Electronic Communications of the EASST
Volume 47 (2012)

Proceedings of the 11th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2012)

Remedy of Mixed Initiative Conflicts in Model-based System Engineering

Fenglin Han and Peter Herrmann

14 pages

Guest Editors: Andrew Fish, Leen Lambers
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Remedy of Mixed Initiative Conflicts in Model-based System Engineering

Fenglin Han[1] **and Peter Herrmann**[2]

[1] sih@item.ntnu.no
[2] herrmann@item.ntnu.no
http://item.ntnu.no/
Department of Telematics
Norwegian University of Science and Technology, Trondheim, Norway

Abstract: SPACE is a technique for model-driven engineering of reactive distributed systems. One of the strengths of its tool-set Arctis is that the system engineer can formally analyze the models for design errors such that these can be corrected early in the development process. In this paper, we go a step further and introduce a technique that refines the fault detection and, in addition, offers a highly automatic mechanism to remedy the errors. For that, we combine model checking, the already existing analysis method of Arctis, with graph transformation. Using graph rewriting rules, we can analyze the state space graph of a system for the exact reason of an error as well as remove the erroneous parts of a model by changing the model description. We exemplify the approach by envisaging the detection and remedy of mixed initiatives, a quite common cause for faulty behavior in event-driven systems that often is overlooked in system development.

Keywords: graph transformation, model driven engineering, mixed initiative.

1 Introduction

New application domains like sensor networks, smart grids, and machine to machine cooperation call for novel networked services and applications. To engineer these often reactive and embedded distributed systems, we provide the development method SPACE and its tool-set Arctis [KSH09, KH09]. System behavior is modeled by UML activities [Obj10] that use a token semantics close to Petri nets. The activities have been provided with a new reactive formal semantics [KH10] that enables to analyze the models formally [KSH09] and to create code automatically [KH07]. The technique is scalable since we can enclose partial behavior into UML call behavior actions that we call *blocks*. On the one hand, a block embraces an activity and, on the other, it can be used as an element in another one. So, it links both activities together. The static role-binding of the blocks is modeled by UML collaborations while we use External State Machines (ESMs, [KH09]) to define the interface behavior of a block. Furthermore, the block structure enables a high degree of reuse of system models which is in average 70% in our models [KH09].

Arctis enables the formal analysis of desirable system properties (e.g., verifying that the activity embraced in a block fulfills its ESM) by model checking [KSH09]. Since model checking

can be executed in a non-interactive way and the traces towards detected errors are animated on the UML activities, the Arctis user does not need a deeper understanding of the formalism used. The state explosion problem of model checking is mitigated by compositional verification since the ESMs allow to prove every activity in the system model separately [KSH09].

Up to now, the Arctis analysis has not supported automatic remedy of detected errors which has to be provided manually by the system engineer. This paper describes a solution to support the Arctis user further. In particular, we introduce a way to analyze the state graph of an erroneous model and to correct the specification with a high degree of automation. For that, we apply graph transformation that already proved helpful for the model transformation of flow-global choreographies, a more abstract way to specify reactive systems, to Arctis models [HKL+11]. Graph transformation is suitable since UML activities, collaborations and state machines all are graphical description techniques which can be directly accessed by graph rewriting rules.

For the error detection, we align graph transformation with model checking in Arctis. In particular, we analyze the state space of an activity generated by the model checker to find out if the detected errors belong to a particular class. Thereafter, we use rules to correct the erroneous part of the activity. We illustrate our approach with a mechanism for the detection and remedy of mixed initiatives between two parties.

2 Mixed Initiatives

Mixed initiatives [GY84, BH93, SDW08] are a special form of race conditions that is often overseen when developing reactive distributed systems. A mixed initiative conflict may occur whenever two (or more) distributed components can trigger an interaction with each other at the same time (see [Flo03]). Due to the asynchronous communication between the two parties, both can start own initiatives before being notified that also the partner triggered one. Of course, this behavior can be mitigated but that often demands for a complex new functionality. While mixed initiatives in practice are often overlooked, the Arctis model checker detects that a system contains faulty behavior (see [KSH07]). By the methods introduced in this paper, we can find out if a mixed initiative is indeed the reason of an error. Further, we introduce a graph-based approach to correct mixed initiatives between two entities. The only condition for the remedy mechanism is that the system behavior contains an interaction between the entities before the mixed initiative can take place.

2.1 Arctis

The Arctis block *Button Game* depicted in Figure 1 (a) describes a simple game with two players which is won by the one who manages first to push a button. The players are represented by two technical components, e.g., two Android devices. As mentioned in the introduction, we model behavior by UML activities which are based on token flow semantics (see [KH10]). The activity shown in Figure 1 (a) is collaborative since it models the combined interaction of the two components *component 1* and *component 2* participating in the button game. To distinguish in which component a certain behavioral step takes place, the activity comprises two partitions marked by the component names.

