Improving Robustness of Evolving Exceptional Behaviour in Executable Models

Nelio Cacho, Thomas Cottenier, Alessandro Garcia

Computing Department, Lancaster University, United Kingdom
{n.cacho, t.cottenier, garciaa}@lancaster.ac.uk

ABSTRACT

Executable models are increasingly being employed by development teams to implement robust software systems. Modern executable models offer powerful composition mechanisms that allow developers to deliver a running system in small increments and in a time-effective fashion. Such models act like code by providing high-level development abstractions and, as a consequence, it is expected that increased software robustness is achieved. However, existing executable models have a number of limitations on the representation of exceptional behaviour. Similarly to exception handling in programming languages, one of the key problems is that the modelling languages and supporting environments do not allow the explicit specification of global exception flows. They require that developers understand the source of an exception, the place where it is handled, and everything in between. As system development evolves, exceptional control flows become less well-understood, with negative consequences for the program maintainability and robustness. In this paper, we claim that such problem can be addressed by an innovative exception handling model which provides abstractions to explicitly describe global views of exceptional control flows. The implementation of our proposed model extends the aspect-oriented language constructs and the control-flow analysis of the Motorola WEAVR with the aim of promoting enhanced robustness and program modularization.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—Modules and interfaces;

General Terms

Design, Reliability.

Keywords

Exception handling, model-driven software development, aspect-oriented programming, modularity, exception control flow.

1. INTRODUCTION

Model-Driven Software Development (MDSD)[13,14] techniques are increasingly used in large development environments where different team members work on distinct executable models [14]. In order to improve software robustness, modelling languages support different levels of abstractions targeted at specific software development stages. They typically allow to: (i) define and visualise the system architecture, (ii) specify the normal and exceptional interfaces between different sets of behaviours, and (iii) detail normal and exceptional behaviours of the system in a platform-independent manner. During the software architecture definition phases, the system behaviour is decomposed according to modules that encapsulate cohesive pieces of behaviour. The decomposition goal is to minimize the inter-module dependencies and, therefore, improve software maintainability.

WEH '08, November 14, Atlanta, Georgia, USA

However, MDSD languages do not provide explicit abstractions to capture the global effects of exception control flows traversing the system architecture. Exceptional behaviour has the tendency to leak out of the module interfaces established during the architectural definition phases [17]. This means that the handling of an exception raised in one module generally requires corrective actions to be taken in a number of other modules. Hence, it is fundamental to have the ability to *explicitly* declare the dependencies between modules introduced by global exceptions. The explicit exception flows are essential to guarantee that faults are not introduced in evolving executable models.

Although MDSD languages, such as UML, support basic exception handling primitives to specify exception interfaces, they do not provide explicit abstractions to specify and reason about exceptional control flows. They do not support modular descriptions of modules that are the source and target of exceptions. These descriptions are essential to ensure that minor modifications in the implementation of a module do not break the expected effects of global exceptional flows. In other words, existing MDSD languages mimic the limitations of contemporary exception handling mechanisms in programming languages [3].

In this context, the contributions of this paper are threefold. First, we discuss the liabilities of modelling exceptional behaviour in MDSD. Second, we propose an exception handling model which provides platform-independent representation of exceptional behaviour. The proposed model aims to make exceptional behaviour more resilient in the presence of changes. Third, we present the concretization of this model by leveraging aspect-oriented language constructs and the control-flow analysis of the WEAVR [6]. WEAVR supports an aspect-oriented extension to UML 2.0. This implementation exploits existing pointcut languages to explicitly capture the semantics of exception control flow that spawn over multiple modules of the system.

The next section introduces some of the notations used in a MDSD environment and discusses the impact of exceptional behaviour on the models of the system. Section 3 introduces the *EFlow* model and details how it relates to the UML. Section 4 discusses the implementation of the EFlow model in terms of successive model transformations. Finally, Section 5 discusses related work and Section 6 concludes this paper.

2. MODEL DRIVEN SOFTWARE DEVELOPMENT

Model-Driven Software Development (MDSD) takes an iterative, top-down approach to software engineering where different models of the system are the central artefacts of the development. The Model-Driven Architecture (MDA) therefore proposes a standard architecture for MDSD based on the UML. The MDA defines two main levels of system abstraction: the Platform Independent Model (PIM) and the Platform Specific Model (PSM).

The PIM is a model of the system structure and behaviour, and is typically developed by system architects. It defines the functionality and behaviour of a system independently of the specific technologies that implement it. The PIM can automatically or semi-automatically be mapped to different Platform-specific Models (PSM) of the system. In this work, we consider that to achieve fully automated code generation, a translationist [14] approach is the most adequate one which is able to support precise modelling of software behaviour.

Executable models need to be manually refined to a level of granularity that enables tools to fully generate code for a target platform given platform specific generation rules. The platform-specific models and the code are fully automatically generated. Next section details how a translationist approach can be used to define executable models.

