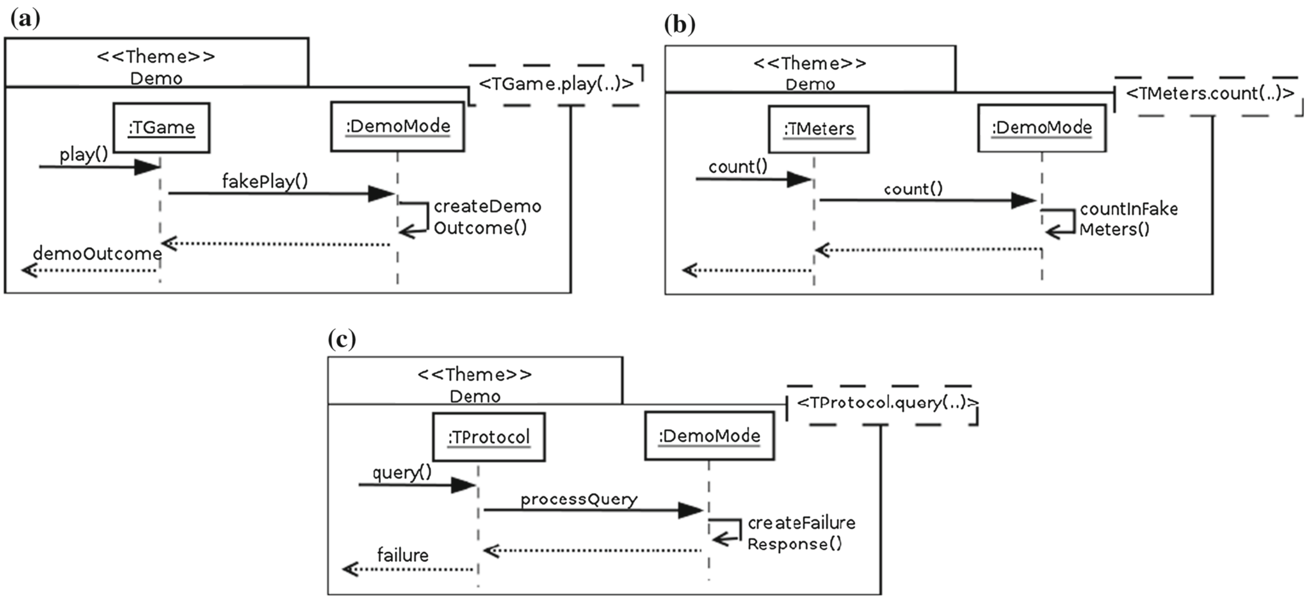

Fig. 3 Demo theme affecting the behavior defined in Game, Metering, and Protocols. **a** Demo on Game, **b** Demo on Meters, **c** Demo on Protocols

sequence diagrams. The figure shows three Themes, each of which has a template parameter in the top right corner, corresponding to the message send that starts the sequence. At composition time, this parameter is bound to a specific message send in the base theme, i.e., the join point in the base code is identified. Also, within a sequence diagram, the behavior of the join point which is matched can be invoked, put differently, Theme has an equivalent of the AspectJ proceed construct. The syntax to express this call is _do_*templateOperation*. Note that the absence of such a call implies that the original behavior never occurs. For instance, in Fig. 3 there are no _do_play, _do_count, or _do_query calls, which means the join point behavior is skipped.

The above proposed solution has two downsides. Firstly and most importantly, the design does not explicitly reveal the intention: the conflict between Demo and Meters, and Demo and the communication protocols. Instead it must be deduced from the implementation proposed in the diagrams. Secondly, we cannot model the conflict resolution strategy differently. Of the three design choices we proposed above, only the first could be modeled in Theme.

4.3 Mutex

Part of the behavior of the communication protocols is configuration command processing, as these game parameters can be set by the servers. Both protocols implement this feature, but it is not permitted that multiple protocols set the same value during a run of the program. The interaction we thus want to model is mutual exclusion between configuration actions: Two protocols cannot configure the same item during a given program execution.

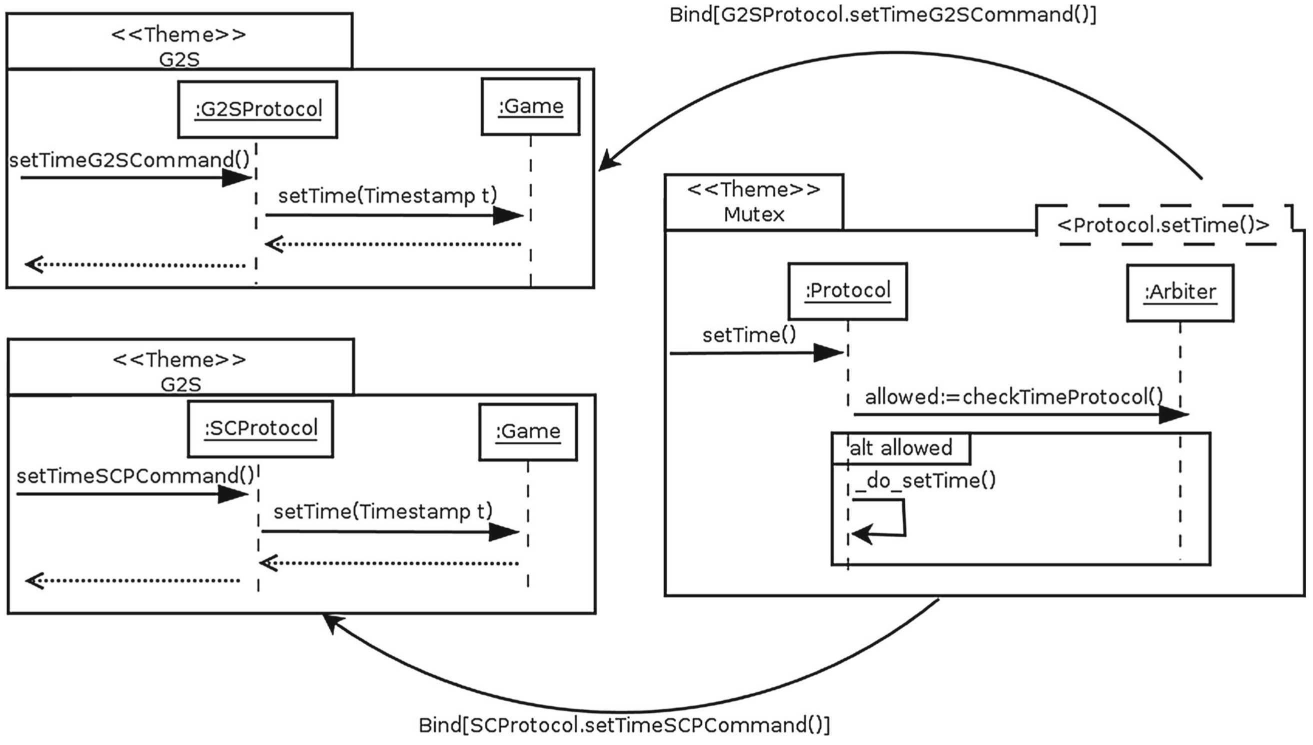

Fig. 4 Two themes configuring the same item in the Game and Mutex interaction arbiter

Concretely, each protocol is modeled as a theme, where each theme defines the behavior through a set of sequence diagrams. Considering the sequence diagrams in the left-hand side of Fig. 4 for the two different protocols, what we need to document is that the behavior in diagrams (a) and (b) cannot happen in the same program execution. As in the case of the conflict interaction, it is possible to specify a mechanism that implements the required mutual exclusion policy between the themes. This is shown at the right-hand side of Fig. 4 where the mutex arbiter theme is defined. Our arbiter theme intercept the setTime() behavior for both protocols and performs a check allowing to execute just one of them. Although it is possible to manually define the mutex, there is no explicit built-in support for the interaction.

4.4 Reinforcement

The error condition aspect reinforces the behavior of the communication protocols, reporting all error conditions to the remote servers. Considering this interaction, we have a situation similar to mutex: We model the communication protocol concern as a theme, and the error conditions concern as a theme, but we are unaware of a way in which to explicitly state the reinforcement semantics. In this particular case, we are able to integrate the reinforcement into the design, but at the cost of making the reinforcement implicit. We show this next.

The left-hand side of Fig. 5 shows a sequence diagram for the most severe type of error condition. It specifies how the error event occurring causes the tower lamp to be lit and the attendant to be called. Reporting the error to the server is specified in the right-hand side of Fig. 5 using a theme for the communication protocol. By binding both themes using the arrow construct, we define a crosscutting behavior of the communication protocol, specifying that it intercepts all calls of ErrorConditionBehavior.processSevere(Error).

However, as this states that the relationship between them is a typical crosscutting relationship, the reinforcement semantics is lost. Even though the generic behavior of the communication protocols captures all error conditions of this type, it is not clear that we know there may be new types of error conditions in the future, and each of them needs to trigger protocol behavior. This information is crucial to check the consistency of the system during maintenance and evolution. As the reinforcement semantics remains implicit here, this verification step might be omitted.

