An Experience on Formal Analysis of a high-level graphical SOA Design

Maurice H. ter Beek

joint work with
Franco Mazzanti and Aldi Sulova

ISTI–CNR, Pisa, Italy

FM+AM @ SEFM 2010

Pisa, Italy
17 September 2010

Outline

Formal verification of SOC applications

Service-Oriented Computing

SOC is an evolutionary novel paradigm for developing loosely-coupled, interoperable, evolvable systems and applications, exploiting the pervasiveness of the Internet — underlying infrastructures called SOA

FP6-IST-FET IP SENSORIA (2005–2010)

Successfully developed a comprehensive approach to the engineering of SOC systems, in which foundational theories, techniques, and methods are integrated into a pragmatic software engineering process

Our research in SENSORIA: UMC/SocL (http://fmt.isti.cnr.it/umc/)

Develop formal reasoning mechanisms and analytic tools for checking that the services resulting from a composition meet desirable correctness properties and do not manifest any unexpected behaviour

Formal verification of SOC applications

Service-Oriented Computing

SOC is an evolutionary novel paradigm for developing loosely-coupled, interoperable, evolvable systems and applications, exploiting the pervasiveness of the Internet — underlying infrastructures called SOA

FP6-IST-FET IP SENSORIA (2005–2010)

Successfully developed a comprehensive approach to the engineering of SOC systems, in which foundational theories, techniques, and methods are integrated into a pragmatic software engineering process

Our research in SENSORIA: UMC/SocL (http://fmt.isti.cnr.it/umc/)

Develop formal reasoning mechanisms and analytic tools for checking that the services resulting from a composition meet desirable correctness properties and do not manifest any unexpected behaviour

Formal verification of SOC applications

Service-Oriented Computing

SOC is an evolutionary novel paradigm for developing loosely-coupled, interoperable, evolvable systems and applications, exploiting the pervasiveness of the Internet — underlying infrastructures called SOA

FP6-IST-FET IP SENSORIA (2005–2010)

Successfully developed a comprehensive approach to the engineering of SOC systems, in which foundational theories, techniques, and methods are integrated into a pragmatic software engineering process

Our research in SENSORIA: UMC/SocL (http://fmt.isti.cnr.it/umc/)

Develop formal reasoning mechanisms and analytic tools for checking that the services resulting from a composition meet desirable correctness properties and do not manifest any unexpected behaviour

UML4SOA

UML4SOA: a UML profile for service behaviour

- A UML 2.0 profile for modelling the *behavioural* aspects of SOA
- Complements OMG's SoaML profile, which defines a metamodel for designing the *structural* aspects of SOA

A UML profile consists of a set of stereotypes and constraints

- A stereotype is a limited kind of metaclass, defining the behaviour of classes and their instances, that may extend all UML modelling elements (e.g. states, transitions, activities, use cases, components)
- Stereotypes of the UML4SOA profile can thus be used to enrich UML models with service-oriented concepts, including structural and behavioural aspects of SOA as well as non-functional notions
- Constraints allow more precise semantics of the newly introduced modelling elements

UML4SOA

UML4SOA: a UML profile for service behaviour

- A UML 2.0 profile for modelling the *behavioural* aspects of SOA
- Complements OMG's SoaML profile, which defines a metamodel for designing the *structural* aspects of SOA

A UML profile consists of a set of stereotypes and constraints

- A stereotype is a limited kind of metaclass, defining the behaviour of classes and their instances, that may extend all UML modelling elements (e.g. states, transitions, activities, use cases, components)
- Stereotypes of the UML4SOA profile can thus be used to enrich UML models with service-oriented concepts, including structural and behavioural aspects of SOA as well as non-functional notions
- Constraints allow more precise semantics of the newly introduced modelling elements

UML4SOA — continued

Orchestration: a key aspect of service orientation

- A description of the interaction of (simpler, existing) services
- A behavioural specification of a service component in SoaML
- Modelled by UML activity diagrams stereotyped ≪service activity≫
- Focus on service interactions, long-running transactions, and their compensation and exception handling

Scope: a structured activity

- Groups actions, possibly with compensation / exception handlers
- Scopes and their handlers are linked by specific compensation and exception edges (stereotypes ≪compensation≫ and ≪event≫)
- Long-running transactions require managing compensations, called on all subscopes in reverse order of their completion

UML4SOA — continued

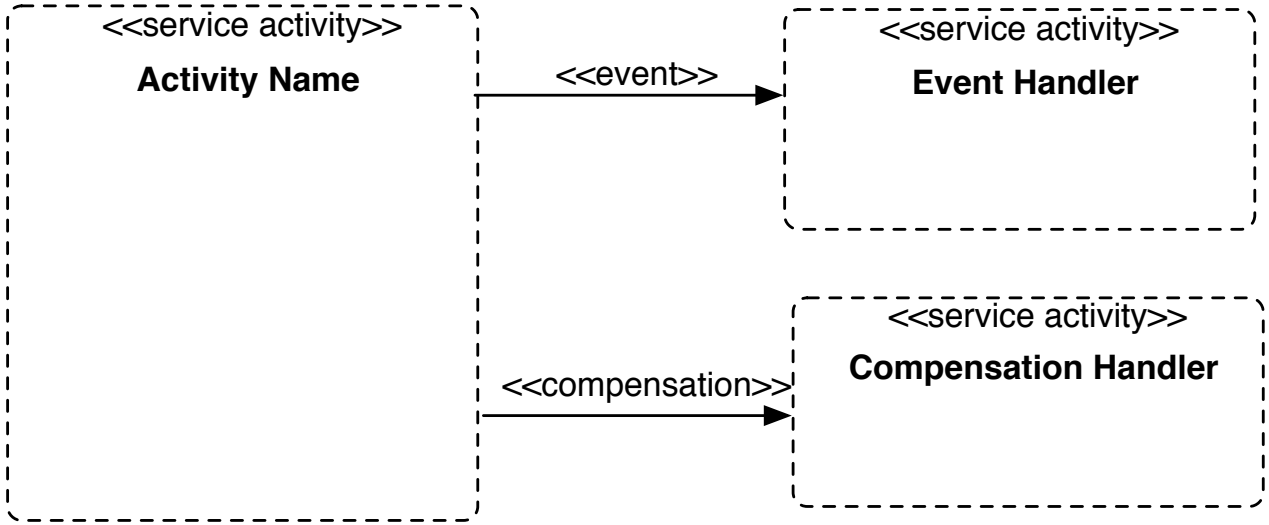
Orchestration: a key aspect of service orientation

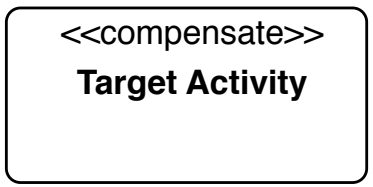
- A description of the interaction of (simpler, existing) services
- A behavioural specification of a service component in SoaML
- Modelled by UML activity diagrams stereotyped ≪service activity≫
- Focus on service interactions, long-running transactions, and their compensation and exception handling