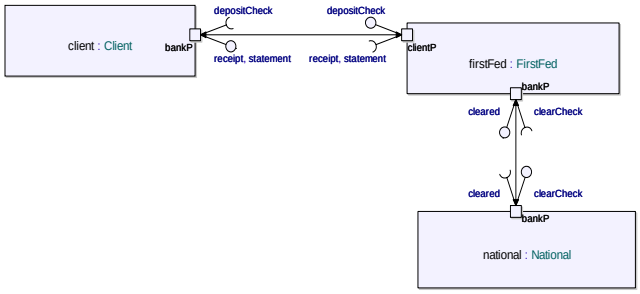

Figure 1. Composite-structure architecture diagram for a Bank application

2.1 Executable Models

The translationist interpretation of the MDA emphasizes model executability. The precise behaviour of the model is defined imperatively by embedding actions within statechart [9] transitions. Actions are executed during a transition from one state to another and as the PIM specifies the complete behaviour of the application, there is no need to manually elaborate the PSM and the generated code. The mappings between models and code generation can therefore include various platform-specific optimizations. Thus the PSM and the generated code are not appropriate for human inspection, and are hidden from the developer. The obvious advantage of the translationist approach is that PIM's can be tested early in the lifecycle. This style of modelling is also in line with agile development methods.

In order to obtain an executable model, the detailed behaviour of components must be specified. First, the structure of a model can be obtained by modelling the static parts of a system, representing elements that are either conceptual or physical [2]. For instance, Figure 1 gives a high level view of the composite-structure diagram which describes some components of a Bank application. Each of those components may be itself composed of multiple active classes. Then, the observable behaviour of each active class is specified using statechart diagrams [9]. For the sake of illustration, Figure 2 depicts the normal behaviour of two operations: *doDeposit* and *clearCheck*. According to the *doDeposit's* statechart, this operation initiates in the *Active* state. Upon receiving a *deposit* invocation, this statchart invokes the *depositCheck* operation of *FirstFed* component. Then the state is transformed into *Processing*. At this state, the condition *id!=null* is tested at each instant of time to identify when the return value is available. When this occurs, the state is transformed into *Clearing*, and it waits for the availability of *idStmt* to print the statement and finalize the operation.

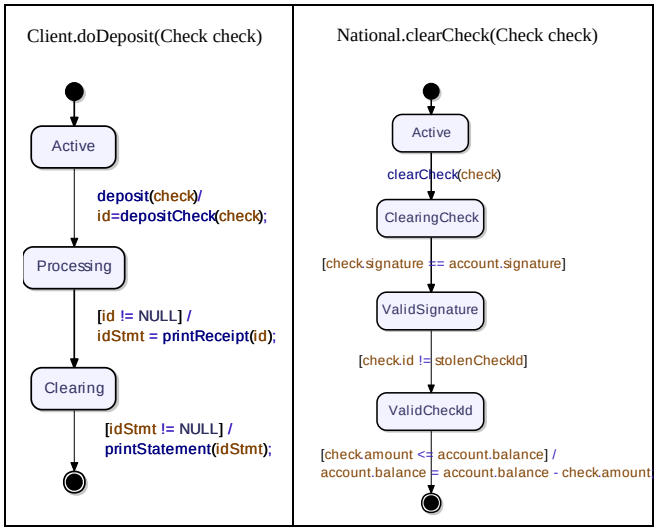

Figure 2. Representation of normal behavior using statecharts.

Note that those statecharts represent only the states and transitions in which all operations are successfully executed. Next section describes some problems that are usually faced when trying to modelling exceptional behaviour.

2.2 Modelling Exceptional Behaviour

Since the system behaviour is represented by a sequence of transitions, a correct behaviour is provided whenever the system implements the *agreeable specification* [1]. Such specification defines the correct sequence of valid transitions to achieve one specific goal. In contrary, misbehaviour might be provided when at least one or more transitions deviate from the correct sequence.

The deviation from the correct transitions represents a deviation from the *normal control flow* to the *exceptional control flow*, and it must be treated as an *exception* by some special processing. Otherwise such a violation can eventually lead the system to a *failure* [1]. For instance, Figure 3 depicts some new states and transitions introduced into *clearCheck* operation in order to support exceptional behaviour.

Ideally, this behaviour should be fully supported by UML 2.0 specification. However, such a specification allows only identifying exceptional flow of events in use case diagrams, representing them as normal signals, and finally attaching these signals to method's specification [2]. However, no reference is made on how to enforce that exceptions propagated by methods described by means of statecharts are going to be handled. It is up

to the designer to reason about the exceptional control flow and ensure that no exception is going to lead the system to a failure.

Exceptions often need to be handled in a global fashion [17,18]. Therefore, as pointed out by many authors [4,12,17,18], reasoning about exceptional control flow is not an easy task. In executable models, this problem can be even worse since exceptional behaviour may be inconsistently represented throughout many diagrams. Furthermore, such representations are based on the implicit assumption that it is enough to specify the places where a program raises exceptions (*raising sites*) and the places where it handles them (*handling sites*). There is no explicit relation between raising and handling sites.

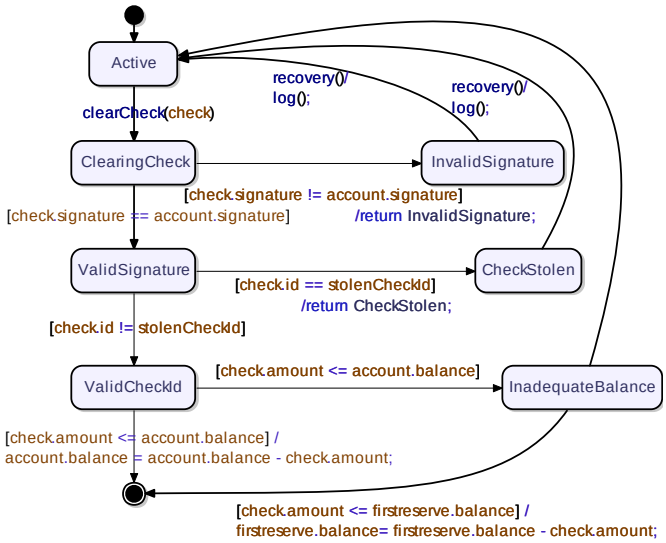

Figure 3. Complexity of exceptional behaviour

For instance, Figure 2 depicts two executable descriptions of methods *doDeposit* and *clearCheck*. Let's suppose that method *clearCheck* evolves to the description illustrated in Figure 3. This later description deals with three new exceptional situations and propagates some of them to its caller. In this scenario, as there is no relation between the exceptional behaviour of both methods, there is no guarantee that the new exceptional situations created by *clearCheck* are going to be handled by *doDeposit* method. This in turn leads to the development of executable models that are less robust and maintainable.