4.5 Scalability

As discussed in Sect. 2.1, an important feature of the modeling methodology we require is support for scalability. We need to be able to abstract over common patterns in the design, in this case in the different themes. We, however, only found one mechanism that allows for such abstraction: tem-

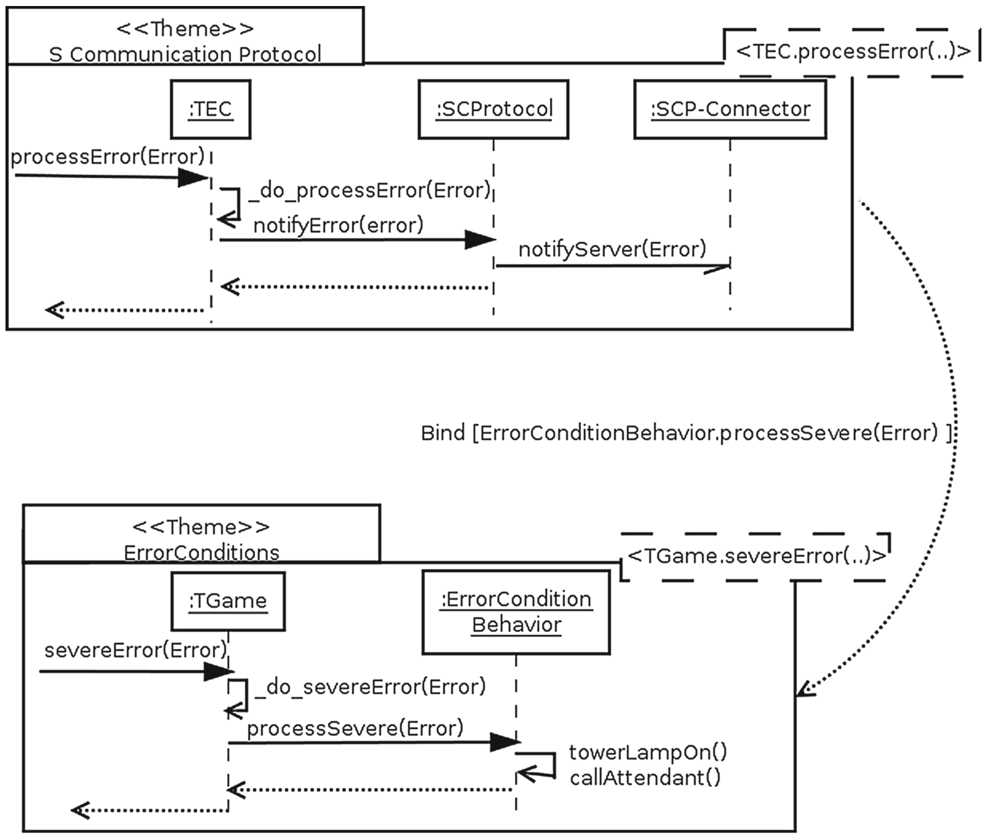

Fig. 5 SCP theme reinforced by error conditions theme

plate parameters for crosscutting templates. We now illustrate how it only addresses some of our scalability issues, using two of the examples we have seen above.

When producing the complete design, the error conditions theme shown in Fig. 5 needs to be bound to the occurrence of errors in a base theme. This binding expression can be a list of methods and may use wildcards as well. As such, this one theme is an abstraction over all events in the base code that trigger a severe error. Note, however, that if this theme would be a base theme, there would be no template instantiation. Therefore, we would need to manually produce a diagram for all events that produce a severe error, which does not scale.

The second example of a need for abstraction is found in the specification of the communication protocols in Fig. 4. The diagrams for both themes are the same, save for the initiating method call and the name of the protocol class. We need to produce such duplicate diagrams for a large amount of configuration setting functionality, as both protocols provide largely the same features. Since these themes are not crosscutting, there is, however, no template functionality available; hence, we must duplicate the work, which does not scale.

To summarize, the composition of crosscutting themes with the base themes gives us a means of abstraction, but it does not address all our needs, and therefore, Theme falls short in this respect.

4.6 Conclusion: Theme/UML

We found that Theme/UML does **not** allow us to express **any** of the four types of interactions in an explicit way. At the most, we are able to integrate support for mutex, conflict resolution, and reinforcement into the design. However, this comes at the cost of obscuring the explicit relationship between different aspects, which is likely to lead to errors during maintenance or evolution.

Concerning scalability, basic support is provided through the binding of crosscutting themes to multiple method calls in base themes. This, however, does not work for base themes and is therefore inadequate.

To conclude, as a result of the lack of interaction support, and only support for basic scalability, we consider Theme/UML inappropriate to specify the design of the slot machine.

5 Evaluation of WEAVR

WEAVR is an add-in extension to the Telelogic TAU (now Rational Tau [18]) MDE tool suite used by Motorola, adding support for AOM to their process of building telecom software [11,13]. The extension consists in providing support for aspect-oriented concepts: aspects, pointcut, and advice.

WEAVR is arguably the best-known industrial application of AOM, with claimed support for interactions.

Next to a UML notation, the Motorola tool suite also uses SDL [40] transition oriented state machines as the graphical formalism to define behavior. These state machines are unambiguous and allow for introducing pieces of code. This enables code generation of the complete application in C and C++.

The WEAVR pointcut notation is based on state machines, permitting the capture of *action* and *transition* joinpoints. Wildcards are allowed to refer to multiple states or actions. Advice is also expressed as state machines and is related to the pointcuts using the `bind` relationship. WEAVR is an aspect weaver: It combines an aspectual state machine with a base state machine when there is a join point match. The tool allows to visualize the new composed state machine so that engineers can verify the composition for correctness before actual code generation.

Note that although WEAVR can be used to generate the code of the application, we do not require this, and we only want to specify the design. Also, due to licensing issues we were not able to use the tool for our evaluation, instead relying on published work [11–13,38]. Lastly, even though SDL is a standard, the notation of its usage by WEAVR is not consistent among all the publications. The diagrams in this text are our best effort to produce a consistent notation, but we are not able to guarantee their notational correctness.

5.1 Dependency

Similar to the design in Theme, shown in Sect. 4.1, we have an interplay between the metering concern and the communication concern. The metering concern captures events regarding game activity and updates the meters, while the communication protocols consult data contained in these meters when processing server requests. In Fig. 6, we show the latter, for the G2S protocol. The action code response := Meters::GetCurrent() refers to data previously stored in the Meters object by the Metering aspect (which is not included in the figure due to the lack of space). The communication protocols thus depend on the meters to provide correct functionality. Put differently, if the Meters object is available but for some reason the behavior of the metering aspect is not executed, the data returned will be inconsistent.

To declare dependency relationships, WEAVR provides the depends_on relationship that is used in the deployment diagrams, where aspects are applied to classes. This relationship states that one aspect depends on another to be able to provide the required functionality. This relationship, however, only applies at the join point level: If AspectA depends_on AspectB, for each shared join point the advice of AspectB will be executed before the advice of AspectA. Additionally, if AspectB does not match a join point matched by AspectA, the match of this joinpoint by AspectA is silently discarded [38]. In this way, AspectB can considered as adding a restriction on the matching of pointcuts of AspectA.

In our case, however, the contact point between two aspects is the existence of the Meters object, not a shared join point. As a consequence, the depends_on relationship does not allow us to express the required dependency. This is as the semantics of the depends_on relationship is too fine grained. In our case, we need to be able to express this relation at the level of aspect deployment, e.g., state that the deployment of AspectA implies the deployment of AspectB. WEAVR does not provide any other dependency construct, and we are not aware of an alternative option to relate the state diagrams above. We are therefore unable to include the dependency specification in the design.

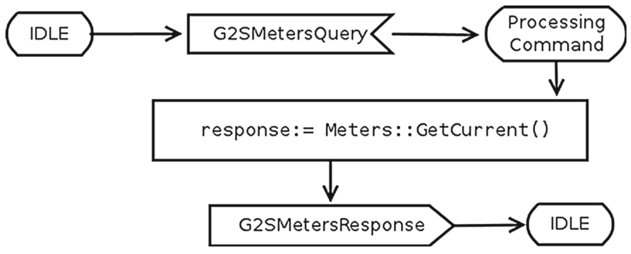

Fig. 6 Part of the G2S protocol state machine depending on meters

5.2 Conflict

Support for conflict resolution in WEAVR is realized by the hidden_by stereotype that is also used in the deployment diagrams. The hidden_by stereotype relates two different aspects that intercept the same join point. The relationship states that the aspect that is hidden does not apply in those cases. For example, specifying AspectA hidden_by AspectB denotes that at a join point captured by both aspects, only the behavior of AspectB will be executed. In other words, we can state that the presence of one aspect implies the absence of another aspect, but again only at the level of join points.

In our case, such conflict resolution is, however, not sufficient as we are faced with aspects that conflict when active on different join points. For example, consider Demo: When it is active, the different protocols must return a failure message upon a query of the server, which is a different join point than starting a play. We require instead of a hidden_by semantics that works at join point level, a similar semantics at the system or aspect level; that is, the activity of Demo should imply the inactivity of the G2S and S Protocol, or the report of the machine as being *out of service* (communication protocols have specific messages for this purpose).

Similar to the work-around for Theme we proposed in Sect. 4.2, we can provide a design that incorporates the

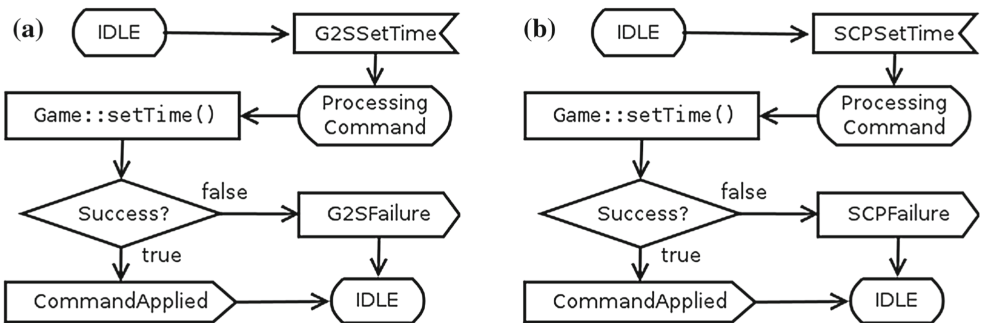

Fig. 7 Mutually exclusive state machines for the setTime command. **a** State machine for G2S, **b** state machine for SCP

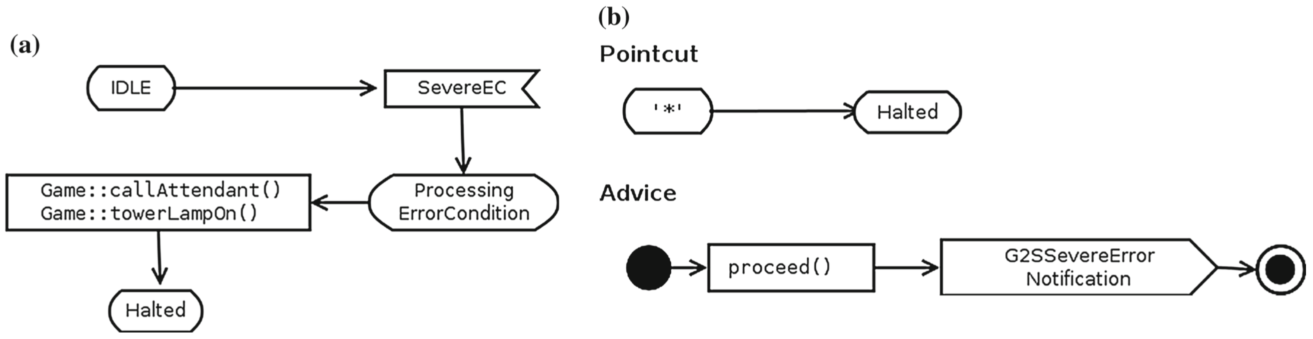

Fig. 8 Reinforcement for tilt error condition on G2S communication protocol. **a** Tilt error condition, **b** notification to the monitoring system

required conflict resolution behavior. Advice in WEAVR is always around advice, and use a proceed call. As in Fig. 3, we can specify an around advice that intercepts Meters and the communication protocols, without performing the original behavior of the intercepted call. This work-around consequently suffers from the same drawbacks as in Sect. 4.2, most importantly the loss of the explicit conflict specification.