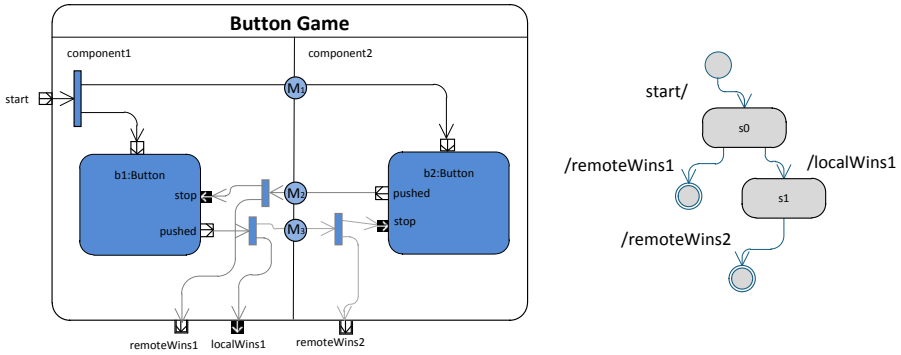

Figure 1: Arctis building block *Button Game* and its ESM

A UML activity [Obj10] is a directed graph and, due to the token flow semantics, behavior is modeled as tokens walking across the nodes of the graph following its edges. The edges may either stay within a partition, specifying local behavior of the corresponding component, or cross the partition borders (e.g., M_1, M_2, M_3). In the latter case, they model asynchronous interaction between two components. Activities offer a set of general node types enabling to start, stop, or interrupt token flows as well as nodes for routing and handling parallel flows respective for the execution of certain operations. An example are forks of which four copies are used in Figure 1 (a). They are expressed by bold bars in right angle to the linked edges. A fork contains one incoming edge and at least two outgoing edges and models that an incoming token is duplicated and a copy is sent via each of the downstream edges. Thus, forks enable to specify parallel flows.

Another node type used in the activity *Button Game* are call behavior actions describing the Arctis building blocks. A block represents an own activity that is linked with the one including it by means of pins[1] which are depicted as small rectangles on the edge of a block respective an activity and filled with in- or outgoing arrows. The interface behavior of a block is specified by External State Machines (ESM, [KH09]) that are simple UML state machines describing in which order flows may pass the various pins.

Figure 1 (a) includes the two blocks *b1* and *b2* of the type *Button* which are taken from an Arctis library for Android devices (see [Kra11]). This block type describes the logic when pushing a certain button of an Android device which initially is inactive. The button is armed by sending a token flow through its pin *start* on the top of the block. Thereafter, pushing the button leads to a flow via the pin *pushed* which, in addition, terminates and disarms the block. Further, the block can be disarmed from its environment by a token passing pin *stop*, too.

The Arctis semantics [KH10] defines so-called activity steps describing the sub-graph passed by a token in an atomic transition. In short, a token may rest only on nodes or edges describing places where it has to wait for a stimulus. An example are the crossing edges. To model the asynchronous communications between the partitions, a token passing a crossing edge has to wait on it until it is passed on in a new activity step. After being triggered by an internal or

[1] Formally, UML distinguishes between parameter nodes laying on the outer edge of an activity (e.g., *remoteWins1*) and pins on the edge of a block included in an activity (e.g., *stop*).

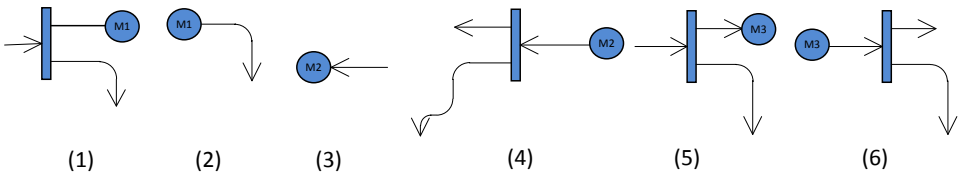

Figure 2: Activity steps of *Button Game*

external event, e.g., the reception of the transmitted data, tokens pass all nodes and edges of the activity step in run-to-completion fashion until they reach nodes and edges on which they have to wait for new triggers.

Figure 2 shows the six activity steps of the UML activity in Figure 1 (a). Activity step (1) describes the start of the game. It is triggered by a token passing the parameter node *start* at the edge of the block *Button Game* which is duplicated at the fork node and copies are sent to both block *b1* and to the crossing edge M_1. Activity step (2) forwards the token to block *b2* such that both buttons are armed. Pushing a button leads to the activity steps (3) and (4) respective (5) and (6) which disarm the other button and notify the environment of the button game about the winner via the parameter nodes *localWins* and *remoteWins*.[2]

Figure 1 (b) shows the External State Machine (ESM, [KH09]) of the block *Button Game*. The block is started by a token passing pin *start*. Thereafter, it terminates either by a token arriving at *remoteWins1* or by one at *localWins1* followed by another one at *remoteWins2*. In the transition markings, the "/" behind a pin designator refers to tokens heading towards the block while positioning "/" in front refers to tokens coming from the block and going towards its environment.

