Towards an Aspect-oriented Language Module: Aspects for Petri Nets

Tim Molderez * Bart Meyers Dirk Janssens Hans Vangheluwe

Dept. of Mathematics and Computer Science
University of Antwerp
Antwerp, Belgium
{tim.molderez,bart.meyers,dirk.janssens,hans.vangheluwe}@ua.ac.be

Abstract

The concept of composing a (domain-specific) language from different reusable modules has gained much interest over the years. The addition of aspect-oriented features to a language is a suitable candidate of such a module. However, rather than directly attempting to design an aspect-oriented language module that is applicable to any base language, this paper focuses on adding aspect-oriented features to a language that is quite different from prevalent base languages (e.g. Java): Petri nets. A running example demonstrates the use of aspects to enforce an invariant on a base Petri net.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; I.6.5 [*Simulation and Modelling*]: Model Development

General Terms Petri nets, language engineering, aspect-oriented modelling, language composition

1. Introduction

Creating a new (domain-specific) language, including the tools, documentation and community that accompany a language, is an undeniably large endeavour. To reduce this effort, there is a growing interest in constructing new languages by extending existing languages, or by composing different languages in a modular manner [3, 5, 8, 9]. . Next to this research area, there also is a large body of work that present aspect-oriented versions of various different languages. The combination of these two research areas leads to the notion of an "aspect-oriented language module". In other words, a module that implements (pointcut-and-advice flavoured) aspect-oriented features, and is applicable to a wide range of base languages, if not any[1] base language. However, rather than attempting to cover all base languages at once, this paper focuses on adding aspect-oriented features to just one language: Petri nets. We chose the Petri nets modelling language as our base language, simply because it is so different from commonly used base languages (e.g. Java): Petri nets are non-deterministic and their state is implicit. In this paper, we are mainly interested in how this choice of base language affects the design of aspect-oriented features. Consequently, we present an aspect-oriented extension to Petri nets and demonstrate its use with an invariant enforcement example. While designing this aspect-oriented Petri nets extension, we keep the general idea of an aspect-oriented language module in mind. This is done by dividing the extension itself into a number of components, such that the extension is no longer specific to Petri nets at this level of abstraction.

The remainder of this paper is structured as follows: Sec. 2 introduces the paper's running example. Sec. 3 then elaborates on this example while the various components of the aspect-oriented extension to Petri nets are discussed. An alternative view on the example's semantics, where it is mapped to a Petri net with guarded transitions, is briefly examined in Sec. 4. Sec. 5 presents related work and Sec. 6 concludes the paper.

2. Invariant enforcement example

We will use a running example throughout this paper to demonstrate the use of our aspect-oriented Petri net extension. The "base Petri net", i.e. the Petri net without any aspects, for this example is shown in Fig. 1. The places within the dotted rectangle (`P5-P8`) represent a number of rooms within a building. All places outside this rectangle (`P1-P4`)

* Funded by a doctoral scholarship of the Research Foundation - Flanders (FWO)

[1] If the notion of a join point is extended to include static points in a model, base languages can be made aspect-oriented even if they do not describe behaviour.

DSAL '12 March 27, 2012, Potsdam, Germany.
Copyright © 2012 ACM 978-1-4503-1128-1/12/03. . . $10.00

represent the outside world. The transitions going in and out of the rectangle represent the building's various entrances (`T1`,`T3`) and exits (`T2`,`T4`). The goal of this example is to enforce an invariant: Only a certain number of people (tokens) may be inside the building at the same time. If more people were to enter the building, this could, for example, violate the building's fire safety regulations. To prevent this from happening, the following sections will use aspects to monitor the building's entrances and exits. While the Petri net shown in the figure is quite small, the same aspects also apply to more complex cases, where the building consists of many more rooms, entrances and exits. The only assumption that we make is that the places and transitions inside the building do not produce any additional tokens, nor remove any tokens.

The aspects enforcing the invariant are shown in Fig. 2 and Fig. 3. The first aspect, called `VisitorCounter`, keeps track of how many people are currently inside the building. It contains two pointcut-advice pairs: The pair called `increaseOnEnter` increases the `Counter` place whenever someone passes through an entrance. That is, its pointcut captures all occurrences of transitions annotated with an `«enter»` stereotype. While our pointcuts make use of these stereotypes, note that stereotypes are not strictly necessary. We could have also explicitly listed the names of the transitions that should be matched by the pointcut. However, the use of stereotypes makes the pointcuts easier to understand and less dependent on concrete transition names, therefore reducing the impact of the fragile pointcut problem. This is based on Noguera et al. [7], which uses Java annotations within AspectJ pointcuts.