In addition, the decision to leave to the method level the responsibility for the definition of exceptional behaviour tends to affect the software evolvability. This occurs because the raising and handling sites are spread out over the different methods of the systems, making the exception handling behaviour tightly coupled with the normal behaviour. The problem arises when requirements with respect to the exception handling behaviour change, the implementation of the system would need to be manually modified. As the implementation of this behaviour is not localized in a separate module, the impact of changing this requirement on the implementation would be massive.

In summary, we have observed that current profiles for executable models do not provide proper support for the task of understanding and ensuring the paths the exceptions take from the raising site to the handling site. The main consequence of this limitation is that if a programmer changes exception-related state transitions, the control flow in apparently unrelated parts of the program may change in surprising ways. This creates two direct complications. First, it becomes difficult to discover where the exceptions raised within a given context will be handled since there is no distinction between normal and erroneous states. It is also difficult to trace a handled exception to the place where it was originally raised.

3. MODEL-DRIVEN EXCEPTION HANDLING MECHANISM

Addressing the problems described above requires finding techniques to fulfil two requirements. First, it is required to decompose the exceptional behaviour into separate modules that encapsulate the different exception handling strategies. Second, these modules need to expose well-defined interfaces that non-ambiguously describe their behaviour.

The fulfilment of these two requirements will make it possible to understand exception flows from an end-to-end perspective by looking at a single part of the executable model. They are also essential to allow executable models to evolve in a robust manner and accommodate changes to their requirements.

In the following, Section 3.1 describes an exception handling model that fulfils such requirements. Section 3.2 describes how this model can be concretized by means of a model-based composite mechanism.

3.1 EFlow: Exceptions at Executable Models

The main aim of an exception handling model is to define the interaction between raising sites and the handling sites. Eventually, exception handling model should provide abstractions to enforce that no exception could inadvertently lead the system to a failure. In this context, we introduce *EFlow* model, whose major goal is to make exception flow explicit, safe, and understandable by means of e*xplicit exception channels* and *pluggable handlers.* EFlow is a more abstract definition of the *EJFlow* mechanism defined elsewhere [3]. This section leverages platform-independent language elements (statecharts [9, 10]) to describe the concepts that underpin EFlow model.

In the following, we assume that a given statechart of a component *A* is a tuple *(S, Act, V, T)* where S, Act, V and T are sets of states, actions, variables and transitions, respectively. Let $S = \text{N} \cup E$ where *N* is a finite set of *valid* states, *E* a finite set of *erroneous* states, and $\text{N} \cap E = \emptyset$. A transition of a statechart is a tuple (s, a, s'), where $s, s' \in S$ are the source state and target state, respectively. For convenience, we write $s \xrightarrow{a} s'$ instead of (s, a, s'). The action $a \in Act$ can modify values or define predicate on variables in *V*, perform method invocations, produce output messages, and so forth.

3.1.1 *Explicit Exception Channels*

An *explicit exception channel* (*channel*, for short) is an abstract duct through which exceptions flow from a raising site to a handling site. More precisely, an explicit exception channel *EEC* is a 4-tuple consisting of:

- *Rs*: a set of raising sites;
- *Hs*: a set of handling sites;
- *Is:* a set of intermediate sites;
- *Ds:* a set of declaration sites;