2.2 A Mixed Initiative Error

It is easy to see that the system renders unexpected behavior if both buttons are pushed at the same time. Then due to the asynchronous communication between the components, the buttons will be terminated too late and tokens leave the pins *pushed* of both block *b1* and *b2*. In consequence, all the activity steps (3) to (6) are executed and the ESM in Figure 1 (b) is violated as all *localWins1*, *remoteWins1* and *remoteWins2* are fired. This will be detected by the Arctis analyzer using model checking. Figure 3 (a) depicts the state space generated by the Arctis analyzer. To facilitate the understanding of the state graph, we added the identifiers of the edges crossing the partition borders. One can see that the traces towards the states 8 and 9 contain a sequence of pins violating the ESM.

[2] For simplicity, we only notify *component 1*, that takes the role of the game manager, fully about the result while *component 2* is just informed if it lost.

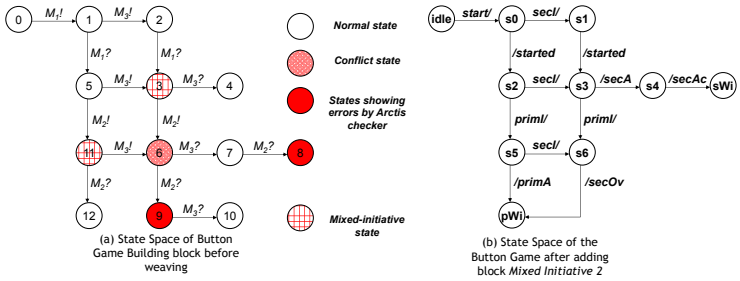

Figure 3: State spaces of the original and the modified block *Button Game*

3 Mixed Initiative Detection

To identify that a mixed initiative is the actual cause for property violations detected by the Arctis model checker, we investigate the state space further using graph rewriting. According to Floch [Flo03], a mixed initiative is indicated by so-called *mixed initiative states* in which a component may both send and consume signals. The danger of a mixed initiative exists if two interacting components are both in a mixed initiative state at the same time.

In a first step, we label the edges with send and receive labels using the technique explained in [Kra09]. In our example that are M_1, M_2 and M_3 referring to the three partition crossing edges in the activity depicted in Figure 1 (a). By the "!" we describe the sending and by "?" the reception of a communication between the two components. Since M_1 and M_3 show communication from *component 1* to *component 2* and M_2 the other way around, one can see that the states 3 and 11 of the state graph refer to mixed initiative states according to the definition of Floch. In state 3, *component 2* may both send M_2 and receive M_3 while in state 11 *component 1* may send M_3 and receive M_2. By executing the corresponding send actions, both mixed initiative states lead to state 6 which expresses that the two signals forwarding the conflicting initiatives just pass each other. We call it a *conflict state*.

Thereafter, we can check if the traces from the initial node towards the states violating a property always lead via a conflict state. To avoid false positives, we only assume a mixed initiative as the source of errors if all traces to all error states pass at least one conflict state. In our example, the violation of the ESM is detected when reaching the states 8 or 9 and it is easy to see that all traces from the initial state 0 to them pass the conflict state 6.

Technically, we export the state graph created by the Arctis model checker and label the states using the graph transformation tool AGG [Tae04] according to the following rules:

1. Initially, the subgraph contains the error states of the state graph.

2. For any vertex in the subgraph that is not a conflict state, we add all its incoming edges as well as their source states to the subgraph.

3. We terminate if we either added the initial state to the subgraph or if all states not yet treated in step 2 are conflict states.

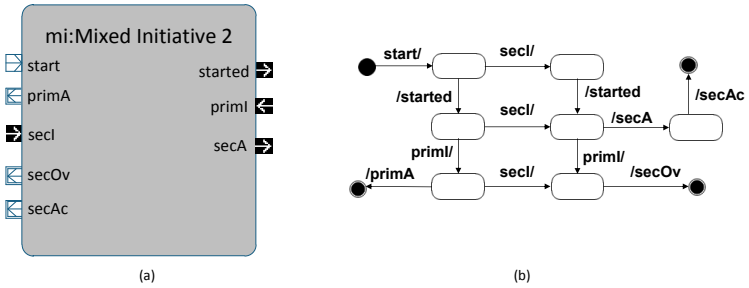

Figure 4: The Arctis block *Mixed Initiative 2*

Thus, if the initial node is not in the resulting subgraph, we know that all traces from it to the error states pass a conflict state which gives us advice that an improperly handled mixed initiative might be the source of the errors. For example, in Figure 3 (a) the subgraph consists of the states 6 to 9 and the edges linking them but not the initial state 0 showing that the mixed initiative between the crossing edges M_2 and M_3 are the likely reason for the problem.