5.3 Mutex

Recall that our mutual exclusion consists of the prohibition that the same configuration item of the SM can be set by multiple protocols. As an example, Fig. 7 shows the design of the setTime functionality for both protocols in WEAVR. If we would have, e.g., the first-come first-serve rule, the mutual exclusion in this case boils down to preventing that the state machine of Fig. 7a executes if the state machine of Fig. 7b was previously run and vice versa. However, WEAVR does not provide for any way in which this can be specified.

It is feasible to produce a design document that implements the mutex, but at the cost of making the explicit information of the mutex implicit. We can manually combine the different state machines for the different protocols such that the mutex relation is implemented. Briefly put, for each configuration action we combine the two state machines of the different protocols into one state machine. This combined machine contains the functionality of both protocols together with the logic that ensures that once the item has been configured by one protocol it cannot be configured by the other.

The downside of this solution is that it adds a considerable amount of tedious work, combining the state machines for all configuration settings, and obscures the intent of the design. Moreover, it produces a design where both protocols are tightly coupled. Consequently, we consider this option unfeasible and discard it.

5.4 Reinforcement

The design of the reinforcement from error conditions to communication protocols is similar to the design in Theme/UML discussed in Sect. 4.4. We have an error conditions aspect that handles the different types of errors that occur. Figure 8a shows how this aspect calls the attendant and switches the tower lamp on, after that the system passes to a halted state. Figure 8b shows an aspect that intercepts the transition to the halted state and reports the error. Even though we were able to express the desired notification of error conditions, we have, however, not found a means to explicitly denote the reinforcement relationship as such.

As in Sect. 4.4, the downside of this is that the explicit reinforcement relationship has become implicit, which may lead to inconsistencies during maintenance and evolution, e.g., when new types of errors are added to the system. An upside of using WEAVR is that its model simulation capabilities allow for consistency checking of the composed models. This could corroborate the whole execution path from the occurrence of a new error condition to the final notification to the server. However, the need for such a verification for all types of error conditions still has to be specified in the design document, and we are unaware of a means to express this in WEAVR.

5.5 Scalability

State machines can grow to be complex artifacts. In our experience, this holds especially in WEAVR where they are aimed at generating a complete model, from which it is possible to derive the code of the final system.

Such complexity can only be mastered using tools that help the engineer to edit, simulate, and debug the behavior of the complete system. In this regard, WEAVR has a strong point as it has been built as an add-in to Telelogic TAU. According to Cottenier et al. [11,38], WEAVR can use Telelogic's infrastructure of the simulation of models. As Telelogic TAU [18] does not natively support aspects, WEAVR requires the weaving of the models before its execution can be simulated. This, however, further complicates the generated models, negating the advantages of separation of concerns when expressing the aspects separately.

On the notational side, we have observed that in our domain the behavior of several state machines (aspectual or not) is similar. Consider, for example, the execution of protocol commands. Essentially these state machines are identical. However, as far as we know, WEAVR does not provide any mechanism for the abstraction of similar behavior. This makes it necessary to develop individual state machines instead of a single one that is parameterized and instantiated as needed. Note that this problem not only affects the base system and the aspects, but also the implementation of interaction resolution aspects, as explained in Sect. 3.2.

To summarize, tool support for WEAVR could help somewhat but has the downside that only woven code is simulated, and the lack of abstraction mechanisms is an important impediment to scalability.

5.6 Conclusion: WEAVR

We have seen that WEAVR does **not** allow us to explicitly express **any** of the four interaction types. If we allow making the explicit relations implicit, we can include support for conflict resolution and mutual exclusion in the design, the latter of which would be a large amount of tedious work. Such implicit relations, however, come at a cost of probable errors during maintenance or evolution.

Considering scalability, WEAVR state machines can become large and complex, making them hard to understand. Also, the language does not provide for any abstraction operators, which forces us to duplicate a significant amount of almost identical state machines.

As a result of these issues, we consider WEAVR unsuited to specify the design of a slot machine.

6 Evaluation of RAM

Reusable aspect models (RAM) [22] is an aspect-oriented multi-view modeling approach. In multi-view modeling, the developer describes the system being modeled from multiple points of view, using the modeling notation that is most appropriate for the part of the system being modeled. To obtain the final model of the complete system, all the different views are composed into one whole. Scalability is a significant issue in multi-view modeling, as models tend to grow rapidly and keeping them consistent, or analyzing the many interrelated models is difficult [22]. RAM aims to address this scalability issue by incorporating the advanced modularity features of aspects into a multi-view modeling approach.

RAM models consist of a joining of a class diagram, sequence diagrams, and state diagrams into one model, where these diagrams may use AOM features, e.g., pointcuts and advice in the sequence diagram. As a result of the latter, RAM is an aspect-oriented modeling technique. This is so even though RAM is not conceived for the modeling of aspects, like the two modeling techniques we discussed above. Instead, the fundamental goal of RAM is the reuse of models and the features of the language are geared toward this. Furthermore, RAM has as a guideline that each RAM model should be designed to be maximally reusable. To achieve this, it should be as compact and self-contained as possible. Ultimately, a RAM model will (ideally) be composed many times to form many different applications, and thus, the model cuts across these different applications. To emphasize this, RAM models are called aspect models.

We wish to highlight at this point that in our experience, RAM aspect models are initially confusing. The confusion arises because RAM aspect models do not necessarily crosscut the application structure, which is what is traditionally expected of aspects. Pointcuts and advice are not present in all RAM aspect models, and moreover, when present, does not necessarily crosscut the application structure. For example, in the large majority of the pointcuts of the SM models, pointcuts pick out just one specific method call. Thus, for clarity of this text, we emphasize: RAM aspect models are aspects because these models cut across different applications.

Since the reuse of models is key in RAM, multiple features of RAM are built to facilitate such reuse. We detail them here, and they are also highlighted in Fig. 9, which we will discuss subsequently.

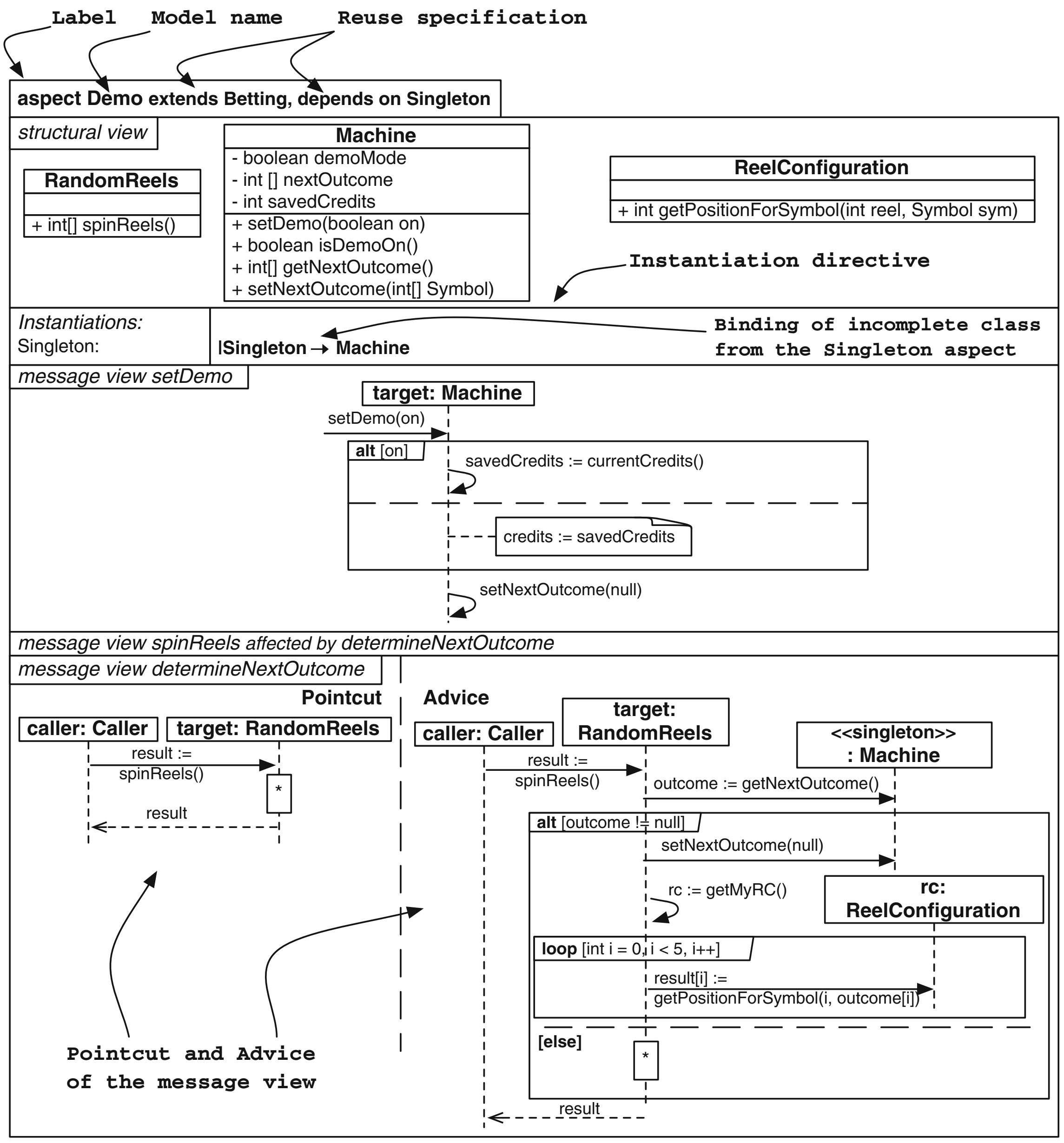

Fig. 9 A part of the Demo aspect in RAM, where different RAM features are identified by annotations

Reuse of Other Diagrams The label of each aspect model gives the model a name and also specifies whether this aspect model reuses other aspect models. By reusing another model, this aspect model either provides an extension of the reused model, or uses that model to realize part of its implementation. We give more detail in Sect. 6.1 on the different types of reuse.