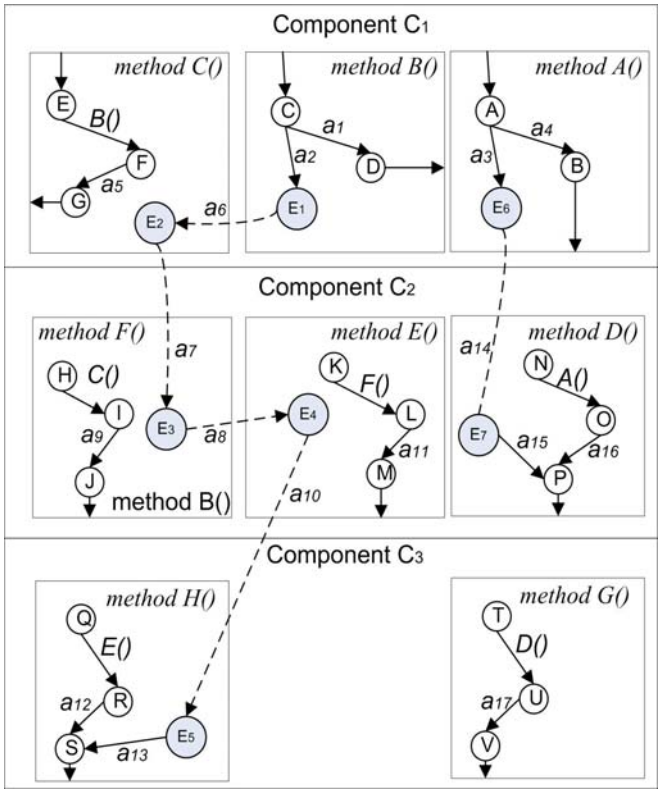

Figure 4. Exception control flow involving multiple methods

Raising sites (*Rs*) are transitions $s \xrightarrow{a} e$ where $RS = \{(s, a, e): s \in S,\ a \in Act, e \in E\}$. This corresponds to all transitions in the application whenever the occurrence of a leads the system to switch to an erroneous state. A raising site is depicted in Figure 4 by the transition $c \xrightarrow{a2} e1$. This figure depicts part of the architecture of a software system, comprising three components: *C1*, *C2*, and *C3*. Component *C3* requires services from component C2 which, in turn, request services from component *C1*. These components are defined in terms of methods, each method denotes statecharts where circles symbolize states, and arrows express transitions. In this context, the occurrence of transition *a2* transforms the state of method *B* into an erroneous state *E1*. In practice, this erroneous state represents a deviation from the normal return (provided by state *D*) to an exceptional return (denoted by the dashed arrow). A dashed arrow indicates that, during the execution of *B*, an exception can be raised and this exception will be signalled to *C*. As would be expected, each dashed arrow also indicates that control flow is passed from one method to the other.

Handling sites (*Hs*) are transitions $e \xrightarrow{a} n$, where. $HS = \{(e, a, n): e \in E,\ a \in Act, n \in N\}$. Handling sites denote all transitions whenever an action a takes place, and in turn switches the system to a valid state. In Figure 4, for example, a handling site is described by the transition $e5 \xrightarrow{a13} s$.

As depicted in Figure 4, raising and handling sites are the two ends of an explicit exception channel. All transitions that are neither handling nor raising sites are considered *intermediate sites* (*Is*). Thus, intermediate sites are transitions $e \xrightarrow{a} e'$, where $Is = \{(e, a, e'): e, e' \in E, a \in Act\}$. Accordingly, intermediate sites comprise all transitions through which erroneous states are successively transformed into other erroneous states from the raising site on its way to the handling site. For instance, in Figure 4, transitions $e1 \xrightarrow{a6} e2$ and $e2 \xrightarrow{a7} e3$ are examples of intermediate sites.

The main purpose of defining raising, handling and intermediate sites is to reveal the implicit exception control flow created by a raising site. For instance, the implicit exception control flow created by the raising site $c \xrightarrow{a2} e1$ can be revealed by means of the following explicit exception channel:

$$Eec1 = \{\left(c \xrightarrow{a2} e1\right), \left(e5 \xrightarrow{a13} s\right), \left(e1 \xrightarrow{a6} e2 \xrightarrow{a7} e3 \xrightarrow{a8} e4 \xrightarrow{a10} e5\right), \emptyset\}$$

As described in Figure 4, the explicit exception channel *Ecc1* percolates through different methods and components of a given system. This characteristic allows (i) to define the two ends of the explicit exception channel and (ii) to check whether all exception control flows created by a raising site are properly handled at handling sites. Clearly, an explicit exception channel does not need to necessarily be limited to any modular unit boundaries. However, it is widely believed that the definition of certain boundaries helps to provide a clear specification and a better understanding of exception handling behaviour [12,17]. Moreover, some modular units, such as components are well known for being independently deployable and composable with third party modules [20]. For these reasons, developers need to take into account the module boundaries when defining the explicit exception channel.

In order to define explicit exception channels which respect module boundaries, declaration sites can be used to slice the explicit exception channel according to each module. For example, the explicit exception channel *Eec1* can now be described in terms of three channels:

$$Eec1 = \{\left(c \xrightarrow{a2} e1\right), \emptyset, \left(e1 \xrightarrow{a6} e2\right), (e2)\}$$

$$Eec2 = \{\left(e2 \xrightarrow{a7} e3\right), \emptyset, \left(e3 \xrightarrow{a8} e4\right), (e4)\}$$

$$Eec3 = \{\left(e4 \xrightarrow{a10} e5\right), \left(e5 \xrightarrow{a13} s\right), \emptyset, \emptyset\}$$

In the first two channels (*Eec1* and *Eec2*), the handling site is not defined whereas the declaration site specifies the last state before the error reaches the component boundary (*e2* and *e4*). The definition of a declaration site means that the channel is not able to deal with all exception control flows and in turn other channel should extend it beyond the module boundaries. In this context, the third channel *Eec3* extends the second by defining its raising site with a reference to the declaration site of channel *Eec2*. The utilization of declaration sites makes it possible to (i) define explicit exception channels of entire systems from extended definitions of individual components, and to (ii) enforce robustness between raising and handling site since whenever a handling site is not available, a declaration site should be provided to make the channel valid.

3.1.2 *Pluggable Handlers*

The EFlow model ensures that all exceptions generated at the raising sites are going to be handled at the handling sites. In this context, the handling site of an explicit exception channel denotes the set of valid states in which the exception handling model should leave the system after the occurrence of an exception. In

order to generate such valid states, the EFlow model introduces the concept of *pluggable handler*.

A *pluggable handler* is one or more corrective actions that can be associated to arbitrary transitions. Let *Ca* be a provided set of pluggable handlers. For s ∈ *S*, a pluggable handler is defined as $PlugH(s) = \left\{ca \in Ca: \exists s' \in E, s' \xrightarrow{ca} s\right\}$.

Note that pluggable handlers belong to a different set of actions *Ca* which in practice denotes the separation between error handling behaviour from normal behaviour. A single pluggable handler can be associated with either intermediary or handling sites. Pluggable handlers whose corrective actions lead the system to a valid state are attached to handling sites. In contrast, actions that alleviate the problem but do not switch the system to a valid state are associated with intermediary sites.