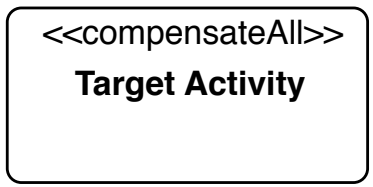
Scope: a structured activity

- Groups actions, possibly with compensation / exception handlers
- Scopes and their handlers are linked by specific compensation and exception edges (stereotypes ≪compensation≫ and ≪event≫)
- Long-running transactions require managing compensations, called on all subscopes in reverse order of their completion

UML4SOA: new graphical elements







Credit Portal scenario of Finance case study

Provided by one of SENSORIA's industrial partners S&N:

I. CreditRequest service offers clients the possibility to ask for a loan and properly orchestrates all steps needed to process this request

II. Client initiates a credit request by uploading data (name, address, desired amount, revenues, expenses, security values)

III. Rating service calculates the rating of the client's request:
   a) AAA automatically leads to an offer to the Client
   b) BBB: the request has some risk, but a clerk may decide
   c) CCC: a much more risky request, requiring a supervisor to decide

IV. The credit request is either rejected, in which case the Client can update the request information and retry, or an offer is sent to the Client, who can then accept or reject the offer

V. At any moment, the Client may abort the request, in which case the request data needs to be deleted: This requires the execution of compensation activities to semantically rollback the action of storing the request data performed by the involved services, preventing services to keep information of aborted requests

Credit Portal scenario of Finance case study

Provided by one of SENSORIA's industrial partners S&N:

I. CreditRequest service offers clients the possibility to ask for a loan and properly orchestrates all steps needed to process this request

II. Client initiates a credit request by uploading data (name, address, desired amount, revenues, expenses, security values)

III. Rating service calculates the rating of the client's request:
   a) AAA automatically leads to an offer to the Client
   b) BBB: the request has some risk, but a clerk may decide
   c) CCC: a much more risky request, requiring a supervisor to decide

IV. The credit request is either rejected, in which case the Client can update the request information and retry, or an offer is sent to the Client, who can then accept or reject the offer

V. At any moment, the Client may abort the request, in which case the request data needs to be deleted: This requires the execution of compensation activities to semantically rollback the action of storing the request data performed by the involved services, preventing services to keep information of aborted requests

Credit Portal scenario of Finance case study

Provided by one of SENSORIA's industrial partners S&N:

I. CreditRequest service offers clients the possibility to ask for a loan and properly orchestrates all steps needed to process this request
II. Client initiates a credit request by uploading data (name, address, desired amount, revenues, expenses, security values)
III. Rating service calculates the rating of the client's request:
   a) AAA automatically leads to an offer to the Client
   b) BBB: the request has some risk, but a clerk may decide
   c) CCC: a much more risky request, requiring a supervisor to decide
IV. The credit request is either rejected, in which case the Client can update the request information and retry, or an offer is sent to the Client, who can then accept or reject the offer
V. At any moment, the Client may abort the request, in which case the request data needs to be deleted: This requires the execution of compensation activities to semantically rollback the action of storing the request data performed by the involved services, preventing services to keep information of aborted requests

Credit Portal scenario of Finance case study

Provided by one of SENSORIA's industrial partners S&N:

I. CreditRequest service offers clients the possibility to ask for a loan and properly orchestrates all steps needed to process this request
II. Client initiates a credit request by uploading data (name, address, desired amount, revenues, expenses, security values)
III. Rating service calculates the rating of the client's request:
   a) AAA automatically leads to an offer to the Client
   b) BBB: the request has some risk, but a clerk may decide
   c) CCC: a much more risky request, requiring a supervisor to decide
IV. The credit request is either rejected, in which case the Client can update the request information and retry, or an offer is sent to the Client, who can then accept or reject the offer
V. At any moment, the Client may abort the request, in which case the request data needs to be deleted: This requires the execution of compensation activities to semantically rollback the action of storing the request data performed by the involved services, preventing services to keep information of aborted requests

Credit Portal scenario of Finance case study

Provided by one of SENSORIA's industrial partners S&N:

I CreditRequest service offers clients the possibility to ask for a loan and properly orchestrates all steps needed to process this request

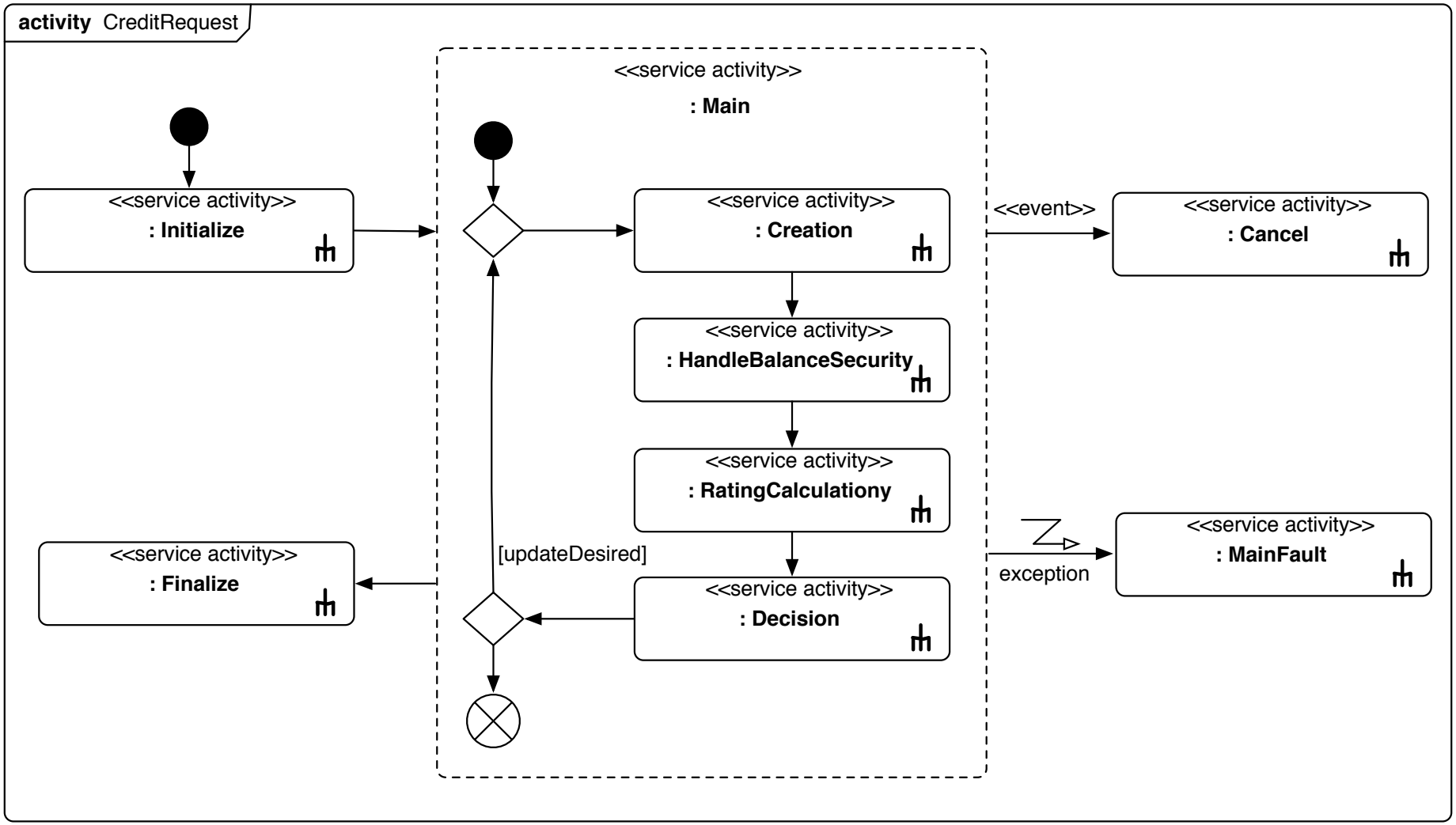
II Client initiates a credit request by uploading data (name, address, desired amount, revenues, expenses, security values)