The AGG rules use the state space generated by the Arctis model checker as input and are executed automatically without further human intervention. Thus, it is also possible to integrate the algorithm into the Arctis model checker which would enable a seamless detection of mixed initiatives already during the analysis. The integration is planned for one of the next revisions of the model checker.

4 Mixed Initiative Remedy

An established way to deal with errors caused by mixed initiatives is to use prioritization [GY84]. Here, the two conflicting initiatives are marked as *primary* respective *secondary* and, in the case of a conflict, only the primary initiative will take place while the secondary is stopped during communication. In our example, we decided that the initiative M_2 leaving *component 2* shall be the primary one, such that it will always be forwarded to *component 1* while M_3 will be stopped in the case of a conflict. Of course, this prioritization scheme demands a somehow complex logic which, however, can be hidden in a reusable Arctis block as we point out in the following.

4.1 Arctis Blocks handling Mixed Initiatives

Since mixed initiatives are a recurrent phenomenon in reactive distributed software, we created two building blocks providing remedy by prioritization (see [KSH07]) which are available in one of the Arctis libraries. Figure 4 (a) shows one of them. It supports two participants *primary* and *secondary* and arranges that an initiative from the primary one is prioritized against the one of the secondary.

The five pins on the left side of the block are allocated to the secondary and the three on the right to the primary participant. The ESM in Figure 4 (b) describes the behavior realized by the block. It is started from the secondary component by a flow through pin *start* which enables this

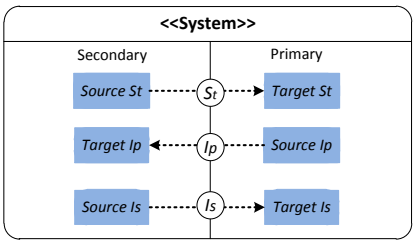

Figure 5: Pattern of a system to be adapted with building block *Mixed Initiative 2*

participant to send its secondary initiative via pin *secI*. Eventually, the start is notified via pin *started* to the primary party who is afterwards enabled to start an own initiative via pin *primI* as long as no secondary initiative passes pin *secA*. If the secondary initiative arrives without being interfered by the primary one, the secondary participant is notified about that via a flow leaving the block through pin *secAc*. If only a primary initiative takes place, the secondary receives it via pin *primA*. If both participants send parallel initiatives via *primI* and *secI*, the secondary is never delivered while the primary one is handed over to the secondary participant via pin *secOv* notifying it that the own one was overridden. The other block in the Arctis library is similar but started from the side of the primary component.

4.2 Adding the Arctis Blocks

It is demanded that the two Arctis blocks introduced above, have to be started using the pins *start* and *started* before they may handle a mixed initiative. Thus, the system needs a crossing edge between the two partitions that may be redirected in order to act as a starter. To find such an edge, we first analyze the subgraph derived by the algorithm in Sect. 3 and check if there is a crossing edge on all traces towards the mixed initiative that can take that role. Thus, in integrating a mixed initiative block, we have to consider three crossing edges as depicted in the pattern description in Fig. 5. This pattern for the Arctis block *Mixed Initiative 2* (and except of the partition designators also of the other available block) consists of a crossing edge called *St* describing the starter while I_s and I_p refer to the crossing edges which may cause the mixed initiative error.

In order to allow an adaptation according to the needs of the system engineer, the transformation process is started by asking the engineer for two decisions depending on which a certain group of AGG graph rewrite rules is selected:

1. The engineer has to determine which component should take the role of the primary party in order to identify which of the two mixed initiative blocks needs to be built in.

2. A principle decision about the strategy to handle detected mixed initiatives has to be taken. This reflects, that the two blocks make the occurrence of a mixed initiative visible to the secondary participant and additional functionality handling this case has to be added. Here, we see two different strategies:

   (a) Delete both signals passing each other after transmission. This strategy is sensible

when the behavior to be performed by a component is the same irrespective of which party triggered the initiative. In this case, we only have to prevent that this behavior is carried out twice.

(b) Let the primary initiative prevail and neglect the impact of the secondary one. This solution is useful if both initiatives lead to a different behavior of the involved components.