3.2 Making the Model Concrete

As described in the previous section, EFlow relies on detailed specification of raising sites, intermediate sites, and handling sites. Pluggable handlers are defined to enforce the definition of safe exceptional control flow between modules. The main benefit of such an explicit specification is that exceptional behaviour becomes more resilient in the presence of changes. In addition, both system's design and architecture can be validated early on in the lifecycle.

However, the applicability of EFlow depends on the capabilities of the chosen technique for concretising the proposed model. For instance, it may be infeasible to give exhaustive specifications of raising sites, intermediate sites, and handling sites for each exceptional control flow since the number of possible state transitions can be extremely large in any medium-size application. Additionally, the composition of pluggable handler may bring new problems of its own. For instance, pluggable handler may incorrectly change the control flow of applications once such handlers tend to introduce new states and new decision actions in multiple state machines to handle erroneous situations.

Given these facts, the chosen technique should be able (i) to describe at high level the notion of explicit exception channels, and (ii) support composition mechanisms that do not require intrusive modifications of the normal behaviour. We believe that Aspect-oriented [11] design mechanisms can fulfil such requirements. These mechanisms can be characterized by their ability to non-invasively introduce behavior into applications and their ability to quantify over the system elements where this behavior should be introduced. In this context, Section 3.2.1 outlines an aspect-oriented modelling language and Section 3.2.2 describes how such modelling language was extended to support the EFlow abstractions.

3.2.1 Weaving Handlers into Models with WEAVR

The Motorola WEAVR [6, 7] provides language constructs to capture aspects in UML 2.0 and perform weaving of state machines before code generation. WEAVR supports two distinct types of pointcut descriptors: *action pointcuts* and *transition pointcuts*. The pointcut descriptors strictly refer to actions and transitions declared in the statechart specification of the system.

The notation used for both types of pointcuts is identical: a pointcut is always represented as a transition ($s \xrightarrow{a} t$) from a set of source states (*s*) to a set of target states (*t*), triggered by an action (*a*) expression. Wildcards can be used to quantify over both the source and target states of the transition. The action expressions are used to match the signatures of transition triggers and the signatures of actions executed in the context of a transition.

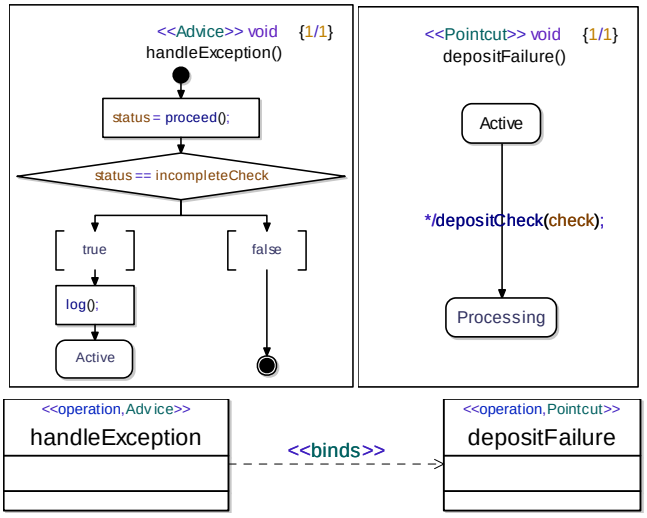

Figure 5. Example of aspect binding diagram, the *depositFailure* pointcut and the *handleException* advice.

For instance, Figure 5 depicts an aspect in WEAVR which handles the exception *incomplete_check* propagated by method *depositCheck*(). This aspect defines one pointcut *depositFailure* and one advice *handleException*. In WEAVR, advices are bound to the pointcut through a dependency that is annotated with the *binds* stereotype. Advices are instantiated for each joinpoint that matches a pointcut descriptor. In Figure 5, the *depositFailure* pointcut is one example of action pointcut. The *depositFailure* pointcut selects remote procedures call or simple method invocations that match the *depositCheck* method of *FirstFed* component. The *handleException* advice executes *depositCheck* by means of using *proceed()* statement, and upon returning, the status variable is tested to determine if *incomplete_check* exception was signalled. If so, the *handleException* advice logs this exception and returns to the *Active* state.

As described above, the Motorola WEAVR allows composing handlers with the control flow of the state machine at multiple

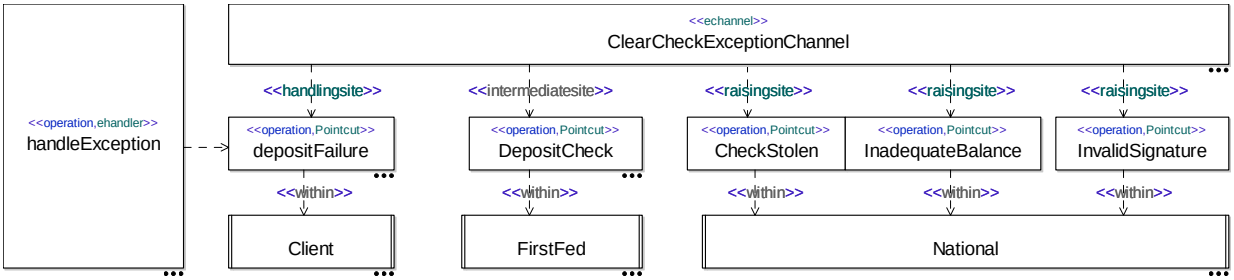

Figure 6. Example of Explicit Exception Channel.

locations. However, WEAVR does not provide any mechanism to support the enforcement of the exceptional control flow from the raising site to the handling site (Section 3.1.1). In other words, if a designer accidentally changes the return from *incomplete_check* to *incompletecheck*, the control flow in *handleException* advice will not be the same since this slightly different value will not be recognized as an exception, and in turn the handler will not properly handle this exception.