III Rating service calculates the rating of the client's request:
   a) AAA automatically leads to an offer to the Client
   b) BBB: the request has some risk, but a clerk may decide
   c) CCC: a much more risky request, requiring a supervisor to decide

IV The credit request is either rejected, in which case the Client can update the request information and retry, or an offer is sent to the Client, who can then accept or reject the offer

V At any moment, the Client may abort the request, in which case the request data needs to be deleted: This requires the execution of compensation activities to semantically rollback the action of storing the request data performed by the involved services, preventing services to keep information of aborted requests

UML4SOA example: CreditRequest service



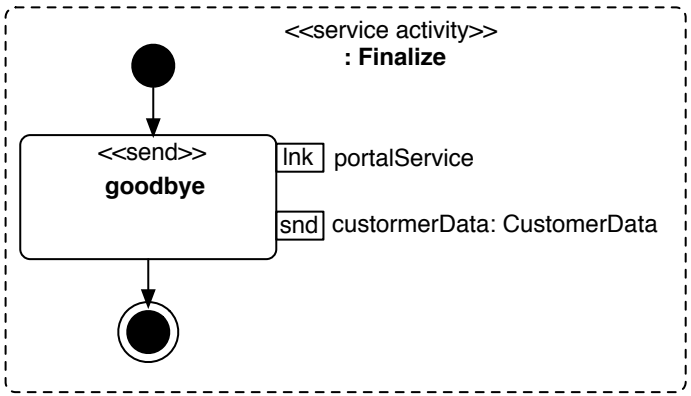
Translating UML4SOA diagrams into UMC statecharts

In general UML statechart transitions modelling the execution of an activity node have this form (tau represents an internal signal of the statechart):

```
s1 -> s1 -- actually no state change
{ tau [enabling conditions for incoming edges] /
      resetting of enabling conditions;
      execution of specific node activity;
      setting of enabling conditions for outgoing edges;
      self.tau;
}
```

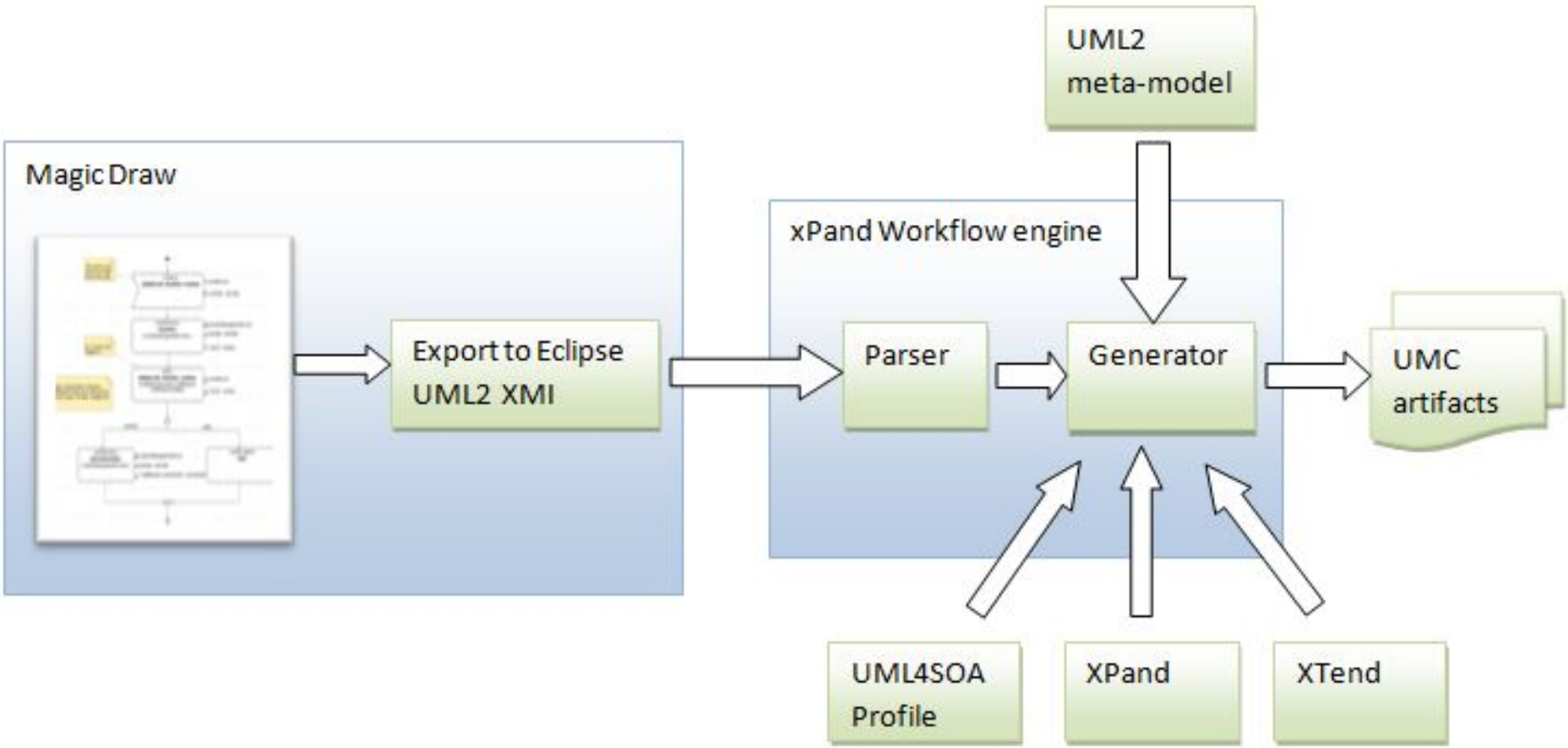
An example (blueprint for an automatic translation tool from UML4SOA specifications into UMC code):

```
s1 -> s1
{ tau [N1_Finalize_DefaultInitialOUT = true] /
      N1_Finalize_DefaultInitialOUT := false;
      LNK_portalService.goodbye([...],
        VAR_CreditRequest_customerData);
      N2_Finalize_Send_goodbyeOUT := true;
      self.tau;
}
```