According to the two decisions, a particular set of AGG graph rewrite rules is selected which execute the integration of a mixed initiative block automatically. If we decide for the strategy to let the primary initiative prevail, however, we face the problem that we need additional functionality to neglect the secondary initiative in case of a conflict. This, however, depends on the particular model, and it is beyond the capabilities of a graph transformation system to decide if any operation executed before passing I_s should also be executed in case of a conflict. To elude this problem, we selected a graph transformation mechanism that renders a correct solution for most functionalities. Nevertheless, it demands that the system engineer looks on the system model resulting from the graph transformation since it is possible that some of the operations have to be rearranged on the local side of the secondary participant. Thus, the graph transformation does not create the correct solution automatically in all cases but we think that it is nevertheless helpful since it reduces a possible manual post-processing to the purely local reordering of single operations which is much easier than the integration of a complex distributed solution from scratch.

At first, the crossing edges corresponding S_t, I_s and I_p in the pattern model are removed and the Arctis mixed initiative block *mi* is added to the model. The remainder of the graph transformation is the connection of the sources respective targets of the removed edges with the pins of *mi* by new edges. For brevity, we describe this process only for the block *Mixed Initiative 2* listed in Fig. 4 as this procedure is similar for the other block. At first *Source St*, i.e., the source node of edge *St*, will be linked with the pin *start* of *mi* while pin *started* is connected to *Target St*.

On the primary partition, the new wiring of the two conflicting edges I_p and I_s is straightforward since the mixed initiative block disburdens the primary component from any error correction handling. *Source* I_p is coupled with the pin *primI* and pin *secA* with *Target* I_s.

The wiring of the secondary component, however, differs depending on the treatment strategy selected. If both conflicting signals shall be deleted, *Source* I_s is connected with pin *secI* and pin *PrimA* with *Target* I_p. The pins *SecOv* and *SecAc* are not linked at all which according to the robust Arctis semantics means that tokens passing them are deleted.

For the strategy to let the primary initiative prevail, the particular wiring affords to link the pin *primA* with *Target* I_p since this pin reflects that there was no conflict at all. A token passing pin *secOv* contains the data of the primary initiative in the case of a mixed initiative conflict. Since this initiative should prevail, there has also to be made a connection from this pin to *Target* I_p. Actually, the graph transformation rules link both pins *primA* and *secOv* with a newly created merge, i.e., an activity node with at least two ingoing edges but only one outgoing edge to which all incoming tokens are routed. The merge is further connected downstream with *Target* I_p.

The wiring of the vertices and edges specifying the secondary initiative I_s has to consider that its token flow may contain operations necessary for a functionally correct behavior. Operations are another type of UML activities. In Arctis, they are carriers of Java methods which are exe-

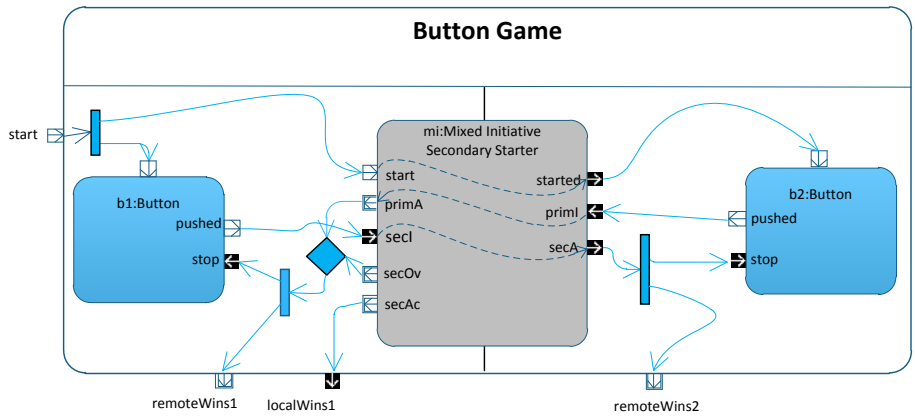

Figure 6: The building block *Button Game* after the transformation

cuted when a token passes. For instance, an operation towards a crossing edge may prepare the transfer format readable by the primary component. After the model modification, such operations shall still be on the path leading to the pin *secI*. Other actions, however, shall only take effect if a conflict does not occur[3] such that they should be linked to the pin *secAc*.

While, as already stated, a general solution to decide about where to place the operations resting on the secondary component before the crossing edge I_s is not possible, we can automate the case that *Source* I_s is a fork node. Here, it is evident that all downstream edges of the fork except for the crossing edge are not relevant for a correct transmission of the secondary initiative. Thus, we can propose a wiring as follows:

1. If *Source* I_s is not a fork, it will be connected with pin *secI* and pin *secAc* will not be linked at all.

2. If *Source* I_s is a fork with two outgoing edges in total, it will be deleted. The source node of its incoming edge will be linked with pin *secI* and pin *secAc* will be connected with the target node of the outgoing edge that is not the crossing edge.

3. If the source node is a fork with three or more outgoing edges, the source node of its incoming edge will also be linked with the pin *secI*. Moreover, we connect the pin *secAc* to the fork such that it is only passed in the case of a successful secondary initiative.