Again, even for composition mechanisms such as AO languages, the exceptional behaviour is defined by means of the places where a program raises exceptions and the places where it handles them. No relation is established to enforce that changes in the raising sites may compromise the effectiveness of handlers defined in the handling sites. This lack of explicit dependency reveals that AO mechanisms alone are not enough to provide safe composition of exceptional behaviour with normal behaviour. In this context, the next section describes a set of abstractions which were included in WEAVR to support the concept of explicit exception channel.

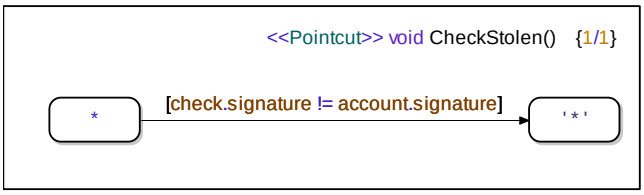

Figure 7. Raising site defined in terms of a transition pointcut

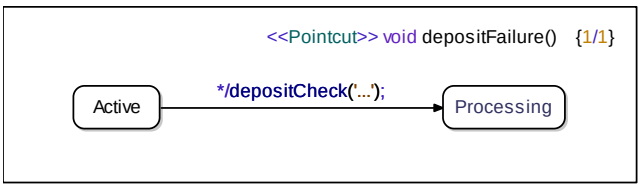

Figure 8. Handling site defined in terms of a transition pointcut

3.2.2 *Exception Handling Modelling Language*

To support the definition of explicit exception channels, WEAVR was extended in two directions: an UML profile and an enforcement mechanism. First, we define a UML profile as an extension of the UML standard language with specific elements to support the definition of explicit exception channels. Such new notational elements include two new types of entities, *echannel* and *ehandler*.

An explicit exception channel is visually represented as a class that is annotated by the *echannel* stereotype. The profile also defines three new dependency stereotypes *raisingsite*, *handlingsite*, and *intermediatesites* which are used to determine the raising sites, handling sites and intermediate sites of explicit exception channels, respectively.

An operation annotated with *ehandler* stereotype is an implementation of a pluggable handler. This operation encapsulates the exception handling behaviour that is executed when a certain point in an explicit exception channel is reached. An *ehandler* operation is bound to the handling site of an explicit exception channel.

For instance, Figure 6 depicts an explicit exception channel *ClearCheckExceptionChannel* that represents one of the exceptional control flows described in Figure 3 and 4. This channel is bound to five pointcuts through dependencies that are annotated with the *raisingsite*, *intermediatesite* and *handlingsite* stereotypes.

The pointcuts annotated with *raisingsite* represent the source of the channel, i.e. the exceptional situations that originate the channel. For instance, Figure 7 describes that an instance of *ClearCheckExceptionChannel* will be created whenever the check signature is not valid.

The *ClearCheckExceptionChannel* channel is further constrained by the *DepositCheck* intermediary site. The channel only captures the *CheckStolen*, *InadequateBalance* and *InvalidSignature* exceptions that flow through the *DepositCheck* method in the *FirstFed* class.

The *handleException ehandler* handles exceptions flowing from the *raisingsite* locations, through the *intermediatesite* locations, at the locations defined by the *handlingsite* pointcuts. Specifically, the *handleException ehandler* is activated when such exceptions reach a transition from state *Active* to state *Processing*, along which the *depositCheck* method is called, within class *Client*, as illustrated in Figure 8.

In addition to model the exception control flow, our approach checks whether all exceptions defined by means of raising sites are handled by a pluggable handler at a raising site. In particular, if *CheckStolen*, *InadequateBalance* or *InvalidSignature exceptions* are not in the control flow of the *depositFailure* pointcut, then they will not be handled by the *handleException ehandler*. The semantics of explicit exception channels enforce that such case are detected by the model checker, through control flow analysis.

4. MODEL TRANSFORMATION

In order to generate executable code, models based on EFlow abstractions go through three transformation processes. First, the EFlow model is transformed into a set of WEAVR aspects. Second, these aspects are composed with the base model through WEAVR model transformations. Finally, the woven models are used to generate platform-specific executables.