Instantiation Directives When reusing another aspect model, classes and methods of the reused aspect model can be bound to classes and methods of the current model. This allows to unify different classes (or methods) from the different diagrams into one class (or method), or to perform a renaming of the reused elements to avoid name clashes.

Incomplete Classes Classes in a structural view do not need to be complete with regard to how they ultimately will take form in the composed application. To be more reusable, they only need to specify what is needed for the concern being modeled in the current aspect model, leaving all other functionality blank. Such partially specified classes are called incomplete classes, and their name should start with a | character to highlight that these are meant to be unified with other classes when the model is composed. Furthermore, incomplete classes of a model also need to be drawn as UML template parameters on the top right-hand side of a model. Similarly, methods can be given a placeholder name and signature. In that case, their name should also start with | and their class is automatically an incomplete class. When the aspect model with such classes or methods is reused, these must be bound in instantiation directives or else they will remain incomplete in the reusing aspect model. In a complete composition of aspect models, i.e., the final application, there may be no incomplete classes.

Pointcuts and Advice Sequence Diagrams in RAM may be divided in pointcut and advice. Both of these concepts are the classical AOP concepts of pointcut and advice, placed in a modeling setting. The fundamental goal of the pointcut here is to simply decouple the behavior expressed in this aspect model from the model of where the pointcut matches, this again to increase reuse. Note that as a consequence of the presence of pointcuts and advice, aspects in the classical AOP sense can also be expressed as RAM models. Hence, RAM can also be used to model aspects in the more conventional term, i.e., to model aspects as we performed in the previous two sections.

Since RAM is arguably more dissimilar from mainstream modeling approaches than the previous two AOM approaches we discussed, we include an example aspect model to describe the various features of RAM that we have used when modeling the SM. The example is shown in Fig. 9, illustrating part of the functionality of Demo. On the top left of the figure is the label, identifying the name of the aspect as well as giving the reuse specifications (discussed in more detail in the next section). This is followed by a structural view: A standard UML class diagram describes the classes used in this aspect. Below this, an instantiation directive specifies that the incomplete class |Singleton, reused from the Singleton aspect, is unified with the Machine class. In other words, the Machine class now contains all functionality that is specified for the |Singleton class in the Singleton aspect. Next is a plain UML message view, named setDemo. It specifies that when Demo is switched on, the credits of the machine are saved, and when switched off this amount is restored. In addition, any predetermined outcomes of spinning the reels are erased. The other message view, determineNextOutcome, contains a pointcut and an advice. Because of this, it is preceded by an explicit announcement of which message view the pointcut is expected to match: the spinReels message view (of the Betting aspect). Note that this announcement is to be treated as additional information only, and it does not prohibit the pointcut to match other message views. The determineNextOutcome message view replaces the behavior of the spinReels message view with new behavior. The result is that if a next outcome has been set (which can only be done in Demo mode), the reels will show this outcome when they are spun.

After initial tests and a prolonged conversation with the main authors of RAM, we set up a (self-funded) short research visit to their laboratory. In this visit, we modeled a large part of the slot machine in RAM, in collaboration with the main authors of RAM. The diagrams were, however, not completed during this visit, and further work was carried out subsequently. For our evaluation, we chose not to use state diagrams, as the combination of class and sequence diagrams already showed to be sufficiently expressive.

6.1 Dependency

At its core, RAM includes the concept of one aspect reusing another (set of) aspect(s). This relationship can either be an augmentation, declared by an **extends** annotation in the label, or a customization, declared by a **depends on** annotation in the label [5].

When an aspect E extends another aspect O, additional structure or behavior (or both) is added to O. These additions can be "additional, alternative, or complementary properties to what already exists" in O [5]. The public interface of the aspect E is the public interface of O augmented with the public operations declared in E.

When an aspect D depends on another aspect O, the structure and behavior of O is used to realize part of the functionality of D. In the customization, "a modeler alters or augments existing base model properties to render them useful for a new purpose" [5]. It is considered that typically D implements a higher-level functionality that uses functionality of O. For example, a communication protocol aspect uses the aspect defining serialization and deserialization of network command objects. The aspect D effectively uses O and is restricted to access only the public model elements of O. Moreover, this use is private; the public model elements of O are not present in the public interface of D.

The RAM definition of **depends on** matches exactly with our requirement of a dependency relation. Hence in our example, communication protocols **depends on** meters, in the RAM sense. This is because the abilities of meters are adapted such that their values can be transmitted over the network. As such, RAM is the first of the three AOM methodologies where we can explicitly declare the dependency and, moreover, the **depends on** notation is in line with the naming of this interaction as a *dependency* in our classification. Figure 10 shows the design of the network aspect, declaring explicitly **depends on Metering** in the label.

Figure 10 does not give the complete design for the network aspect. For the sake of the example, it is restricted to the commands that read meters. In the structural view, it shows which classes from Metering are used, and both message views detail how these are used. The first message view specifies that when a command is received from the network, the corresponding SMCommand (a Command object) is created and executed. Lastly, if the execution produces a result, this result is sent back over the network in a separate communication. The second message view states how command that read

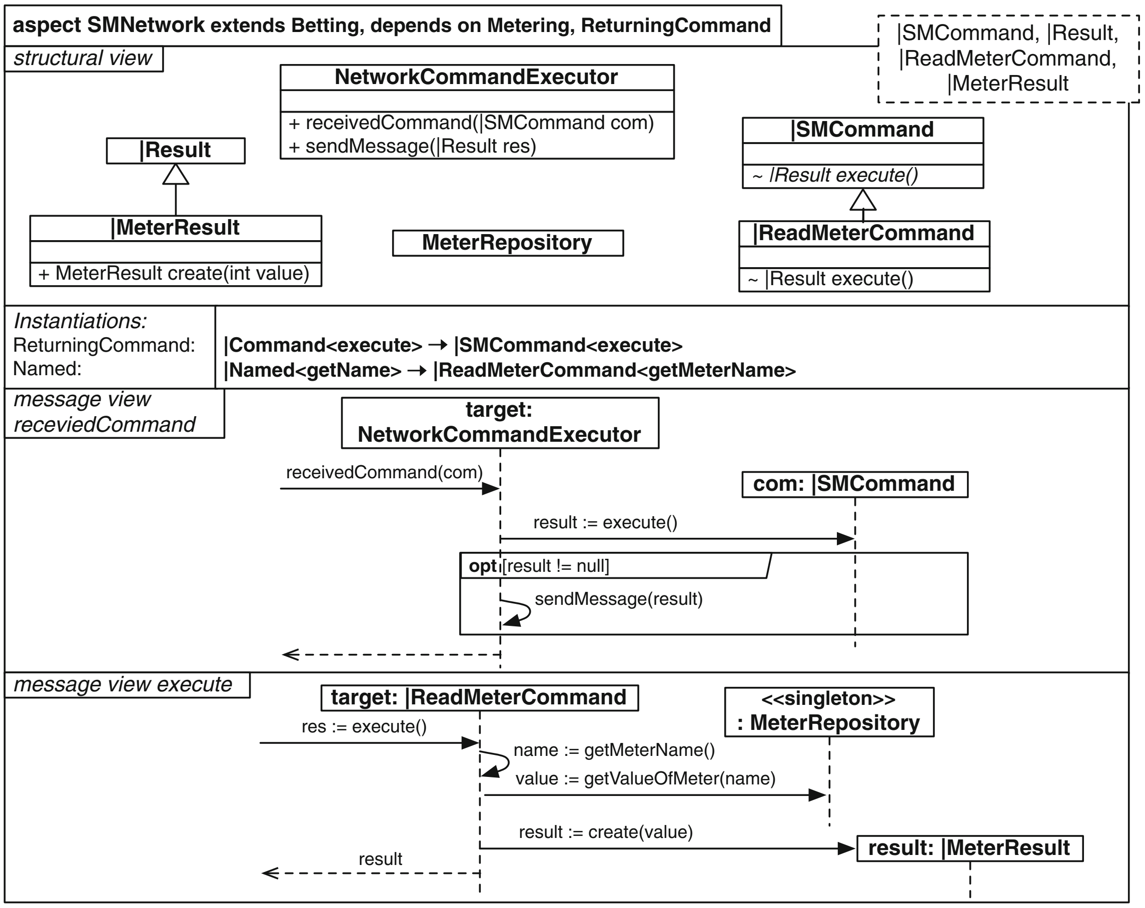
aspect SMNetwork extends Betting, depends on Metering, ReturningCommand
|SMCommand, |Result, |ReadMeterCommand, |MeterResult
structural view
NetworkCommandExecutor
+ receivedCommand(|SMCommand com)
+ sendMessage(|Result res)
|Result
|MeterResult
+ MeterResult create(int value)
MeterRepository
|SMCommand
~ |Result execute()
|ReadMeterCommand
~ |Result execute()
Instantiations:
ReturningCommand: |Command<execute> → |SMCommand<execute>
Named: |Named<getName> → |ReadMeterCommand<getMeterName>
message view receviedCommand
target: NetworkCommandExecutor
receivedCommand(com)
com: |SMCommand
result := execute()
opt [result != null]
sendMessage(result)
message view execute
target: |ReadMeterCommand
<<singleton>> : MeterRepository
res := execute()
name := getMeterName()
value := getValueOfMeter(name)
result := create(value)
result: |MeterResult
result

Fig. 10 Part of the slot machine network, which depends on the meters

meters should be implemented. Note that the top right-hand side of the diagram highlights the incomplete classes: the |SMCommand class, the |Result class, and their subclasses, by drawing them as UML template parameters. These classes are partial because their actual implementation is delegated to the network protocol that will be used (SCP or G2S).

This aspect model is, however, not without its issues. This is because, in contrast to the typical use of **depends on**, the network aspect is not a higher-level design concern than metering. Instead both of these are at the same level, as we have established at requirements elicitation time [36]. Conversely, the dependency on ReturningCommand, however, is a typical use: It is the reliance on the implementation of network commands we used as an example in the presentation of the **depends on** relation. This illustrates the downside of the dependency declaration in RAM: From the declaration, it is unclear whether the dependency arises from functional requirements or whether it is an implementation issue.