As expected, the `decreaseOnExit` pointcut-advice pair will decrease the `Counter` place whenever someone leaves the building, i.e. an `«exit»` transition is fired.

The second aspect, `WaitingLine`, will enforce the invariant that only a certain number of people, say 30, may be in the building at the same time. This aspect is dependent on the `VisitorCounter` aspect, as it needs access to its `Counter` place. The `moveToWaitingLine` pointcut-advice pair acts whenever the building is full and someone tries to enter. That person will then be redirected to a waiting line, represented by the `WaitHere` place. There is such a waiting line for each entrance. If someone in the waiting line tries to enter the building while it still is full, the `preventEntrance` pair will prevent that person from entering. To make sure that the `Counter` place is only increased if the building is not full, the `declare precedence` statement expresses that the pointcuts in the `WaitingLine` aspect have precedence over those in `VisitorCounter`. We will elaborate on the semantics of these aspects over the course of the following sections.

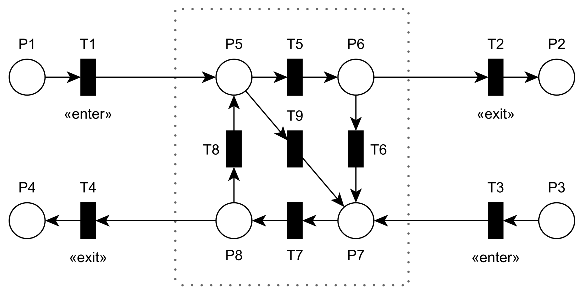

Figure 1: Base Petri net

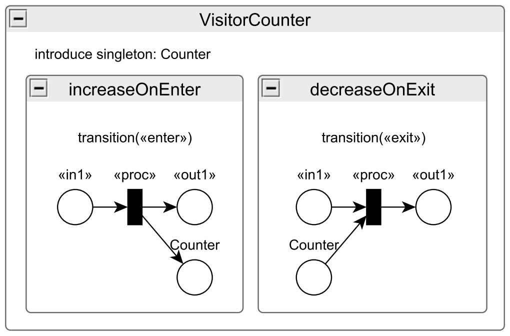

Figure 2: `VisitorCounter` aspect

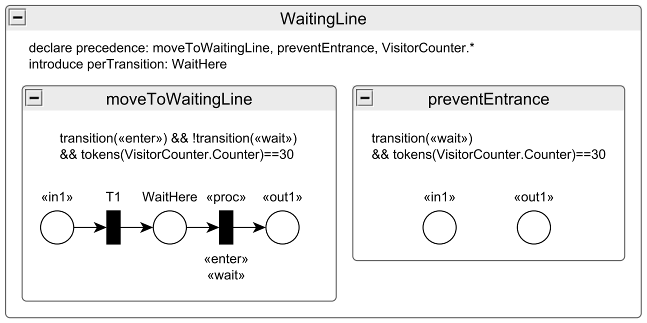
WaitingLine
declare precedence: moveToWaitingLine, preventEntrance, VisitorCounter.*
introduce perTransition: WaitHere
moveToWaitingLine
transition(«enter») && !transition(«wait»)
&& tokens(VisitorCounter.Counter)==30
«in1»
T1
WaitHere
«proc»
«out1»
«enter»
«wait»
preventEntrance
transition(«wait»)
&& tokens(VisitorCounter.Counter)==30
«in1»
«out1»

Figure 3: `WaitingLine` aspect

3. Overview of the aspect-oriented extension

To establish aspect-oriented Petri nets as an extension to Petri nets, there are a number of different components to be determined: the base language (Petri nets), a weaver, join point model, pointcut language, advice language and a composition mechanism. Note that, at this level, none of these components are specific to Petri nets; they may as well be applied to another base language. The graph in Fig. 4 shows the dependencies between the different components at a high level.