For our button game example, we selected *component 1* as the secondary and *component 2* as the primary partition. Further, we decided to let the primary initiative prevail since, otherwise the pin *remoteWins1* would not be executed in the case of conflict which would violate the ESM of block *Button Game* (see Fig. 1). The result of the graph transformation is depicted in Fig. 6. The modified model produces the state space shown in Fig. 3 (b) such that the ESM of the surrounding block *Button Game* is obeyed. One should mention that the analyzer issues no error

[3] In the button game example, that holds for activity step (5) in Fig. 2 leaving the overall block *Button Game* via the parameter node *localWins1* that should only be triggered if there is no primary initiative at all.

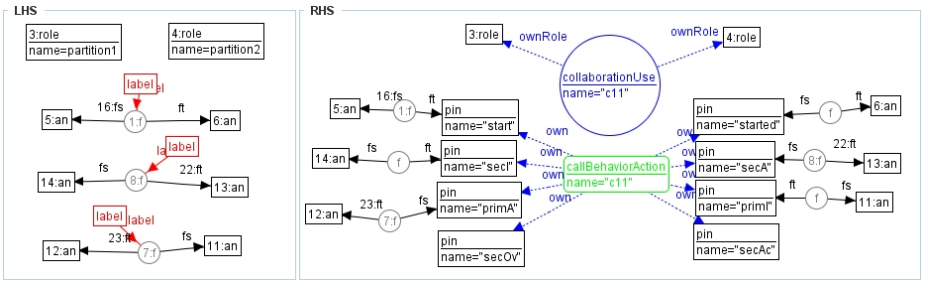

Figure 7: Rule inserting the block *Mixed Initiative 2*

but a warning since in the case of a mixed initiative a flow leads to the pin *stop* of block *b1* which is already terminated. This flaw is of no practical relevance as Arctis simply removes tokens in this case what is exactly what we want. So, we do not need any manual re-orderings.

5 Graph Transformation Rules

In this section, we briefly introduce the concept of graph rewriting rules used for the various mixed initiative detection and remedy steps discussed above. As tool-set for the graph transformation we use the Attributed Graph Grammar System[4] (AGG) [Tae04] that, in a flexible way, allows the visually supported creation of rules. Since AGG offers Java APIs, it could be easily integrated into Arctis that is also Java-based.

The transformation rules mainly consist of two parts. A pre-pattern describes a graph pattern that has to be replaced while the corresponding post-pattern models the result of the replacement. Moreover, a rule may contain additional conditions to constrain when it may be applied. The input to a rule is a so-called host-graph which, by replacing the part matching the pre-pattern by the post-pattern, will be transformed to a post-graph.

Altogether 22 rules are used to detect mixed initiatives and to add one of the two Arctis mixed initiative blocks to a UML activity-based system model. For the sake of brevity, we list only the different categories of rules and provide a closer description of only one rule while the others can be looked at on the WWW.[5] The rules can be structured in three groups:

1. Label the states of an Arctis state graph to detect a mixed initiative as discussed in Sect. 3.
2. Search for the pattern described in Fig. 5 of the state graph to create a subgraph with labels. During this stage the crossing edges I_p, I_s and S_t are marked.
3. Insert the selected Arctis mixed initiative block and wire it with its environment as described in Sect. 4.2.

Figure 7 shows a rule of group 3 that is used to add the block *Mixed Initiative 2* to an activity. On the left side, the pre-pattern is depicted. It contains the classes 3 and 4 of type *role* which refer

[4] We currently replace AGG by Henshin [ABJ+10] which is more flexible and supports the Eclipse Modeling Framework (EMF) also used by Arctis.
[5] http://www.item.ntnu.no/people/personalpages/phd/simon/start

to the two partitions of the activity. These two constructs are applied to enhance the collaboration description used in Arctis to model the relation between blocks and particular components. The other constructs refer to the activities to be amended. In particular, the three labeled cycles refer to the crossing edges involved while label 1:f describes the one to be used to start the block. Label 7:f refers to the primary and 8:f to the secondary initiative. The post-pattern is shown on the right side. The collaboration is extended by a new collaboration use that corresponds to the added mixed initiative block as well as to the links to the two components. The activity is supplemented by a call behavior action, i.e., an Arctis block, as well as references to its pins. The various edges of the pre-pattern are now replaced by others linking the original source respective target nodes with the appropriate pins of the mixed initiative blocks. This rule is used for both strategies mentioned in Sect. 4.2. It renders the final result if we want to delete both mixed initiatives. If we like to let the primary initiative prevail and neglect the secondary one, it creates an intermediate system model which will be further amended by other rules.