The inability to distinguish between functional and implementation dependencies is, in our opinion, a serious issue. In our case study, the RAM model of the slot machine consists of 16 aspects and depends on an additional six aspects. In total, 20 **depends on** relations are declared of which only two are functional dependencies, i.e., caused by functional requirements. These two are effectively lost in a forest of dependencies, and as a result, we cannot consider the use of the **depends on** as a true explicitation of the dependency relationship we require.

6.2 Conflict

RAM also includes support for managing conflicts between different aspects, in the form of conflict resolution aspects: aspects whose label is **conflict resolution aspect** (instead of simply **aspect**). The sole purpose of conflict resolution aspects is describing how the conflict needs to be resolved. Hence, the presence of a conflict resolution aspect is an explicit indication that there is a conflict, satisfying our requirement for such explicit indications.

Conflict resolution aspects define what are termed “modification views” instead of the standard views [23]. These views modify the aspects affected by the conflict, but only if the *conflict criteria condition* of the conflict resolution aspect holds. Put differently, the role of this condition is to specify in what case the conflict is present, and the conflict resolution aspect is automatically woven when the condition holds.

The purpose of conflict resolution aspects appears to be resolving conflicts when there are different possible configurations of functionalities of the application being built. In some configurations, the aspects which are in conflict will not be both present, and in some configurations, they will be present. Hence, there is a need for conflict criteria condition to trigger automatic weaving of the conflict resolution aspect when required. In our case, however, all configurations of the slot machine will include the conflict between Demo and Network, as both features are always present. This requires us to specify a conflict criteria condition which will always hold, which is arguably superfluous. Hence, for the definition of this conflict resolution aspect, in Fig. 11, we have chosen not to include this criterion, assuming this means that it always holds.

Figure 11 first shows how the slot machine should reply to network queries when it is in demo mode. The outOfService message view specifies that when commands are received and demo mode is on, the command should return an out of service result. Second, the figure shows that error messages should not be sent over the network when the machine is in demo mode. This as the muteErrors message view states that in demo mode the behavior of message sending by the NetworkCommandExecutor should be skipped.

In summary, we are able to specify conflicts using RAM in an explicit fashion through the specification of an aspect that provides the resolution of the conflict. For conflicts that are always present, needing to specify a conflict criteria condition is, however, arguably superfluous, so these should be allowed to be omitted.

6.3 Mutex

The mutex interaction in our case study manifests itself when both network protocols are present in the application. When both protocols are present, both can execute commands that alter the machine state, and hence, mutual exclusion between these commands must be ensured. When the machine only includes one protocol, there is no mutual exclusion. This exactly matches the intent of conflict resolution aspects. Therefore, by specifying a conflict criteria condition that matches when both communication protocols are present, the mutex will automatically be instantiated. Figure 12 shows the relevant conflict criterium: when the slot machine command can be a command of the SCP protocol as well as a command from the G2S protocol.

The conflict resolution strategy presented in Fig. 12 is detailed in the arbitExec message view. It consists fundamentally in performing a check of whether the command may be executed, executing it if so, and returning an error if not. The implementation of the check is implemented by an arbitration class called Arbiter, and this class may implement the check following any of the strategies presented in Sect. 3.2. Note that for the sake of brevity of this article, the figure does not detail the behavior of logging the error.

We are hence able to specify the implementation of the mutual exclusion semantics and also benefit from the automatic application of the conflict resolution aspect, such that it is only present when needed. What is, however, not present here is the explicit semantical information of this being a *mutex*. Instead, this interaction is considered as being a conflict. Consequently, we lose the *mutex* classification of this interaction; i.e., we lose semantic information.

6.4 Reinforcement

Lastly, reinforcement can also be expressed as a conflict resolution aspect, hence also benefiting from the explicit notation of the interaction in the design. The reinforcement from error conditions to communication protocols, however, is not a true conflict resolution aspect. It is more in line with the conflict resolution aspect of the conflict case because there will always be at least one network protocol. Hence, the conflict criteria condition must always match and we omit it in the model, presented in Fig. 13.

The conflict resolution aspect realizes the reinforcement in the sendCritical message view. This message view extends the normal behavior of IReportingInterface, which is the class that is responsible for reporting errors to the outside world, e.g., by lighting the tower lamp of the SM. The addition to the behavior consists in creating a result message that encapsulates the error and sending it over the network. Note that the requirements for this aspect include a check that the error condition is either sent over the network or intentionally not treated. For brevity's sake, this feature is not included in this aspect model. It can be specified, e.g., by having two kinds of error results and extending the sendCritical message view to only send one of both kinds over the network.

So, identical to the mutex case, we are able to state that there is a reinforcement and provide the implementation of the reinforcement, but lose the explicit semantic information that this interaction is of the *reinforcement* kind.

6.5 Scalability

Considering scalability in RAM, our first observation is actually a matter of managing the different RAM diagrams. Due to the fine-grained and detailed nature of RAM, we quickly reach a large number of diagrams. As mentioned before, our modeling of a limited part of the machine already results in 16 aspects. In total, these aspects have 30 dependencies (either through **depends on** or **extends** relationships). At least some kind of overview diagram is required to be able to understand this web of aspects and dependencies. Such depen-

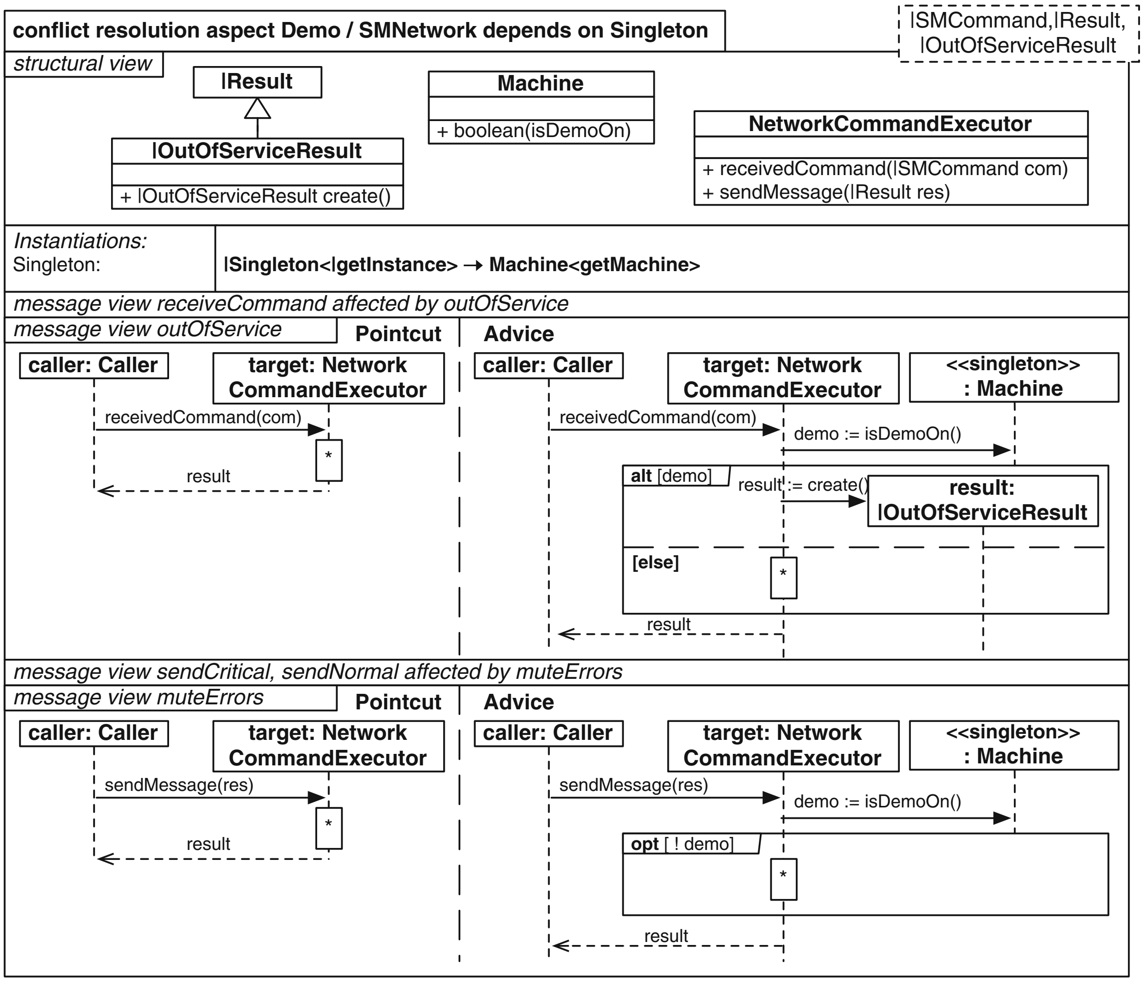
conflict resolution aspect Demo / SMNetwork depends on Singleton
|SMCommand,|Result, |OutOfServiceResult
structural view
|Result
|OutOfServiceResult
+ |OutOfServiceResult create()
Machine
+ boolean(isDemoOn)
NetworkCommandExecutor
+ receivedCommand(|SMCommand com)
+ sendMessage(|Result res)
Instantiations:
Singleton:
|Singleton<|getInstance> → Machine<getMachine>
message view receiveCommand affected by outOfService
message view outOfService
Pointcut
Advice
caller: Caller
target: Network CommandExecutor
receivedCommand(com)
result
<<singleton>> : Machine
demo := isDemoOn()
alt [demo]
result := create()
result: |OutOfServiceResult
[else]
message view sendCritical, sendNormal affected by muteErrors
message view muteErrors
sendMessage(res)
opt [! demo]

Fig. 11 Slot machine demo network conflict resolution aspect

dency graphs are present in the publications on RAM [22,23], but it is, however, not clear if such diagrams are considered to be a standard notation of RAM.

The most explicit support for scalability in RAM is support for wildcards in instantiation directives. This can unify one generic class with a family of classes providing specific functionality. For example this can be used to bind the general SCP network command class (SCPCommand) to all of the concrete SCP commands. As the example shows, this feature is indeed useful in our case and does help significantly to deal with issues of scalability. We have used it to abstract general patterns of the design in one class that is later bound to a collection of more specific classes.

6.6 Conclusions

Although it does not fully meet our needs, RAM has shown to be the most adequate AOM approach for describing aspectual interactions in the design of the SM. We were able to express the four kinds of interactions in the design, but these are not completely explicit.