Tool support: from MagicDraw to Eclipse to UMC

Accompany automatic translation from UML4SOA into executable BPEL code



Example transformation of a UML4SOA «send» node

Template.xpt

```
«REM»  "send" activate «ENDREM»
«ELSEIF currentNode.metaType.name.matches("uml::CallOperationAction")
              && ((uml::CallOperationAction)currentNode).
                    getAppliedStereotypes().exists(e | e.name.matches("send"))»
«"--- "»
«"--- Activation of node <<send>> -> "+
                           ((uml::CallOperationAction)currentNode).operation.name»
«"--- "»
«"s1 -> s1   { tau [  "»
«"          ("+currentNode.incoming().typeSelect(uml::ControlFlow).
                    first().source.getNodeVariables().select(e |
                          e.toString().contains("OUT")).first()+ "  = true) ] /"»

«"         "+currentNode.incoming().typeSelect(uml::ControlFlow).
                    first().source.getNodeVariables().select(e |
                             e.toString().contains("OUT")).first()+ " := false;"»
«"         -----<<send>> execution"»
«"        "+currentNode.getLNK_Var()+"."+
                  ((uml::CallOperationAction)currentNode).
                        operation.name+" ( "+"[self]"+
                          foreachVariable(currentNode.getSndPinParameter())+");"»

«"        "+currentNode.getNodeVariables().select(e |
                              e.toString().contains("OUT")).first()+" := true;"»
«"          self.tau;"»
«"    }"»
```

if the current node is a UML4SOA <<send>> node

testing the OUT flow variable of the node referred by the incoming edge

user- defined operations written in xTend

Lessons learned, or why FM+AM?

Purpose of case study: experiment novel design notation and novel model transformation and verification approaches — thus very informal software development process

1. Revealed several uncertainties in the language definition
2. Revealed several flaws in the designed model
3. Better understood hidden risks of apparently intuitive graphical design notations

Adoption of rigorous and formal semantics for these notations and of formal verification methods allow to overcome these problems

Automatic generation of formal models from high-level graphical designs is desirable and feasible, using appropriate model transformation techniques

Particularly valuable in the context of agile development approaches based on rapid and continuous updates of system designs

Lessons learned, or why FM+AM?

Purpose of case study: experiment novel design notation and novel model transformation and verification approaches — thus very informal software development process

1. Revealed several uncertainties in the language definition
2. Revealed several flaws in the designed model
3. Better understood hidden risks of apparently intuitive graphical design notations

Adoption of rigorous and formal semantics for these notations and of formal verification methods allow to overcome these problems

Automatic generation of formal models from high-level graphical designs is desirable and feasible, using appropriate model transformation techniques

Particularly valuable in the context of agile development approaches based on rapid and continuous updates of system designs

Lessons learned, or why FM+AM?

Purpose of case study: experiment novel design notation and novel model transformation and verification approaches — thus very informal software development process

1. Revealed several uncertainties in the language definition
2. Revealed several flaws in the designed model
3. Better understood hidden risks of apparently intuitive graphical design notations

Adoption of rigorous and formal semantics for these notations and of formal verification methods allow to overcome these problems

Automatic generation of formal models from high-level graphical designs is desirable and feasible, using appropriate model transformation techniques

Particularly valuable in the context of agile development approaches based on rapid and continuous updates of system designs

Lessons learned, or why FM+AM?

Purpose of case study: experiment novel design notation and novel model transformation and verification approaches — thus very informal software development process

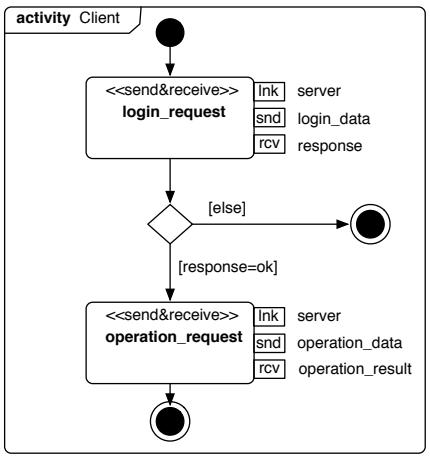
1. Revealed several uncertainties in the language definition
2. Revealed several flaws in the designed model
3. Better understood hidden risks of apparently intuitive graphical design notations

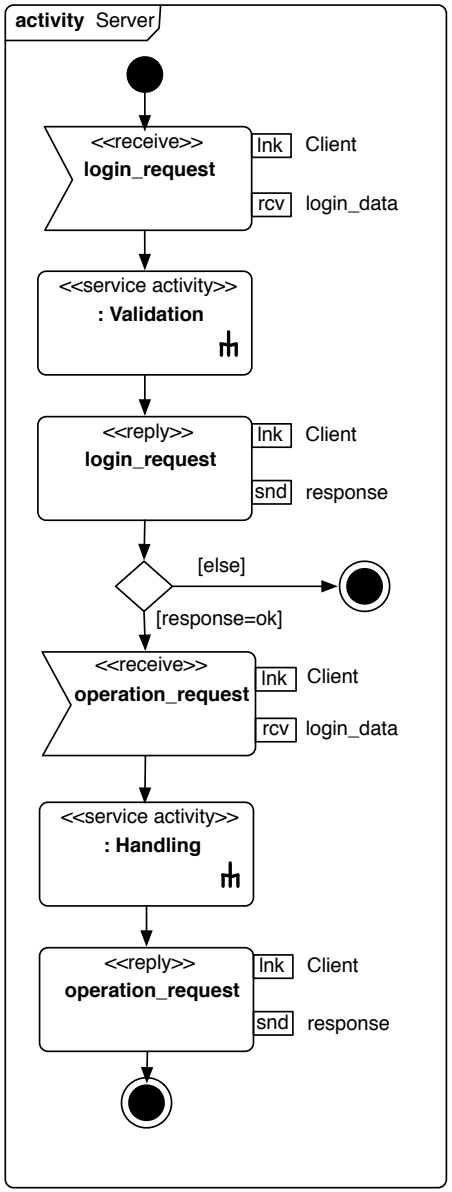
Adoption of rigorous and formal semantics for these notations and of formal verification methods allow to overcome these problems

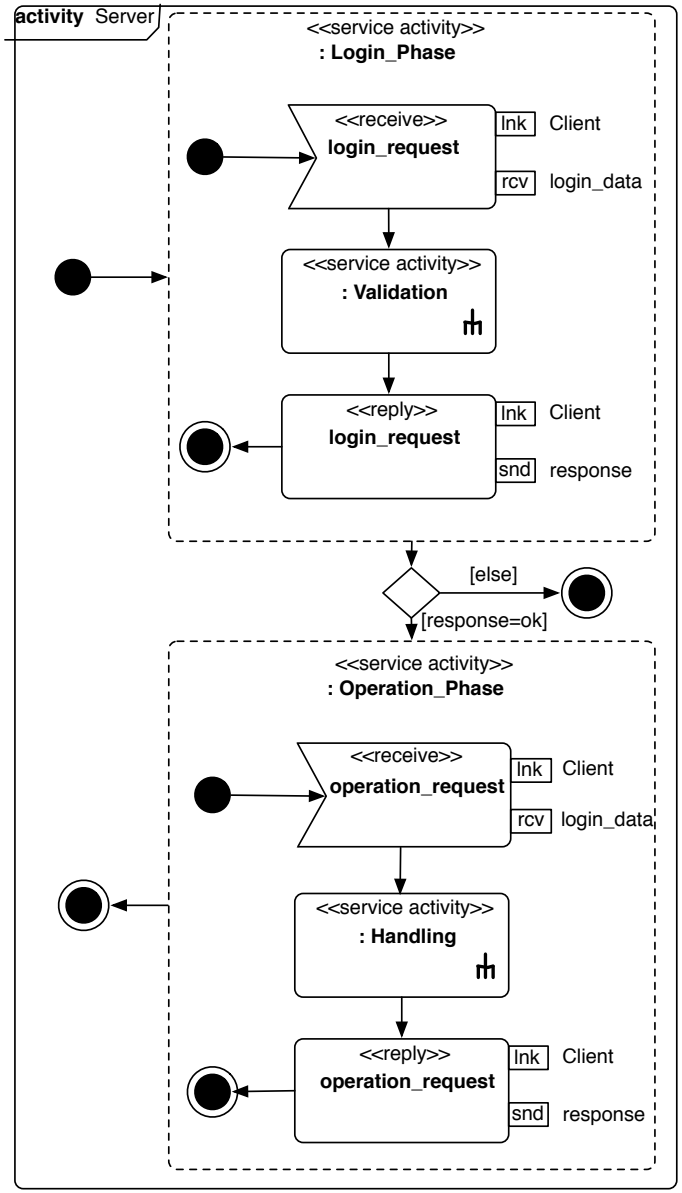
Automatic generation of formal models from high-level graphical designs is desirable and feasible, using appropriate model transformation techniques