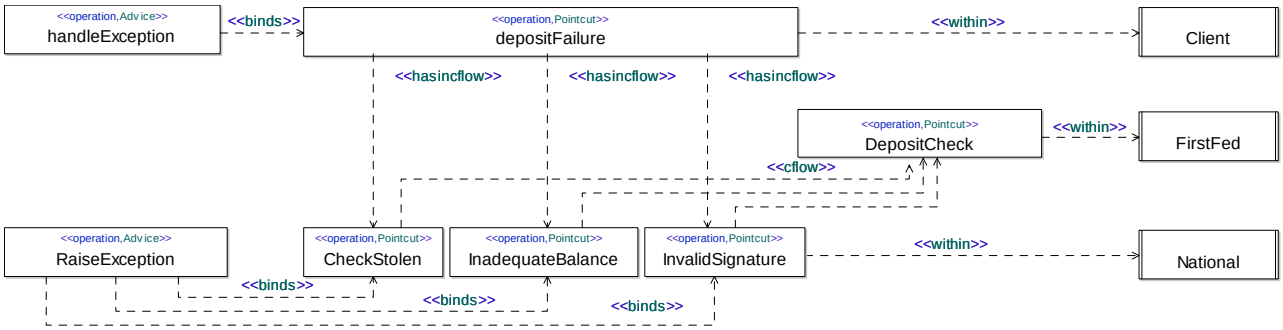

Figure 9. Explicit exception channel translated into WEAVR aspect model

4.1. First Transformation: Translation of EFlow model to WEAVR Aspect Model

EFlow provides explicit abstractions to capture exceptional control flows. These abstractions capture control flow relationships between locations in the models. The semantics of the EFlow abstractions can be translated into regular WEAVR aspects, which explicitly declare these control flow relationships. This is achieved by introducing join point composition operators: between the pointcuts defined in the EFlow model: *cflow*, *isincflow* and *hasincflow*.

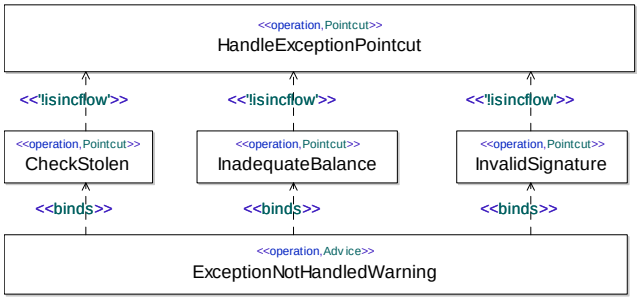

Figure 10. Aspect responsible for checking *echannel* semantics.

cflow is a standard aspect-oriented programming construct. The following pointcut matches all calls to method *c* (see Figure 4) that occur in the control flow of a call to method *b*.

pointcut foobar : call(* *.c(…)) && cflow(call(* *.b(…))

cflow is a dynamic operator. In the general case, it can not be determined at compile time whether a call to *b* occurs in the control flow of a call to method *c*.

As a consequence, we extended WEAVR with a static approximation to *cflow* called *isincflow*. *isincflow* would match all calls to method *c* for which it can be statically determined that the call occurs in the control flow of a call to method *b* through static control flow analysis. However, exceptional control flow is the opposite of normal control flow. Hence, we introduce the *hasincflow* operator, which captures the opposite semantics of *isincflow*. call(* *.c(…)) && hasincflow(call(* *.b(…)) matches all calls to method *c* that may have a call to *b* in their control flow.

Given these join point composition operators, the *EFlow* model of Figure 6 can be translated into the *WEAVR* aspect model of Figure 9. The *RaiseException* advice throws an exception at the locations defined by the *raisingsite* pointcuts, within class *National*, when they are in the control flow of the *DepositCheck* intermediate site pointcut. The *handleException* advice corresponds to the *ehandler*. It introduces error handling code at the locations defined by the handling site pointcuts that may have one of the raising site pointcuts in their static control flow.

Finally, the *EFlow* checking semantics are implemented by the aspect of Figure 10. An exception not handled warning is declared at weaving time whenever one of the raising sites is not in the static control flow of the *HandleExceptionPointcut*, which captures the error handling code introduced by the *handleException ehandler*.

4.2. Second Transformation: Weaving of Aspects

After translation from EFlow models to WEAVR aspects, the aspects are integrated with the base model through the WEAVR model transformations. These transformations take as input the base model and the aspect definitions and generate a new model of the application. The WEAVR engine performs reachability, data flow and control flow analysis over the state machines of the model and introduces the behaviour defined by the aspect advice at the locations specified by the pointcuts, according to their control flow dependencies. The generated model is UML 2.0 compliant model that captures the semantics of the base entities and the aspects.

4.3. Third Transformation: Code Generation

Finally, the woven model is translated into a platform-specific executables according to a set of platform transformation rules. These rules map a generic exception handling mechanism used at the model level to the specific throw/try/catch constructs used in the target programming language. For example, if the target platform runs *C* code, exception handling mechanisms would be mapped to the *C setjmp*/*longjmp* primitives. Hence, the characterization of exception handling behaviour at the modelling-level facilitates the control flow analysis because we can abstract from the details and specificities of the target language used.

5. RELATED WORK

There are a number of approaches that try to systematically integrate exceptional behaviour into early phases of software development process. Shui et al. [19] proposes an approach to reveal the presence of exceptional behaviour by means of the definition of exceptional use cases. This approach guides developers to describe possible exceptional situations that may affect system reliability and safety. Subsequently, Mustafiz et al. [15] proposes a model-driven approach whose main goal is to assess system reliability and safety. In order to obtain such quality attributes, exceptional use cases are mapped to a statechart extension which added a probability attribute for each transition. Reliability and safety are then obtained by calculating the product of all transition probabilities that lead from one source state to a target one. These two works are complementary to our own since they provide a link between use case description and statechart specification.

Recent work by Castor et al. [5], in the Aereal framework, leverages existing languages and tools to support the description and analysis of exception control flow in software architectures. That work is similar to ours in its focus, but it differs on the way exception control flows are represented and on the goal of verification. While we provide a precise and light-weight specification of control flow at statechart level, and a high level representation at architecture level, the Aereal framework requires the exhaustive definition of complicated formalisms to represent flows only between two interconnected architectural elements. Moreover, besides the properties verified by Aereal (e.g., the existence of uncaught exceptions and useless handlers), we also support the enforcement of exception control flow during software evolution.