Dependencies between different aspects could be explicitly declared using the **depends on** relationship. However, this relation is less strict than the dependency relationship we mean to communicate. As a result of this, in our models only a select few of these declarations actually coincide with the desired meaning. Considering conflicts, mutex, and reinforcement, all three of these are declared in an conflict resolution aspect. This immediately implies defining their resolution strategy, but comes at the cost of not differentiating between the three different kinds of interactions.

Considering scalability, firstly RAM requires the use of an overview diagram to be able to understand the relationships between the different aspects. Secondly, the use of wildcards in instantiation directives does successfully allow us to tackle scalability at the level of functionalities to model.

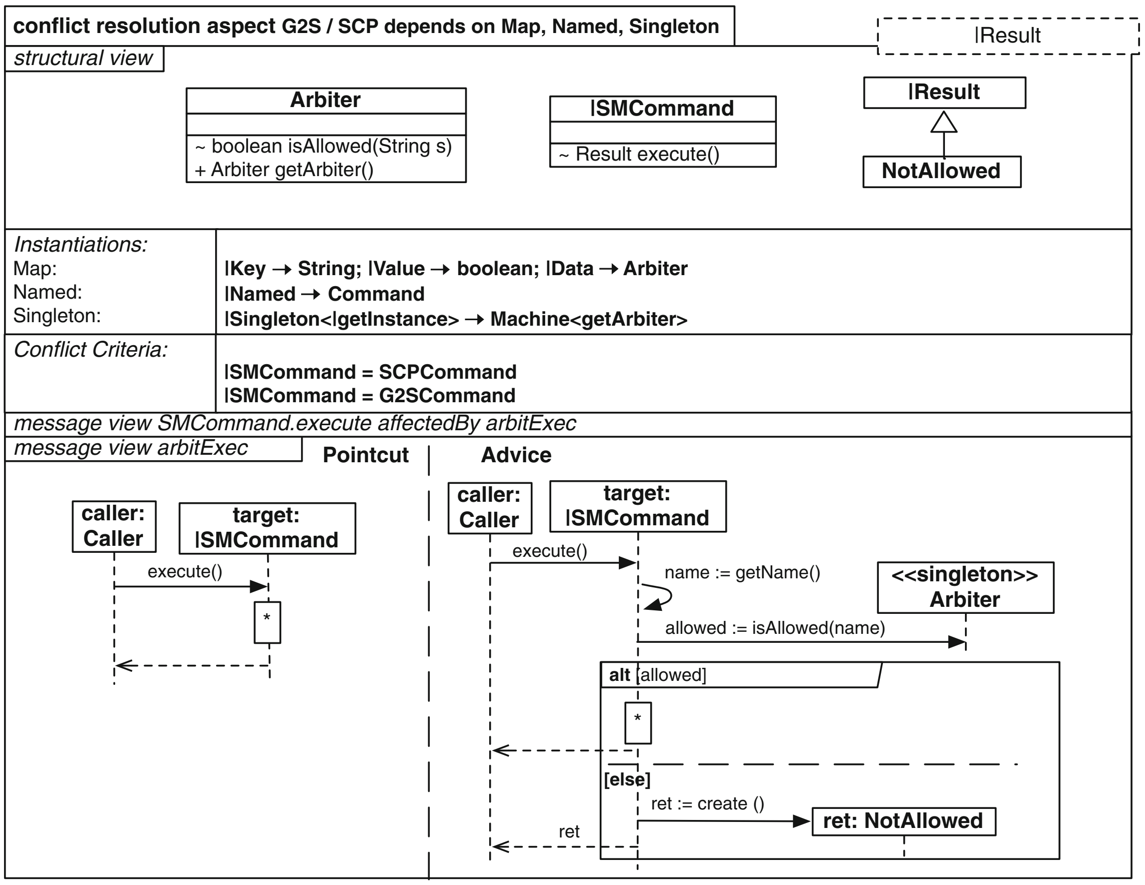

Fig. 12 Slot machine network conflict resolution aspect

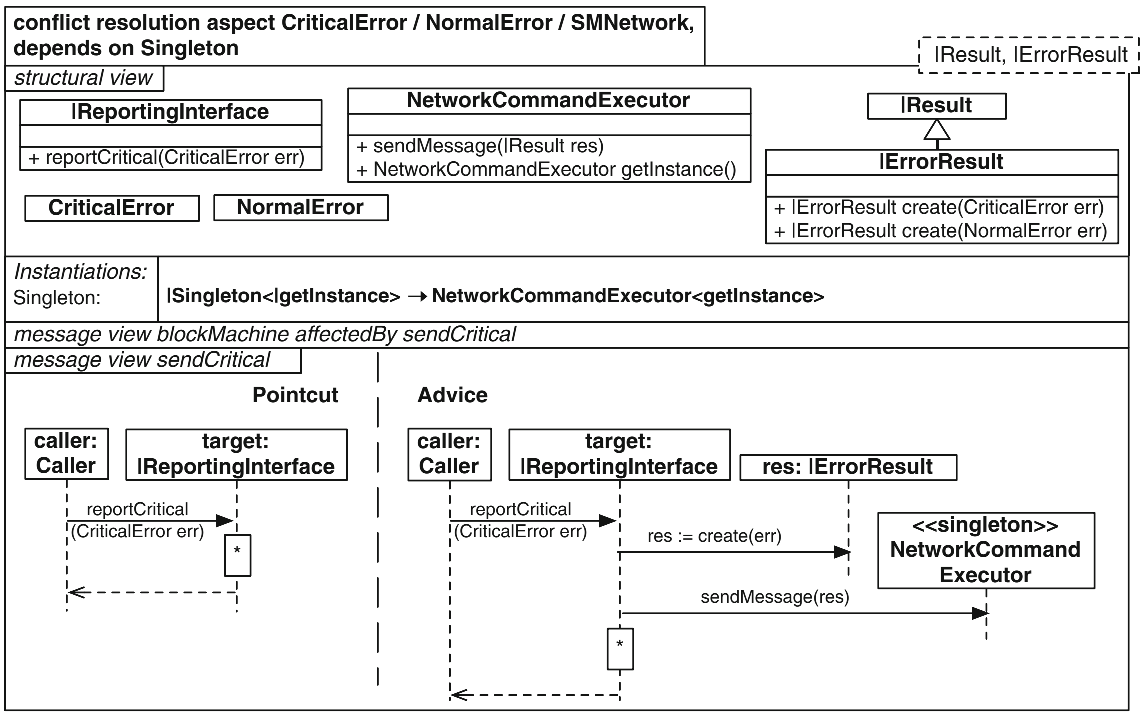
conflict resolution aspect CriticalError / NormalError / SMNetwork, depends on Singleton
IResult, IErrorResult
structural view
IReportingInterface
+ reportCritical(CriticalError err)
NetworkCommandExecutor
+ sendMessage(IResult res)
+ NetworkCommandExecutor getInstance()
IResult
IErrorResult
+ IErrorResult create(CriticalError err)
+ IErrorResult create(NormalError err)
CriticalError
NormalError
Instantiations:
Singleton: ISingleton<IgetInstance> → NetworkCommandExecutor<getInstance>
message view blockMachine affectedBy sendCritical
message view sendCritical
Pointcut
Advice
caller: Caller
target: IReportingInterface
reportCritical (CriticalError err)
*
res: IErrorResult
res := create(err)
<<singleton>> NetworkCommand Executor
sendMessage(res)

Fig. 13 Slot machine network errors conflict resolution aspect

7 Discussion

We have reviewed three AOM approaches with a focus on their ability to express the different kinds of interactions that are present in our domain. In this section, we summarize the strong points of the different approaches and highlight how the different kinds of interactions are materialized in each one. This is followed with a comparison of this work with research on interactions in arguably closely related research: Feature-Oriented Software Development and Software Product Lines.

7.1 Strong points and cross-pollination

Each of the three approaches we have evaluated has its own strong points, and there are, moreover, opportunities for cross-pollination between the different approaches. We now first revisit the different strong points in order to highlight the best features of each approach in the context of the best features of the others. This may help to choose a specific approach depending on the circumstances of its use.

The strong points of Theme/UML are:

- The integration with an AORE approach (Theme/Doc) makes Theme/UML relevant when aspect orientation can already be considered at the requirements engineering phase.
- The mapping from requirements to models is quite well documented, facilitating this task.
- In [36], we proposed extensions to Theme/Doc in order to support interactions. It is then possible to document them during the requirement engineering phase so that they can be straightforwardly used in the design phase.

The strong points of WEAVR are as follows:

- The existence of built-in interaction resolution mechanisms in WEAVR indicates that joinpoint interactions were found in its use in the *telecommunication* domain, where WEAVR has its origins. It is no surprise that such interactions were found in this domain, as similar cases have been reported since 1988 in [17], and in a more recent bibliography [6,28].
- The SDL aspectual machines weaver helps to check the consistency of the generated model before code generation occurs. A composed model makes it possible to confirm positive interactions to be there, while detecting undesirable interactions.
- The support of WEAVR for C++ code generation is a plus. According to the authors, Telelogic TAU (of which WEAVR is an extension) was successfully used for embedded telecom software.

The strong points of RAM are as follows:

- Models can be explicitly tagged to be for conflict resolution; i.e., it is explicit that this model exists to resolve issues that arise due to interactions between other models.
- The models are highly reusable, and there is a library of models available for the modeler to incorporate.

Cross-pollination of Approaches The different approaches we reviewed each have characteristics that could be added to other approaches to improve them. We highlight here three such possible cases of cross-pollination that, in our view, would improve the different approaches:

Explicit interaction resolution tagging: Identifying models as existing for the resolution of an interaction, as in RAM, is a significant step toward making explicit that one of the different kinds of interactions is present.

Automatic weaving: (as in WEAVR and RAM) Automatic weaving is useful, especially if the generated model can be checked for interaction satisfaction (as in WEAVR).

Interaction support starting at the requirements engineering phase: Integrated approaches that document interactions beginning with early stages in the development life cycle should allow for better traceability. Although Theme/UML does not support interactions, Theme/UML could be extended in order to take advantage of existing extensions in Theme/Doc for interactions.

7.2 Summary: Interactions in the AOM approaches

Considering the results we obtained during the modeling phase using the three AOM approaches, we now summarize their support for interactions as well as the form that these interactions and their resolutions take.

Table 1 shows how the interactions in the slot machine domain can be expressed in each approach. For all the approaches, it was possible to express the interaction and/or the associated resolution mechanism in some manner. The main problem we encountered with the different AOM approaches is the lack of **explicit** support for documenting the interactions. Considering WEAVR, it is worth mentioning that it does provide interaction resolution mechanisms, yet these are aimed at join point interactions. This renders them ineffective in our case study, which is why they are marked as "implicit*" in the table.