Particularly valuable in the context of agile development approaches based on rapid and continuous updates of system designs

1. Hidden implementation-dependent assumptions inside “high-level” “platform-independent” designs







Examples of hidden implementation-dependent assumptions in UML4SOA design of case study

"Structured" design: Server might receive **operation_request** before **Operation_Phase** is entered (and before receiving request is enabled)

Semantics of design highly implementation dependent: depends on what should happen when a message arrives at a component when it is not yet ready to accept it (e.g. queued on Server? discarded?)

Designer is probably making some *implicit* underlying assumption:

- ? Time needed by (remote) Client to receive login response and produce operation request is much higher than that needed by Server to perform all internal steps between when response is provided and when receive is enabled (even if reasonable, this should be reflected by the design)
- ? Incoming operation request messages are not discarded, but queued

Examples of hidden implementation-dependent assumptions in UML4SOA design of case study

"Structured" design: Server might receive **operation_request** before **Operation_Phase** is entered (and before receiving request is enabled)

Semantics of design highly implementation dependent: depends on what should happen when a message arrives at a component when it is not yet ready to accept it (e.g. queued on Server? discarded?)

Designer is probably making some *implicit* underlying assumption:

- ? Time needed by (remote) Client to receive login response and produce operation request is much higher than that needed by Server to perform all internal steps between when response is provided and when receive is enabled (even if reasonable, this should be reflected by the design)
- ? Incoming operation request messages are not discarded, but queued

UMC can detect hidden assumptions in UML4SOA

Without assumptions, generating a UMC model of "structured" Server design allows to detect deadlocks in the design

Example: model checking the property "a successful login request is always eventually followed by a response to the operation request"

UMC can make *explicit* the probable underlying assumptions:

- ? Raise the priority of a component for a short period of time in order to execute a sequence of internal steps as a unique indivisible activity
- ? Define any incoming messages not supposed to be discarded, even if arriving when a component is not yet able to handle them, as "deferred" (in UML statecharts sense) to achieve behaviour of queued messages

Result: behaviour of the "structured" design becomes, as intuitively expected, equivalent to that of the flat design

UMC can detect hidden assumptions in UML4SOA

Without assumptions, generating a UMC model of "structured" Server design allows to detect deadlocks in the design

Example: model checking the property "a successful login request is always eventually followed by a response to the operation request"

UMC can make *explicit* the probable underlying assumptions:

- ? Raise the priority of a component for a short period of time in order to execute a sequence of internal steps as a unique indivisible activity
- ? Define any incoming messages not supposed to be discarded, even if arriving when a component is not yet able to handle them, as "deferred" (in UML statecharts sense) to achieve behaviour of queued messages

Result: behaviour of the "structured" design becomes, as intuitively expected, equivalent to that of the flat design

UMC can detect hidden assumptions in UML4SOA

Without assumptions, generating a UMC model of "structured" Server design allows to detect deadlocks in the design

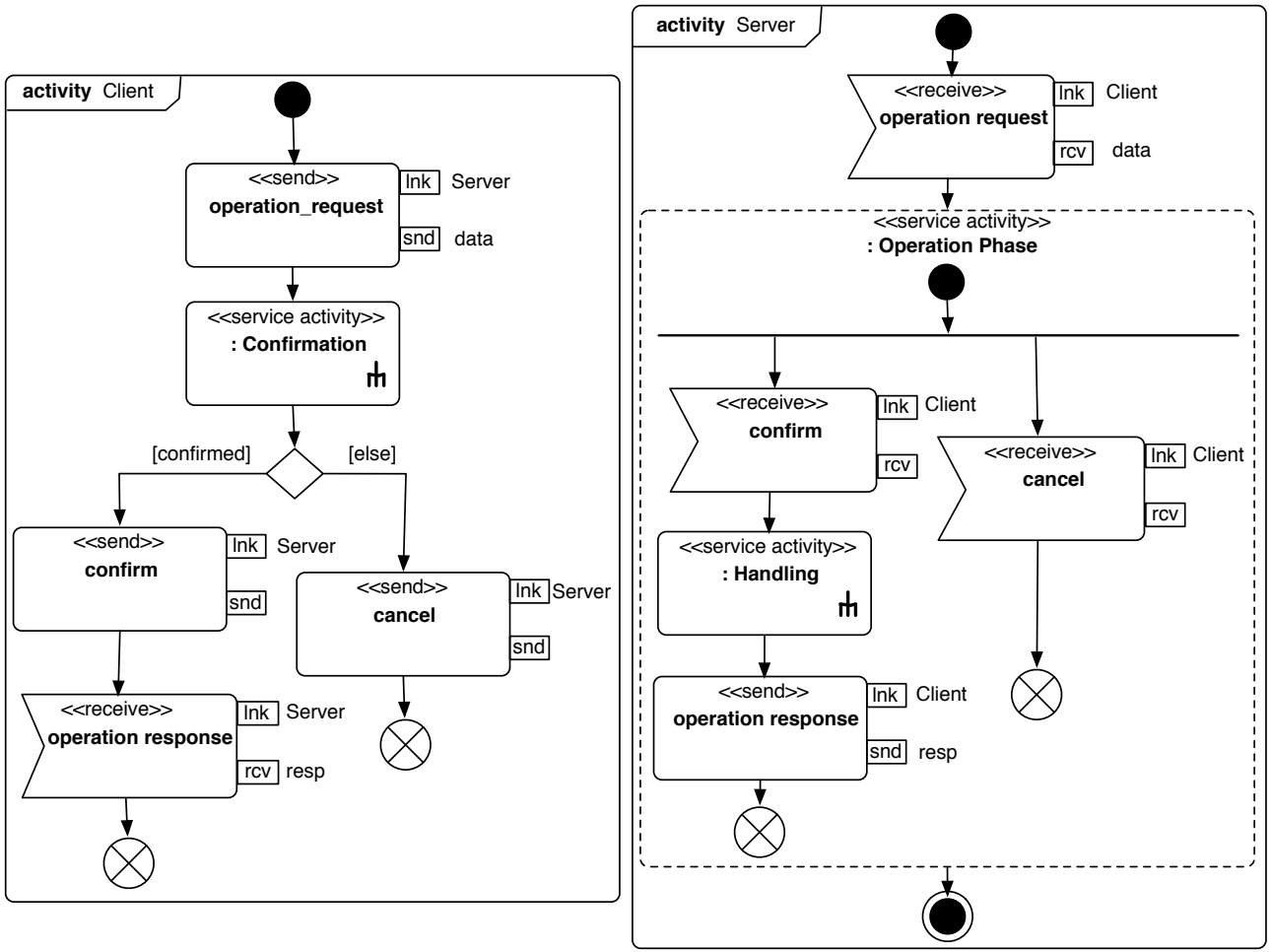
Example: model checking the property "a successful login request is always eventually followed by a response to the operation request"