6 Related work

In visual language-based specification techniques like the UML, graph grammar techniques are more and more utilized. For instance, in [WTEK08] Winkelmann et al. translate restricted OCL constraints into equivalent graph constraints which enables an automatic generation of instance models from the OCL meta-models. Gronmo and Møller-Pedersen propose so-called aspect activity diagrams that extend activity models by aspect-oriented weaving [GM08]. Likewise, Mussbacher et al. use an extension of the User Requirement Notation (URN) to weave in aspects [MWA10]. A difference to our approach is that both techniques demand for explicit syntax extensions to define aspect orientation concepts like point cuts, which makes the understanding of the models more complicated. In [HHR+11], Hegedüs et al. use graph grammars as the fundamental technique of the framework to generate quick fixes of business flows specified in the Business Process Model and Notation (BPMN). Like our work, this approach uses graph transformation for the remedy of errors, albeit on a more abstract modeling level. Our work is also similar with [LK10] who use graph transformation rules to slice UML models using transformation rules. The difference is that Lano and Kolahdouz-Rahimi concentrate on the slicing of state machines.

Graph grammar systems are further used to support system development in specific domains. Mens et al. [MVDJ05] use graph transformation to formalize refactorings of software. Bucchiarone et al. [BPVR09] give a formal definition of self-adaptiveness and self repair systems based on the T-typed hyper-graph grammar system. They also use AGG to model and verify the hyper-graphs. In [JWEG07], critical pair analysis is used to detect the dependencies and conflicts between features of a Software Product Line (SPL). Domain specific concerns are also addressed by particular patterns. For instance, in [JPW02] security patterns are abstracted from model-based system development and specially treated in security-critical systems.

7 Concluding Remarks

We introduced the use of graph transformation for the detection and remedy of mixed initiative conflicts, a particular kind of development errors in distributed systems. The approach is highly automatic and promises to support the engineer significantly in error recovery and remedy. The set of rules was also applied to a wake-up call scenario [KSH07] as it is used in hotels. Again we received a correct result that did not need further manual corrections.

To generalize this experience, we have also to guarantee that the model transformation does not introduce new errors. In particular, we must assure that the new wiring complies with the properties of the integrated mixed initiative block. Further, we have to prove that the transformed model is consistent with the original one. Of course, as desired, the model transformation changes the system behavior to solve the conflict but it should not be changed in the ordinary case that only one initiative takes place at a time. These two questions can be verified in temporal logic which, however, cannot be shown here due to the space limit.

There are other system layouts which might lead to mixed initiatives, e.g., three or more components may use ring-shaped communication such that there are no conflicting crossing edges between any two of them. We want to find out more system patterns that indicate mixed initiatives and create corresponding graph transformation rules to alleviate them. Further, we intend to use the approach for the detection and remedy of other kinds of errors.

Graph transformation seems also promising to add security protection against malicious attacks. While the integration of counter-measures, in general, is complex and tedious, in some cases it can be done with limited human guidance such that graph transformation is the appropriate means. An example is [GKH11] introducing the structured integration of security mechanisms to protect sensible communications against wiretapping.

The work introduced above was the second approach utilizing graph transformation in SPACE. Before, we used this technique to transform flow-global choreographies, a more abstract modeling technique, to Arctis models [HKL+11]. Both approaches profit from the fact that it is much easier to create and change a set of graph rewrite rules than to develop a model transformation tool doing the same model changes manually. This allows for an easy adaptation of engineering tools for visual languages to new challenges arising during deployment.

A strength of Arctis is that it supports reuse of sub-models in certain application domains. In average, 70% of a system model consist of building blocks reused from previous projects (see [KH09]). Utilizing the flexibility of graph rewriting, one can complement the domain-specific libraries of Arctis blocks with sets of graph transformation rules such that an engineer is not only provided with suitable sub-models but also with a convenient functionality to deal with them.

Acknowledgments The research was carried out under the research and development project "Infrastructure for Integrated Service" (ISIS) funded by the Research Council of Norway.

Bibliography

[ABJ+10] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In Petriu et al. (eds.),

Proceedings of the 13th Int. Conference on Model Driven Engineering, Languages and Systems (Models). LNCS 6394, pp. 121–135. Oslo, 2010.

[BH93] R. Bræk, Ø. Haugen. *Engineering Real Time Systems — An Object Oriented Methodology using SDL*. Prentice Hall, 1993.

[BPVR09] A. Bucchiarone, P. Pelliccione, C. Vattani, O. Runge. Self-Repairing systems modeling and verification using. In *Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*. Pp. 181–190. 2009.

[Flo03] J. Floch. *Towards Plug-and-Play Services: Design and Validation using Roles*. PhD thesis, Department of Telematics, Norwegian University of Science and Technology (NTNU), 2003.