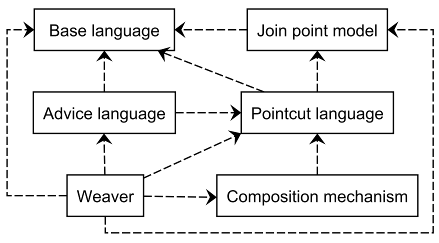

Figure 4: Dependencies between the language components

3.1 Weaver

The core component of the extension is the weaver, which manages most communication between the other components. It roughly performs the following steps:

1. During execution, whenever a join point (as determined by the join point model) is reached, normal Petri net execution is paused and control is passed to the weaver.
2. Given the current state and the current join point, the weaver tests all pointcuts to find the ones that match.
3. Given the pointcuts that match, the composition mechanism determines the right order in which the corresponding advice should be inserted.
4. Each advice now is inserted into the basic Petri net to create an extended Petri net. This is done in the order determined by the composition mechanism.
5. Petri net execution resumes.

These steps are clearly not aimed at performance, as all advice insertion and pointcut checking is done as late as possible. Keeping the general-purpose aspect-oriented language module in mind, this weaver aims to be simple and allow the other components to be as expressive as needed. Mainly depending on the pointcut language's available constructs, the weaver can be optimized for performance, such that weaving can happen earlier while the semantics of aspects remain the same. Additionally, the above steps can be easily adjusted to another base language: We just need to be able to pause execution at each join point to hand control over to the weaver, which then modifies the model/program in execution according to its aspects and finally resumes execution.

3.2 Joinpoint model

In general, any point in time during a model/program execution could be considered an element of a language's join point model. Because it is possible to generate execution traces for any language that defines behaviour, we are confident that a meaningful join point model can be defined for any such language as well. In case of Petri nets, we have chosen our join point model to consist of all run-time occurrences of transitions. That is, advice can be executed whenever a transition is *about to* fire. We have chosen this particular join point model, because it is not affected by the nondeterministic nature of Petri nets: Whenever a Petri net execution arrives at a particular join point, it has not just determined which transitions *can* be fired (i.e. which transitions are enabled); it has also chosen which transition(s) *will* be fired. Therefore, it is clear which behaviour will be extended or replaced whenever an advice is applied.

3.3 Pointcut language

After choosing a join point model, we can create the language used to describe pointcuts. In general, a pointcut describes a set of join points. Alternatively, a pointcut can also be seen as a boolean function in terms of a join point, and any other contextual information. If this function is true for a particular join point, the pointcut matches. The pointcut language that we have chosen for aspect-oriented Petri nets is minimal: Our pointcuts are boolean expressions that can make use of two constructs: `transition` and `tokens`. An example of the `transition` construct is shown in Fig. 2: `transition(«enter»)`. This construct is true if the current join point, being an occurrence of a transition, corresponds to one of the parameters in the `transition` construct. In this example, the parameter indicates all transitions with the `«enter»` stereotype. It is of course also possible to pass in concrete transition names as parameters, or to make use of wildcards. Note that the transition construct does not make use of a join point's dynamic nature, as the mapping to join point shadows, i.e. transitions (not *occurrences of* transitions), is trivial. Pointcuts do get more interesting when combined with the `tokens` construct. This construct is used to reason about the amount of tokens in a certain place. We have used it in the pointcuts of the `WaitingLine` aspect in Fig. 3. For example, `tokens(VisitorCounter.Counter)==30` is true if there are 30 tokens in the `Counter` place of the `VisitorCounter` aspect.

3.4 Advice language

If a pointcut matches in a pointcut-advice pair, the corresponding advice should be inserted into the base Petri net, resulting in an augmented Petri net. In our extension, an advice looks like a regular Petri net, but is extended with a few different constructs. First, there are special stereotypes for places that are used to specify how the input and output places of a join point should be bound to an advice. This is similar to the notion of ports in CPN tools' [1] hierarchical Petri nets. These bindings are discussed in Sec. 3.4.1. Second, transitions can also carry a `«proc»` stereotype, indicating a proceed transition, similar in concept to proceed calls. Proceed transitions are covered in Sec. 3.4.2. Finally, places can be shared among advice by means of introductions (also known as inter-type declarations in AspectJ), introduced in Sec. 3.4.3.