UMC can make *explicit* the probable underlying assumptions:

- ? Raise the priority of a component for a short period of time in order to execute a sequence of internal steps as a unique indivisible activity
- ? Define any incoming messages not supposed to be discarded, even if arriving when a component is not yet able to handle them, as "deferred" (in UML statecharts sense) to achieve behaviour of queued messages

Result: behaviour of the "structured" design becomes, as intuitively expected, equivalent to that of the flat design

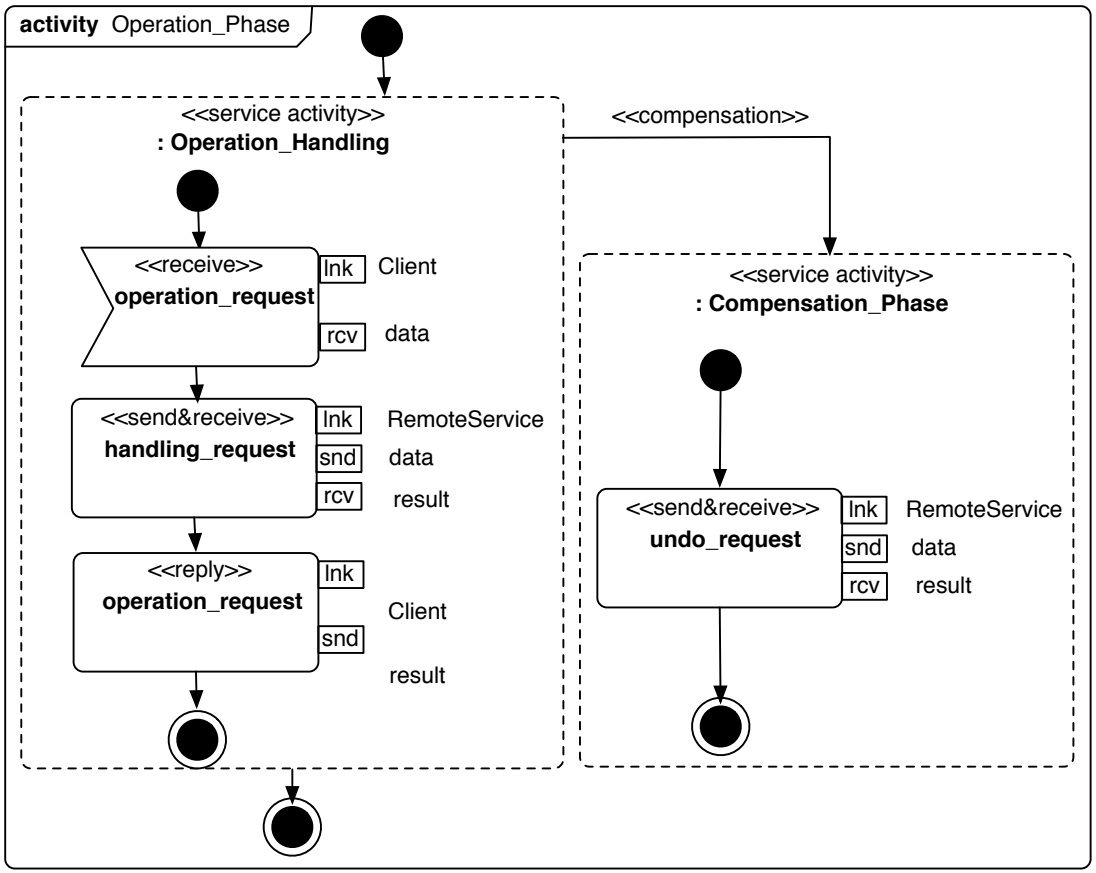
Another exemplary hidden assumption UMC can detect



What if **operation_request** is delivered after **confirm** or **cancel**? Strictly implementation-dependent: deadlock if wrongly timed messages are discarded

In UMC messages can be delivered in a different order from the one in which they are sent by specifying the specific queueing policy (RANDOM vs. FIFO)

2. Uncertain semantics of UML4SOA features



Compensation: **undo_request** to "undo" **handling_request** (no instance of the latter, associated to an unsuccessful credit request, should be present)

If compensation request is executed after **handling_request** service call is issued, but before **Operation_Handling** activity is completed, then no compensation is activated and an orphan **handling_request** instance does exist

UMC evinces two kind of problems

1. The semantics of compensation should probably be tied with an atomic transaction mechanism, a fact that currently is not well described by the UML4SOA definition of service activities and compensations
2. The designer might implicitly be assuming that execution of the activity **Operation_Handling** cannot be interrupted from the outside

   As it contains no error-path, it is assumed to always finish successfully

   Unfortunately, the overall system design allows two parallel threads to asynchronously interfere with one another, and at first the impact of this possibility was not well understood in the case study's UML4SOA design (where the asynchronous interfering activity is the one originated by a **cancel** operation triggered by the Client)