Table 2 presents the form that interactions and their resolution take when modeled using each approach, where:

Lang	means that some built-in support in the AOM approach is used to model interactions and their resolution mechanism.

Table 1 Implicit and explicit support for interactions in each approach

	Dependency	Conflict	Mutex	Reinforcement
Theme/UML	Implicit	Implicit	Implicit	Implicit
WEAVR	Implicit*	Implicit*	Implicit*	Implicit
RAM	Explicit	Explicit	Implicit	Implicit

Table 2 Form of the interactions for each approach

	Dependency	Conflict	Mutex	Reinforcement
Theme/UML	Use	Artifact	Artifact	CC
WEAVR	Use	Artifact	Artifact	CC
RAM	Lang, Use	Lang, Artifact	Artifact	CC

Use	means that one element in one concern model *uses* another element as a resolution mechanism; i.e., there is a reference to another element.
CC	means that the resolution mechanism is implemented as a standard crosscutting relationship of the approach.
Artifact	means that the interaction resolution is specified as an additional artifact, e.g., a model diagram.

Note that certain interaction resolutions in Table 2 can ultimately materialize themselves in a more subtle manner than others. At implementation level, a *Dependency* is just a reference to another module, while *Reinforcement* may be implemented as a crosscutting concern. This can result in them being implicit, making more evident the need for proper documentation. Complimentarily, *Mutex* and *Conflict* require a dedicated artifact in order to implement the resolution of the interaction; that is, their resolution needs to be mapped to an aspect, theme, or state machine that implements the behavior needed to satisfy the interaction. In this way, although the interaction is still implicit it is somewhat more evident.

7.3 Interactions in Feature-Oriented Software Development and software product lines

In this section, we contrast how interactions need to be treated in our domain with their treatment in the related research areas of Feature- Oriented Software Development and Software Product Lines.

7.3.1 Feature-Oriented Software Development

Feature-Oriented Software Development (FOSD) is a field of research that encompasses feature-oriented domain analysis (FODA), feature- oriented modeling (FOM), and feature-oriented programming (FOP). There are, however, multiple definitions of what a *feature* is in software engineering. For the following discussion, we consider the definition of Kang et al.: "a distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained" [19].

Clearly, all the concerns from our case study match the definition of a *feature*. Therefore, it is also possible to consider the interactions we found as feature interactions [6,7,26]. In FOSD, feature interaction refers to how features influence each other and the feature interaction problem lies in detecting, managing, and resolving interactions among features [4].

Most of the research we found considers feature interactions to be roughly equivalent to what we call dependencies [27,34,39]. Only some of the work on feature interactions matches the aspect interaction definitions of Saenen et al. [33] that we use to classify interactions. Notably, Metzger et al. [27] refine the definition of dependency interaction into: *requires dependency*, *exclusive dependency*, *hints dependency*, and *hinders dependency*. These map to dependency, mutex, reinforcement, and conflict, respectively.

Some authors focus their research on the detection of such interactions, using algebras, graphs, and other representations, but not on how to model them explicitly [7,27,30]. Complementarily, other authors approach the problem from the code perspective. From this perspective, interactions must be managed in order to clearly state their impact in the source code [2,24,32].

Calder et al. [7] have reviewed interactions in the telecommunication domain and classified several approaches for handling feature interactions in this domain. The kind of mechanisms for mutex and conflict that we designed falls into the category of *Feature Manager* using *A Priori Information*. However, these proposed solutions are intended to be used to organize a telecom system at runtime, instead of being oriented to work at a modeling phase.

Similar to our requirement of explicit documentation, Silva et al. acknowledged the need for a proper documentation of the interactions and dependencies between features [34]. They propose to use a separate diagram to document such interactions. In this spirit, it is reasonable to use feature diagrams as a complement to an AOM approach, allowing for the explicitation of the interaction relationships. For example, Fig. 14 shows such a feature diagram for our case study. However, such a diagram on its own is not sufficient as not all the interactions can be expressed. Consider the conflict between Demo and the other features: All of them are required, and the conflict must be managed at runtime. This cannot be denoted in the diagram.

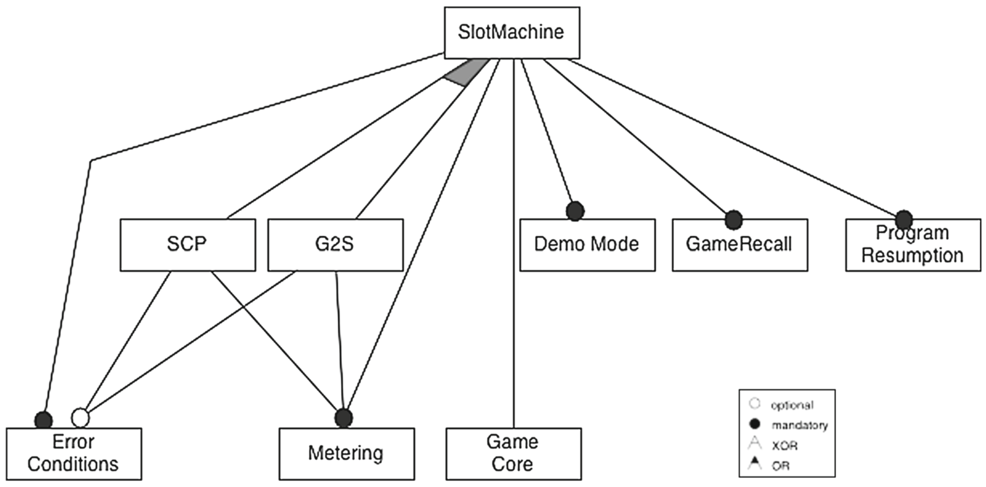

Fig. 14 Features diagram for the slot machine domain

Finally, note that in Sect. 7.2 we concluded that interactions can take different forms at design level, and therefore, it is necessary to establish the relation between feature models and interactions and the possible realizations for them at modeling level in the solution space.

7.3.2 Software product lines

Related to FOSD is the research on Software Product Lines (SPL). A SPL group a set of systems that share a common core of features. A product of a product line is specified by a valid feature selection (a subset of the features of the product line) [1]; that is, the production of a new system in a product line is done by selecting a consistent set of features [31].

The goal of SPL engineering is to capitalize on commonality and manage variability in order to reduce the effort of creating and maintaining a similar software systems (products). SPL considers the interaction problem as a building or configuration problem, where it is necessary to consider the interactions to pick the valid set of features for building a given product [1].

This above approach is, however, not suitable in our case. Even though many slot machine games can be considered as part of a product line, this is not correct. All slot machine games must have the *same set* of legally defined features. This contradicts the idea of product in the sense that each product in an SPL involves a *different set* of features.

Put differently, in our scenario there is no choice regarding which features are or are not included in a build of the system. On the contrary, all of them must always be included as most of them derive from legal requirements. In our case, features such as demo must be activated at runtime, and conflicting features must be deactivated solely when demo is working.

8 Related work

To the best of our knowledge, there are no previous publications on the application of AOM with a focus on aspect interactions in the context of industrial software. The most related work is WEAVR [11,13], which we have included in our evaluation (Sect. 5).

Wimmer et al. authored a survey of AOM approaches [35] where concern interactions are part of the evaluation framework. It shows that most of the surveyed approaches do not provide for interaction support. Of those that do, most focus on detection of syntactic and semantic interactions. For such detection, typically the UML models are transformed into graphs which are then analyzed to look for interactions. This approach is also advocated by Ciraci et al. [9] and Mehner et al. [25].

Analogous to the work in the aspect-oriented software engineering community, detection of interactions in the design phase has also been considered in the feature-oriented programming community. For example, the work of Apel et al. on FeatureAlloy [3] detects structural (syntactic) and semantic dependencies as well.

The basic assumption in all the above research is that interactions are unintended and arise at composition time. This, however, does not hold in our case. The interactions may be planned and, moreover, have already been detected during the requirements phase [36,37]. Instead of detection of interactions, we need for the design to effectively document the decisions made to manage these interactions.

Other authors purely focus on avoiding interactions. For example, Katz and Katz [20] describe how to build an interference-free aspect library. In our case, however, some interactions are required to obtain the desired behavior and other interactions cannot be removed but should be controlled instead. An example of the former is that communication protocols depend on meters, and of the latter that mutual exclusion between different protocols should be controlled.

It is interesting to note that, in our experience, the vast majority of AOM work on interactions refers to dependencies and conflicts, but neglects or minimizes reinforcement or mutex. This may indicate that these types of interactions are considered less frequent. However, they nonetheless occur in our context, so they should be treated by the AOM approaches as well. This is as we see no reason why the SM case would be exceptional in the kinds of interactions that it presents.

Lastly, in the fields of Feature-Oriented Software Development and of Software Product Lines various authors have published work that is related to our efforts, and we have analyzed this in Sect. 7.3.

9 Conclusions and future work

The AOSD-Europe technical report on interactions [33] classifies interactions in four types: dependency, conflict, mutex, and reinforcement. In the slot machine software, all four types are present, as we have established in earlier work that analyses the requirements for these machines [36]. In this text, we focus on the design phase of the development process. We evaluated the abilities of three mature AOM approaches: Theme/UML, WEAVR, and RAM to explicitly communicate these interactions in the design. For our work, it is key that these relations are explicit instead of implicit. If this is not the case, it is very likely for errors to arise in later maintenance and evolution phases. This is even more so due to the high number of feature requirements (approximately 600), which regularly change. We therefore also require the approach to provide some means of scalability, to abstract over similar patterns in the design.

The somewhat surprising result of our study is that neither Theme/UML nor WEAVR allow us to satisfactorily express any of the four types of dependency. In addition, their support for scalability is lacking, forcing us to repeat a large number of almost identical diagrams. These downsides are present even though both approaches are considered mature and accepted by the community, furthermore claiming to have support for specific kinds of interactions. In our experience, interaction support is, however, at the wrong level of granularity and scope to be useful to us. In both methodologies, the support is too fine-grained and the scope is too restricted. Considering scalability, in both approaches this has not received the in-depth attention required to provide useful scalability operators to the modeler.