[GKH11] L. Gunawan, F. A. Kraemer, P. Herrmann. A Tool-Supported Method for the Design and Implementation of Secure Distributed Applications. In *Engineering Secure Software and Systems*. LNCS 6542, pp. 142–155. Springer, 2011.

[GM08] R. Grønmo, B. Møller-Pedersen. Aspect Diagrams for UML Activity Models. In Schürr et al. (eds.), *Applications of Graph Transformations with Industrial Relevance*. Pp. 329–344. Springer-Verlag, Berlin, Heidelberg, 2008.

[GY84] M. G. Gouda, Y.-T. Yu. Synthesis of Communicating Finite State Machines with Guaranteed Progress. *IEEE Transactions on Communications* 32(7), July 1984.

[HHR+11] Á. Hegedüs, Á. Horváth, I. Ráth, M. Branco, D. Varró. Quick fix generation for DSMLs. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE Computer, Pittsburgh, PA, USA, 2011.

[HKL+11] F. Han, S. B. Kathayat, H. N. Le, R. Bræk, P. Herrmann. Towards Choreography Model Transformation via Graph Transformation. In *IEEE Conference on Software Engineering and Service Science, ICSESS 2011*. IEEE Computer, Beijing, 2011.

[JPW02] J. Jürjens, G. Popp, G. Wimmel. Towards Using Security Patterns in Model-based System Development. 2002.

[JWEG07] P. K. Jayaraman, J. Whittle, A. M. Elkhodary, H. Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In Engels et al. (eds.), *MoDELS*. LNCS 4735, pp. 151–165. Springer, 2007.

[KH07] F. A. Kraemer, P. Herrmann. Transforming Collaborative Service Specifications into Efficiently Executable State Machines. *ECEASST* 6, 2007.

[KH09] F. A. Kraemer, P. Herrmann. Automated Encapsulation of UML Activities for Incremental Development and Verification. In Schürr and Selic (eds.), *Proceedings of the 12th Int. Conference on Model Driven Engineering, Languages and Systems (Models), Denver, Colorado, USA, October 4-9, 2009*. LNCS 5795, pp. 571–585. Springer-Verlag Berlin Heidelberg, 2009.

[KH10] F. A. Kraemer, P. Herrmann. Formal Techniques for Distributed Systems, Joint 12th IFIP WG 6.1 International Conference, FMOODS 2010 and 30th IFIP WG 6.1 International Conference, FORTE 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings. In Hatcliff and Zucca (eds.), *Formal Techniques for Distributed Systems*. LNCS 6117. Springer, 2010.

[Kra09] F. A. Kraemer. Automatic Generation of Compatible Interfaces from Partitioned UML Activities. In Reed et al. (eds.), *SDL 2009: Design for Motes and Mobiles*. LNCS 5719, pp. 182–199. Springer Berlin / Heidelberg, 2009.

[Kra11] F. A. Kraemer. Engineering Android Applications Based on UML Activities. In Whittle et al. (eds.), *Model Driven Engineering Languages and Systems*. LNCS 6981, pp. 183–197. Springer Berlin / Heidelberg, 2011.

[KSH07] F. A. Kraemer, V. Slåtten, P. Herrmann. Engineering Support for UML Activities by Automated Model-Checking — An Example. In *4th International Workshop on Rapid Integration of Software Engineering Techniques (RISE)*. 2007.

[KSH09] F. A. Kraemer, V. Slåtten, P. Herrmann. Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services. *Journal of Systems and Software* 82(12):2068–2080, 2009.

[LK10] K. Lano, S. Kolahdouz-Rahimi. Slicing of UML Models Using Model Transformations. In Petriu et al. (eds.), *Model Driven Engineering Languages and Systems*. LNCS 6395, pp. 228–242. Springer Berlin / Heidelberg, 2010.

[MVDJ05] T. Mens, N. Van Eetvelde, S. Demeyer, D. Janssens. Formalizing refactorings with graph transformations: Research Articles. *J. Softw. Maint. Evol.* 17:247–276, 2005.

[MWA10] G. Mussbacher, J. Whittle, D. Amyot. Modeling and detecting semantic-based interactions in aspect-oriented scenarios. *Requirements Engineering* 15:197–214, 2010.

[Obj10] Object Management Group. Unified Modeling Language: Superstructure, Version 2.3. 2010.

[SDW08] C. Shin, A. K. Dey, W. Woo. Mixed-initiative conflict resolution for context-aware applications. In *UbiComp'08*. Pp. 262–271. 2008.

[Tae04] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In Pfaltz et al. (eds.), *Applications of Graph Transformations with Industrial Relevance*. LNCS 3062, pp. 446–453. Springer, 2004.

[WTEK08] J. Winkelmann, G. Taentzer, K. Ehrig, J. M. Küster. Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars. *Electronic Notes on Theoretic Computer Science* 211:159–170, 2008.