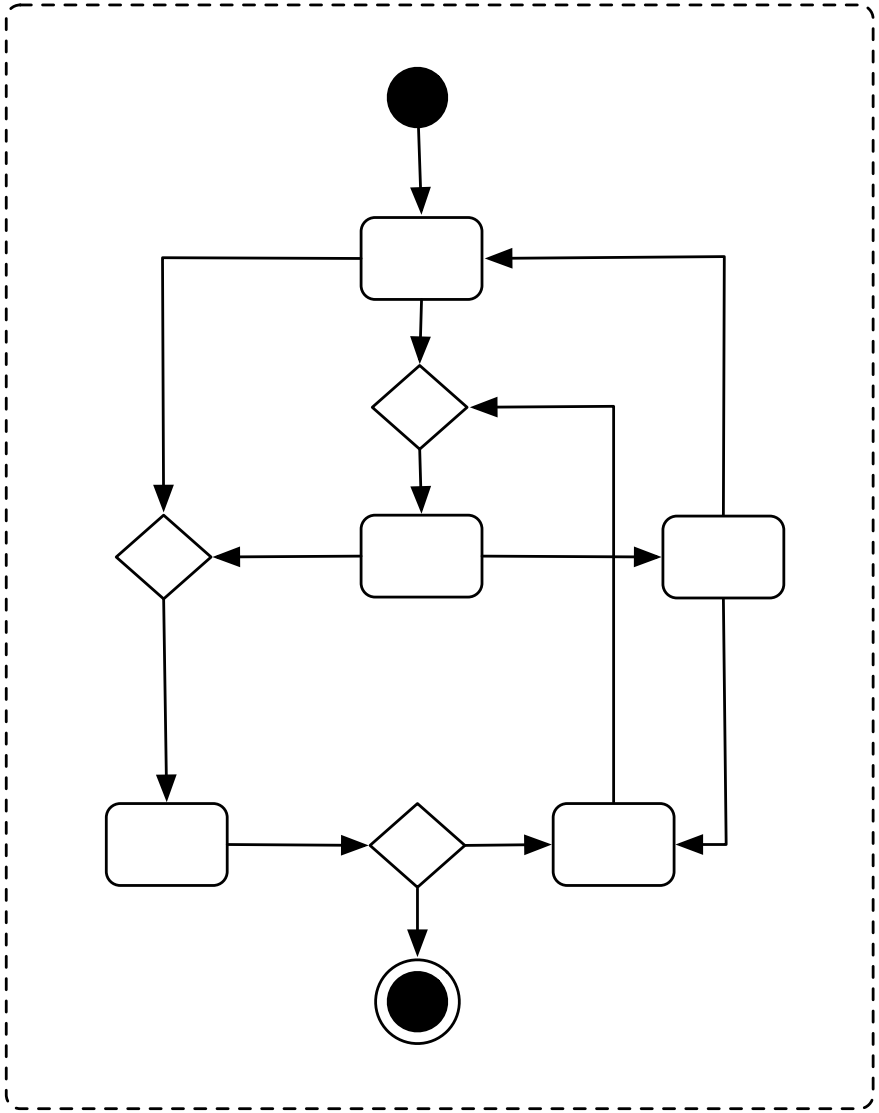
UMC evinces two kind of problems

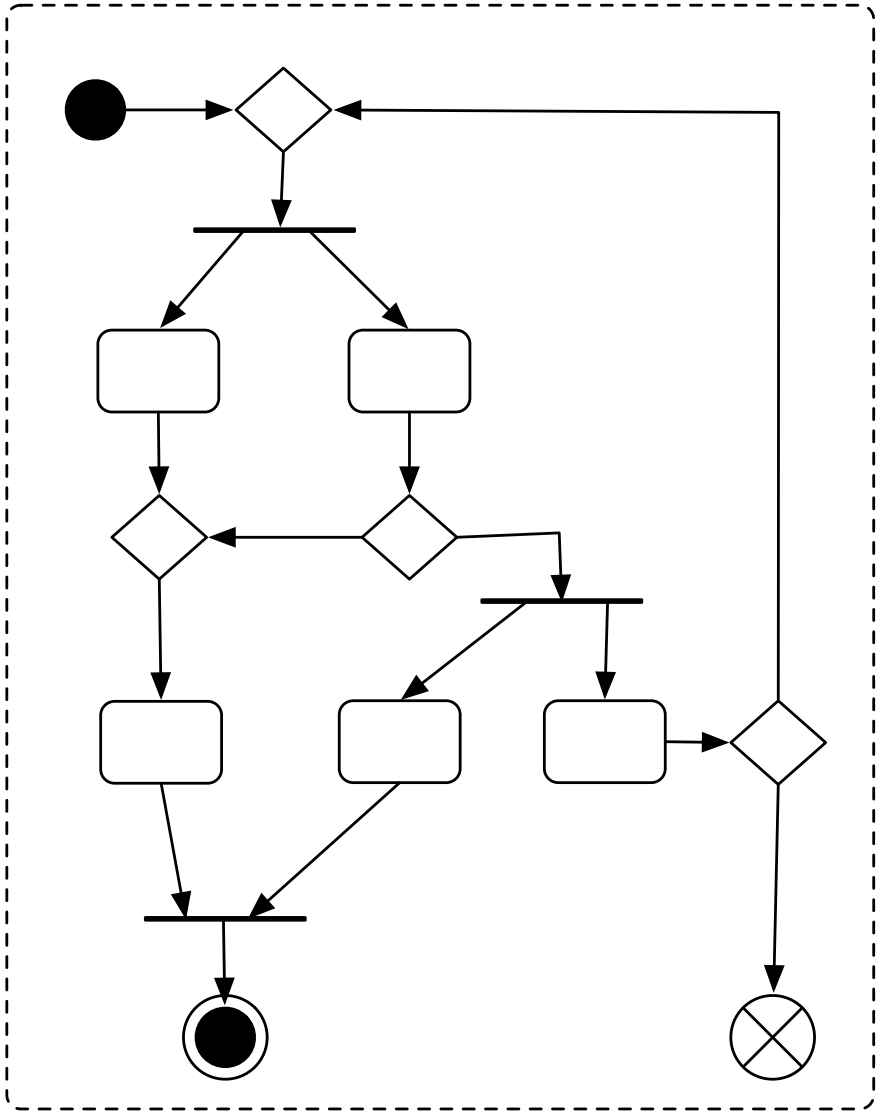
1. The semantics of compensation should probably be tied with an atomic transaction mechanism, a fact that currently is not well described by the UML4SOA definition of service activities and compensations
2. The designer might implicitly be assuming that execution of the activity **Operation_Handling** cannot be interrupted from the outside

   As it contains no error-path, it is assumed to always finish successfully

   Unfortunately, the overall system design allows two parallel threads to asynchronously interfere with one another, and at first the impact of this possibility was not well understood in the case study's UML4SOA design (where the asynchronous interfering activity is the one originated by a **cancel** operation triggered by the Client)

3. Hidden complexity of scarcely structured designs





Weren't gotos considered harmful?

Edsger W. Dijkstra, "Go To Statement Considered Harmful". *Communications of the ACM* 11(3): 147–148 (**March 1968**).

> *The unbridled use of the go to statement has as an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. [...] The go to statement as it stands is just too primitive, it is too much an invitation to make a mess of one's program.*

(This is at the same time one of the most often cited as well as one the least read documents about programming)

High-level graphical design languages often lead to "spaghetti design", whose true behaviour is difficult to understand in all its ramifications

Weren't gotos considered harmful?

Edsger W. Dijkstra, "Go To Statement Considered Harmful". *Communications of the ACM* 11(3): 147–148 (**March 1968**).