The good news is that RAM fares much better than the other two approaches. Firstly, it allows us to express all four kinds of interactions at the right level of granularity and, moreover, specifying an interaction resolution strategy when needed. Secondly, its support of wildcards in the instantiation directives, combined with the flexibility of renaming, does result in an adequate level of scalability support. However, where RAM falls short is that the different interactions are not expressed in a clearly distinct fashion. Dependencies expressed in the models can be at the level of implementation as well as at the level of requirements. Moreover, all three conflict, mutex, and reinforcement interactions are denoted in the same way, losing the distinction between the three. Hence, we consider RAM an acceptable approach at best.

The key question for future work is how we would be able to satisfactorily express the interactions in our design. The most straightforward solution would be to extend one of the above methodologies such that it includes the support we are lacking and the most likely candidate is RAM. A second avenue for future work is further down the line in the development process: an evaluation of implementation languages for the slot machine. This to see how the design can be rendered into an implementation in the most straightforward fashion. In the same vein, an interesting research question is how advanced features of aspect languages, e.g., dynamic deployment, can be mapped back to features in AOM approaches. This as AOM approaches seem to ignore these language features, which arguably can serve to simplify both design and implementation.

Acknowledgments Many thanks go to Jörg Kienzle and his team at McGill for fluid conversations (via e-mail and other means) to help us with understanding RAM, receiving us at their lab, and co-modeling some diagrams of the slot machine.

References

1. Apel, S., Batory, D., Kstner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer Publishing Company, Incorporated (2013)
2. Apel, S., Kästner, C.: An overview of feature-oriented software development. J. Object Technol. **8**(5), 49–84 (2009)
3. Apel, S., Scholz, W., Lengauer, C., Kästner, C.: Detecting dependences and interactions in feature-oriented design. In ISSRE, pp. 161–170. IEEE Computer Society (2010)
4. Apel, S., von Rhein, A., Thüm, T., Kästner, C.: Feature-interaction detection based on feature-based specifications. Comput. Netw. **57**(12), 2399–2409 (2013)
5. Ayed, A., Kienzle, J.: Integrating protocol modelling into reusable aspect models. In: Proceedings of the 5th ACM SIGCHI Annual International Workshop on Behaviour Modelling—Foundations and Applications, BMFA'13, pp. 2:1–2:12. ACM, New York, NY, USA (2013)
6. Calder, M., Kolberg, M., Magill, E. H., Marples, D., Reiff-Marganiec, S.: Hybrid solutions to the feature interaction problem. In: Amyot, D., Logrippo, L. (eds.) FIW, pp. 295–312. IOS Press (2003)
7. Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S.: Feature interaction: a critical review and considered forecast. Comput. Netw. **41**(1), 115–141 (2003)
8. Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Alarcon, M., Bakker, J., Tekinerdogan, B., Clarke, S., Jackson, A.: Survey of analysis and design approaches. Technical Report AOSD-Europe Deliverable D11, AOSD-Europe-ULANC-9, University of Lancaster (2005)

9. Ciraci, S., Havinga, W., Aksit, M., Bockisch, C., van den Broek, P.: A graph-based aspect interference detection approach for uml-based aspect-oriented models. Trans. Asp. Oriented Softw. Dev. **7**, 321–374 (2010)
10. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design. The Theme Approach. Object Technology Series. Addison-Wesley, Boston (2005)
11. Cottenier, T., Berg, A.V., Elrad, T.: The Motorola WEAVR: model weaving in a large industrial context. In: Proceedings of the International Conference on Aspect Oriented Software Development, Industry Track (2006)
12. Cottenier, T., van den Berg, A., Elrad, T.: Stateful aspect: The case for aspect-oriented modeling. In: Proceedings of the 10th International Workshop on Aspect-Oriented Modeling. ACM (2006)
13. Cottenier, T., van den Berg, A., Elrad, T.: Motorola weavr: aspect and model-driven engineering. J. Object Technol. **6**(7), 51–88 (2007)
14. Fabry, J., Zambrano, A., Gordillo, S.: Expressing aspectual interactions in design: experiences in the slot machine domain. In: Whittle, J., Clark, T., Kühne, T. (eds.) Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science, vol. 6981, pp. 93–107. Springer, Berlin/Heidelberg (2011)
15. Gaming Laboratories International. Gaming Devices in Casinos (2007). http://www.gaminglabs.com/
16. Gaming Standard Association. Game to Server (G2S) Protocol Specification (2008). http://www.gamingstandards.com/
17. Homayoon, S., Singh, H.: Methods of addressing the interactions of intelligent network services with embedded switch services. IEEE Commun. Mag. **26**(12), 42–46, 70 (1988)
18. IBM: Rational TAU (2014)
19. Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: Form: a feature-oriented reuse method with domain-specific reference architectures. Ann. Softw. Eng. **5**, 143–168 (1998)
20. Katz, E., Katz, S.: Incremental analysis of interference among aspects. In Clifton, C. (ed.) FOAL, pp. 29–38. ACM (2008)
21. Kellens, A., Mens, K., Brichau, J., Gybels, K.: Managing the evolution of aspect-oriented software with model-based pointcuts. In: European Conference on Object-Oriented Programming (ECOOP), number 4067 in LNCS, pp. 501–525 (2006)
22. Kienzle, J., Abed, W.A., Klein, J.: Aspect-oriented multi-view modeling. In: Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development. AOSD '09, pp. 87–98. ACM, NY, USA (2009)
23. Kienzle, J., Abed, W.A., Fleurey, F., Jazcquel, J.-M., Klein, J.: Aspect-oriented design with reusable aspect models. In: Katz, S., Mezini, M., Kienzle, J. (eds.) Transactions on Aspect-Oriented Software Development VII. Lecture Notes in Computer Science, vol. 6210, pp. 272–320. Springer, Berlin, Heidelberg (2010)
24. Liu, J: Feature interactions and software derivatives. J. Object Technol. **4**(3), 13–19 (2005). GPCE Young Researchers Workshop 2004
25. Mehner, K., Monga, M., Taentzer, G.: Interaction analysis in aspect-oriented models. In: Requirements Engineering, 14th IEEE International Conference, pp. 66–75. IEEE Computer Society (2006)
26. Metzger, A., Bühne, S., Lauenroth, K., Pohl, K.: Considering feature interactions in product lines: towards the automatic derivation of dependencies between product variants. In: Reiff-Marganiec, S., Ryan, M. (eds.) FIW, pp 198–216. IOS Press (2005)
27. Metzger, A., Bühne, S., Lauenroth, K., Pohl, K.: Considering feature interactions in product lines: towards the automatic derivation of dependencies between product variants. In: Reiff-Marganiec, S., Ryan, M. (eds) Proceedings of the 8th International Conference on Feature Interactions in Telecommunications and Software Systems, ICFI'05 (Leicester, UK, June 28–30, 2005), Amsterdam . IOS Press (June 2005)
28. Nakamura, M., Reiff-Marganiec, S, (eds.): Feature Interactions in Software and Communication Systems X, International Conference on Feature Interactions in Software and Communication Systems, ICFI 2009, 11–12 June, 2009, Lisbon, Portugal. IOS Press (2009)
29. Nevada Gaming Commission: Technical Standards For Gaming Devices And On-Line Slot Systems (2008). http://gaming.nv.gov/stats_regs.htm
30. Nhlabatsi, A., Laney, R., Nuseibeh, B.: Feature interaction: the security threat from within software systems. Prog. Inform. **5**, 75–89 (2008)
31. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations. Principles and Techniques. Springer, Secaucus (2005)
32. Prehofer, C.: Feature-oriented programming: a fresh look at objects. In: ECOOP, pp. 419–443 (1997)
33. Sanen, F., Truyen, E., Win, B.D., Joosen, W., Loughran, N., Coulson, G., Rashid, A., Nedos, A., Jackson, A., Clarke, S.: Study on interaction issues. Technical Report AOSD-Europe Deliverable D44, AOSD-Europe-KUL-7, Katholieke Universiteit Leuven (2006)
34. Silva Filho, R.S., Redmiles, D.F.: Managing feature interaction by documenting and enforcing dependencies in software product lines. In: 9th International Conference on Feature Interactions in Software and Communication Systems (ICFI'07), pp. 33–48, Grenoble, France, Sept 3–5 (2007)
35. Wimmer, M., Schauerhuber, A., Kappel, G., Retschitzegger, W., Schwinger, W., Kapsammer, E.: A survey on UML-based aspect-oriented design modeling. ACM Comput. Surv. **43**(4), 28:1–28:33 (2011)
36. Zambrano, A., Fabry, J., Gordillo, S.: Expressing aspectual interactions in requirements engineering: experiences, problems and solutions. Sci. Comput. Program. **78**(1), 65–92 (2012)
37. Zambrano, A., Fabry, J., Jacobson, G., Gordillo, S.: Expressing aspectual interactions in requirements engineering: experiences in the slot machine domain. In: Proceedings of the 2010 ACM Symposium on Applied Computing (SAC 2010), pp. 2161–2168. ACM Press (2010)
38. Zhang, J., Cottenier, T., van den Berg, A., Gray, J.: Aspect interference and composition in the motorola aspect-oriented modeling weaver. In: Workshop on Aspect-Oriented Modeling at the 9th International Conference on Model Driven Engineering Languages and Systems (2006)
39. Zhang, W., Mei, H., Zhao, H.: A feature-oriented approach to modeling requirements dependencies. In: 2013 21st IEEE International Requirements Engineering Conference (RE), vol. 0, pp. 273–284 (2005)
40. Z. 100: ITU. Specification and description language (SDL). In International Telecommunication Union (2000).

Johan Fabry obtained his Ph.D. in Computer Science from Vrije Universiteit Brussel. He is professor at the Computer Science Department (DCC) of the University of Chile. His research interests include Aspect-Oriented Programming, dependencies and interactions with aspects, Domain-Specific Aspect Languages, and the use of Domain-Specific Languages and advanced language features to enhance programmer productivity.

Arturo Zambrano holds a Master in Software Engineering and a Ph.D. in Computer Science from Universidad Nacional de La Plata. He is professor at Universidad Nacional de Quilmes, Argentina. His experience in the industry includes 7 years developing software for US gambling market and 5 years leading the development of software for digital tv, streaming and related technologies. His research interest lies on advanced modularization techniques.

Silvia Gordillo received a Ph.D. diploma from INSA-Lyon, France. She has a M.Sc. in Software Engineering diploma from La Plata University, Argentina. She is the head of the Mobile Application Systems project at the Laboratory for Research and Training in Advanced Information Systems (LIFIA) of La Plata University. She is professor at the School of Computer Science of the University of La Plata, both in undergraduate and graduate courses related with Mobile Applications.