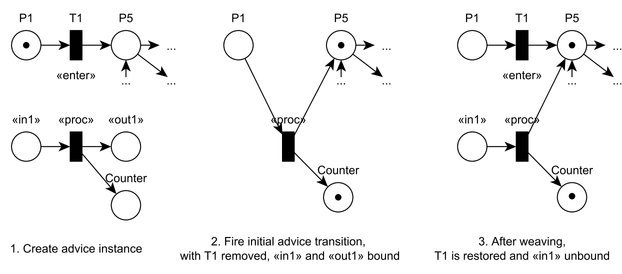
P1
T1
P5
«enter»
«in1»
«proc»
«out1»
Counter
1. Create advice instance
2. Fire initial advice transition, with T1 removed, «in1» and «out1» bound
3. After weaving, T1 is restored and «in1» unbound

Figure 5: Weaving an advice into the base Petri net

3.4.1 Input and output place binding

Looking at the advice of `increaseOnEnter` in Fig. 2, the `«in1»` and `«out1»` stereotypes instruct how this advice must be bound to the input and output places of a transition. In this case, the `transition(«enter»)` pointcut always matches whenever a transition with an stereotype `«enter»` is about to be fired. Note that these `«enter»` transitions always have one input and one output place. When applying our advice, the effect we want to achieve is that we want to intercept the firing of an `«enter»` transition and replace it with an advice execution. To be able to do this, we need to bind the intercepted `«enter»` transition's input place to the advice's `«in1»` place. Similarly, the output place should be bound to the `«out1»` place. Note that the number of input and output places of the intercepted transition must match with the `«inN»` and `«outN»` stereotypes in the advice. For example, if the `«enter»` transition in the base net would have two input places, our advice must also have an `«in2»` place. Otherwise, the advice cannot be applied, which implies that the pointcut does not match. In this sense, the `«inN»` and `«outN»` stereotypes also form a part of the pointcut language. Additionally, if an advice can be applied and there are multiple input or output places to be bound, the choice of which input/output place must bind to which `«inN»` / `«outN»` place is currently left non-deterministic. This could be resolved quite easily by extending the pointcut language with constraints on input/output places, such that they can be distinguished if needed. However, we won't go into any further details as our example is not confronted with such situations; all of its transitions have one input and one output place.

After introducing the `«inN»` and `«outN»` stereotypes , inserting an advice into a base Petri net is performed as follows: Suppose that `increaseOnEnter`'s pointcut just matched on transition `T1` in the base Petri net and we wish to insert the corresponding advice. We will also assume that only one advice matched on this join point, in which case the proceed transition (with a `«proc»` stereotype) simply acts as a normal transition. (Shared join points are discussed in Sec. 3.4.2.)

To insert `increaseOnEnter`'s advice, a new instance of the advice is created and added to the base Petri net. This is shown in step 1 of Fig. 5. (Only the relevant part of the base Petri net is shown.) This advice instance is created such that the `«out1»` place is bound to the join point's output place, `P5`. Note that this binding is permanent. The input place's binding only holds for an instant however: The effect that we want to achieve is, instead of firing `T1`, we want to fire the transition attached to the advice's `«in1»` place. To accomplish this effect, we temporarily remove `T1` and bind the input place, i.e. `«in1»` is bound to `P1`. This only lasts until we have fired. This situation is shown as step 2 in Fig. 5. Once this is done, `T1` is added to the net again and `«in1»` is no longer bound and, as of now, is nothing but a regular (empty) place. At this point, we have a normal Petri net once more, meaning that advice insertion has completed and regular Petri net execution can continue. This resulting Petri net is shown as step 3 in Fig. 5.

The reason that the input place's binding is removed as soon as the advice initially fired, is because our advice insertion should only intercept one single join point, i.e. one occurence of a transition. If the binding remained, the advice would of course interfere with subsequent occurences of that transition. On the other hand, the output place binding is permanent because we wish to preserve non-determinism: We do not enforce the advice to finish its execution (i.e. to continue firing transitions in the advice as long as possible) before execution of the base Petri net can resume. As the resulting Petri net is a normal Petri net, it is allowed that transitions from the base net are fired even though the advice has not finished its execution yet. To allow the result of the advice execution to be passed to the base net, binding of output places is permanent.