> *The unbridled use of the go to statement has as an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. [...] The go to statement as it stands is just too primitive, it is too much an invitation to make a mess of one's program.*

(This is at the same time one of the most often cited as well as one the least read documents about programming)

High-level graphical design languages often lead to "spaghetti design", whose true behaviour is difficult to understand in all its ramifications

Concurrency

In most programming languages great care is given to concurrent features (task, threads, co-routines) with the design goal of well identifying the concurrently evolving activities and, above all, keeping the flow of concurrent elements independent as much as possible

High-level graphical design languages often resort to low-level graphical elements, like "fork" and "join", to handle concurrency, potentially allowing even nastier "spaghetti designs"

Situation even worse when exploiting "activity final" nodes (killing all concurrent subactivities inside a parallel activity) or raising exceptions

(a consequence may be that an error in one concurrent subactivity asynchronously terminates other concurrent activities in an unclear, implicit, or uncontrolled way)

Concurrency

In most programming languages great care is given to concurrent features (task, threads, co-routines) with the design goal of well identifying the concurrently evolving activities and, above all, keeping the flow of concurrent elements independent as much as possible

High-level graphical design languages often resort to low-level graphical elements, like "fork" and "join", to handle concurrency, potentially allowing even nastier "spaghetti designs"

Situation even worse when exploiting "activity final" nodes (killing all concurrent subactivities inside a parallel activity) or raising exceptions

(a consequence may be that an error in one concurrent subactivity asynchronously terminates other concurrent activities in an unclear, implicit, or uncontrolled way)

Concurrency

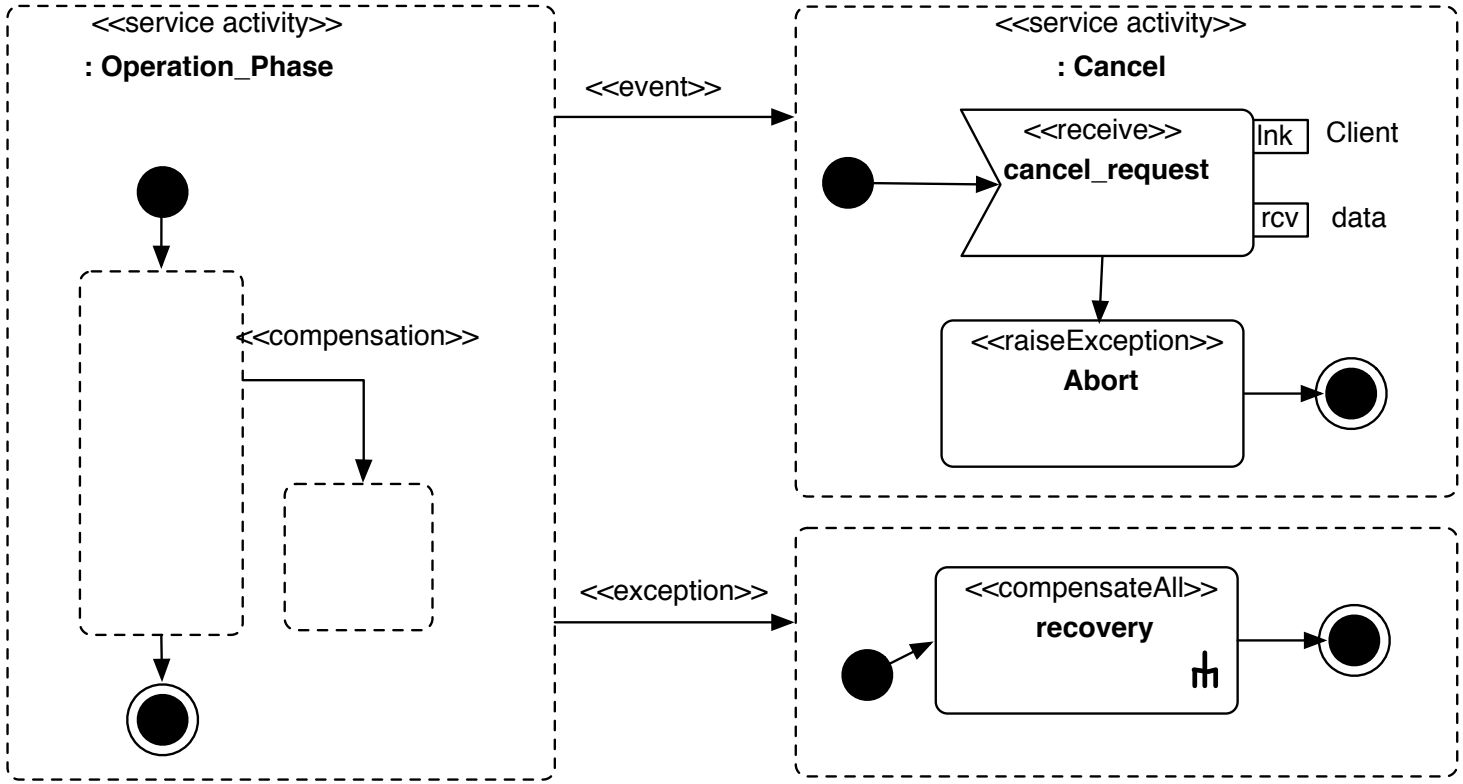
In most programming languages great care is given to concurrent features (task, threads, co-routines) with the design goal of well identifying the concurrently evolving activities and, above all, keeping the flow of concurrent elements independent as much as possible

High-level graphical design languages often resort to low-level graphical elements, like "fork" and "join", to handle concurrency, potentially allowing even nastier "spaghetti designs"

Situation even worse when exploiting "activity final" nodes (killing all concurrent subactivities inside a parallel activity) or raising exceptions

(a consequence may be that an error in one concurrent subactivity asynchronously terminates other concurrent activities in an unclear, implicit, or uncontrolled way)

Example of bad interference between two concurrent activities in UML4SOA design of case study



Operation_Phase is unexpectedly aborted because of an exception raised inside the **Cancel** activity, asynchronously triggered as a concurrent flow by an external event

Conclusions

High-level graphical design notations very intuitive and very efficient to describe the current structure and status of ongoing software projects

Especially if associated with automatic code generation / verification features, they may play an important role in agile software development

Facilitate the cooperation between the clients and the developers

Our case study shows they may also be a source of problems, most of which the use of formal methods can avoid

We believe high-level graphical design notations always need to be backed up by precise and rigorous semantics, i.e. by formal methods, especially when planned to be used in agile software development

Conclusions

High-level graphical design notations very intuitive and very efficient to describe the current structure and status of ongoing software projects

Especially if associated with automatic code generation / verification features, they may play an important role in agile software development

Facilitate the cooperation between the clients and the developers

Our case study shows they may also be a source of problems, most of which the use of formal methods can avoid

We believe high-level graphical design notations always need to be backed up by precise and rigorous semantics, i.e. by formal methods, especially when planned to be used in agile software development

Conclusions

High-level graphical design notations very intuitive and very efficient to describe the current structure and status of ongoing software projects

Especially if associated with automatic code generation / verification features, they may play an important role in agile software development

Facilitate the cooperation between the clients and the developers

Our case study shows they may also be a source of problems, most of which the use of formal methods can avoid

We believe high-level graphical design notations always need to be backed up by precise and rigorous semantics, i.e. by formal methods, especially when planned to be used in agile software development

Conclusions

High-level graphical design notations very intuitive and very efficient to describe the current structure and status of ongoing software projects

Especially if associated with automatic code generation / verification features, they may play an important role in agile software development

Facilitate the cooperation between the clients and the developers

Our case study shows they may also be a source of problems, most of which the use of formal methods can avoid

We believe high-level graphical design notations always need to be backed up by precise and rigorous semantics, i.e. by formal methods, especially when planned to be used in agile software development

Conclusions

High-level graphical design notations very intuitive and very efficient to describe the current structure and status of ongoing software projects

Especially if associated with automatic code generation / verification features, they may play an important role in agile software development

Facilitate the cooperation between the clients and the developers

Our case study shows they may also be a source of problems, most of which the use of formal methods can avoid

We believe high-level graphical design notations always need to be backed up by precise and rigorous semantics, i.e. by formal methods, especially when planned to be used in agile software development