Some authors have proposed model-driven approaches [8,16] that support the definition of exception handling behaviour throughout

different perspectives. Entwisle et al. [8] proposes a model driven management framework that supports the definitions of strategies and policies across different tiers of an application. Pintér and Majzik [16] define a modelling pattern to support the definition of exception handling behaviour into statecharts. Unlike the EFlow approach, these works are not based on executable models. They rely on model transformation rules to automatically map the exceptional behaviour from the PIM to a PSM skeleton. Once the code generator converts the structural model to the target language, the normal behaviour code is added manually to the code skeleton. The problem with these approaches is that after generating the PSM skeleton, the developer is free to evolve the code as he/she desires to. Hence, these approaches do not provide any enforcement to guarantee the reliable evolution of the exceptional behaviour after deployment.

6. CONCLUDING REMARKS

This paper has presented an exception handling model whose main purpose is to make exceptional behaviour more resilient in the presence of changes. We leverage the WEAVR specification-level pointcuts to promote improved separation between normal and error handling code, while keeping track of exception control flows. We have implemented the most part of proposed model, with small syntactic additions to WEAVR. Our ongoing work encompasses the extension of the proposed model to support the notion of coordinated atomic actions.

7. REFERENCES

[1] Anderson, T. and Lee, P. A. *Fault Tolerance: Principles and Practice*. Prentice-Hall, 1981.

[2] Booch, G., Rumbaugh, J., and Jacobson, I. *Unified Modeling Language User Guide*, the (2nd Edition) (Addison-Wesley Object Technology Series). Addison-Wesley Professional.

[3] Cacho, N., Filho, F. C., Garcia, A., and Figueiredo, E. 2008. *EJFlow: Taming Exceptional Control Flows in Aspect-Oriented Programming*. In Proceedings of the 7th International Conference on Aspect-Oriented Software Development (Brussels, Belgium, March 31 - April 04, 2008). AOSD '08. ACM, New York, NY, 72-83.

[4] Cargill, T. *Exception Handling: A False Sense of Security*. C++ Report, vol. 6, no. 9,pp. 21-24, Nov.-Dec. 1994.

[5] Castor Filho, F., Brito, P. H. S., and Rubira, C. M. F. *Specification of exception Flow in software architectures*. Journal of Systems and Software, 79(10):1397.

[6] Cottenier, T., van den Berg, A., Elrad, T. *Joinpoint Inference from Behavioral Specification to Implementation*, In Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07), Berlin, Germany, LNCS 4609, Springer-Verlag, July 2007.

[7] Cottenier, T., van den Berg, A., Elrad, T. *The Motorola WEAVR: Model Weaving in a Large Industrial Context*, In Proceedings of the Industry Track at the 6th International Conference on Aspect-Oriented Software Development (AOSD'07). March 2007.

[8] Entwisle, S., Schmidt, H., Peake, I., and Kendall, E. 2006. *A Model Driven Exception Management Framework for Developing Reliable Software Systems.* In Proceedings of the 10th IEEE international Enterprise Distributed Object Computing Conference (October 16 - 20, 2006). EDOC. IEEE Computer Society, Washington, DC, 307-318.

[9] Harel, D. 1987. *Statecharts: A visual formalism for complex systems*. Sci. Comput. Program. 8, 3 (Jun. 1987), 231-274.

[10] Hong, H. S., Kim, Y. G., Cha S. D., Bae D.-H. and Ural, H. *A Test Sequence Selection Method for Statecharts*. STVR, 10(4):203--227, 2000.

[11] Kiczales, G. et al. *Aspect-oriented programming*. In Proceedings of ECOOP'97, pages 220–242, 1997.

[12] Malayeri, D. and Aldrich, J. *Practical Exception Specifications*. Advanced Topics in Exception Handling Techniques, LNCS, 2006.

[13] Mellor, S. J., Kendall, S., Uhl, A., and Weise, D. 2004 *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc.

[14] Mellor, S. J. and Balcer, M. 2002 *Executable Uml: a Foundation for Model-Driven Architectures.* Addison-Wesley Longman Publishing Co., Inc.

[15] Mustafiz, S. Sun, X. Kienzle, J. and Vangheluwe, H. 2006. *Model-Driven Assessment of Use Cases for Dependable Systems.* In 9th International Conference on Model Driven Engineering Languages and Systems -- MoDELS 2006, Genova, Italy, Oct. 1--6, 2006, no. 4199 in Lecture Notes in Computer Science, pp. 558 -- 573, Springer Verlag.

[16] Pintér, G. and Majzik, I. *Modeling and Analysis of Exception Handling by Using UML Statecharts*. FIDJI 2004. 58-67.

[17] Robillard, M. P. and Murphy, G. C. *Designing robust Java programs with exceptions.* In Proceedings of the 8th ACM SIGSOFT international Symposium on Foundations of Software Engineering: Twenty-First Century Applications. ACM Press, New York, 2000, 2-10.

[18] Robillard, M. P. and Murphy, G. C. *Static analysis to support the evolution of exception structure in object-oriented systems.* ACM Trans. Softw. Eng. Methodol. 12, 2 (Apr. 2003), 191-221.

[19] Shui, A., Mustafiz, S., Kienzle, J., Dony, C. *Exceptional use cases*. In Briand, L.C., Williams, C., eds.: MoDELS. Volume 3713 of Lecture Notes in Computer Science., Springer (2005) 568–583

[20]Szyperski, C. 2002 *Component Software: Beyond Object-Oriented Programming*. 2nd. Addison-Wesley Longman Publishing Co., Inc.