Related to this discussion is the reason why we create a new advice instance for *every* matching joinpoint. This design choice was made as well to preserve Petri nets' non-deterministic nature. Because it is possible that the `«inN»` and `«outN»` bindings are different for two join points, even if the two join points are different occurrences of the same transition, the same instance may not be reusable across different join points. An example of such a situation can occur due to the advice composition changing when that transition is fired again later. While we could allow for instance reuse, this again requires that we sacrifice non-determinism and enforce that advice transitions have a higher priority than all other transitions. We did not choose this path, as the result of advice insertion would no longer be a simple Petri net, but a prioritised Petri net, which would mean that the base language is being modified. The main disadvantage of creating a new instance for each matching join point is of course that the augmented Petri net easily grows in size. However, this can be heavily optimized: For example, if `«outN»` places only have incoming edges, then an advice instance can be safely removed as soon as no more enabled transitions are found within the instance. Such an instance can only produce output to the base net, and if no transitions are enabled, nothing can be produced and therefore the instance can be removed without altering the augmented Petri net's behaviour.

3.4.2 Proceed transitions

As mentioned earlier, a transition with a «proc» stereotype indicates a proceed transition. Similar to AspectJ, an advice usually contains one proceed transition, but it may contain none or multiple of them as well. If there are multiple advice at the same join point, the proceed transitions indicate where the next advice in the aspect composition must be woven. In order to make this possible a proceed transition must also be compatible with the number of input and output places of the join point that was intercepted. For example, if a pointcut captures transitions with one input and one output place, a proceed transition must (at least) have one input and one output place as well. (In case ambiguities arise, this can be resolved by adding stereotypes to map the input and output places of the proceed transition.) Our invariant enforcement example also includes an instance where join points can be shared by multiple pointcuts: Once the building is at full capacity, i.e. `Counter` contains 30 tokens, then the pointcuts of `moveToWaitingLine`, `preventEntrance` and `increaseOnEnter` will match. As determined by the composition mechanism, the advice of `moveToWaitingLine` is inserted first. Immediately after this insertion, the `preventEntrance` advice is inserted into the proceed transition of `moveToWaitingLine`. (The insertion process for proceed transitions is similar, except that both «inN» and «outN» bindings are permanent.) If the `preventEntrance` advice would contain a proceed transition, `increaseOnEnter` would be inserted. The `preventEntrance` advice however consciously does not contain a proceed transition, as its purpose is to prevent entrance to the building and to prevent the number of tokens in the `Counter` from increasing.

3.4.3 Introductions

To be able to share information between different advice instances, introductions can be used. This is used in the `VisitorCounter` aspect in Fig. 2 to make the `Counter` place global. The `introduce singleton: Counter` statement in the aspect declares that there only is one instance of `Counter`, shared among all advice instances in `VisitorCounter`. That is, if there is a place named `Counter` in the advice, that place will be bound to the actual `Counter` instance whenever an advice instance needs to be inserted. If there are multiple instances of `increaseOnEnter` or `decreaseOnExit`, the `Counter` place will also get several incoming and outgoing edges.

The `WaitingLine` aspect in Fig. 3 makes use of introductions as well. Rather than introducing a global place, it introduces `WaitHere` places with the `introduce perTransition` statement. This statement declares that a `WaitHere` place will be created for each transition corresponding to the join points that matched. In other words, all advice that matched on the same join point shadow will share the same `WaitHere` place. In the context of our example, a `WaitHere` place is thus created for all entrances to the building, as `moveToWaitingLine`'s pointcut looks for «enter» transitions.

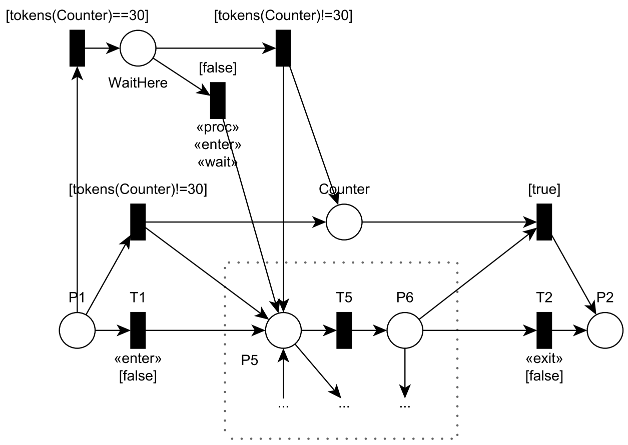

Figure 6: All aspects woven into the base Petri net

3.5 Composition mechanism

The final component of the aspect-oriented Petri net extension is the composition mechanism. The mechanism that we use works quite similar to AspectJ's, by means of a `declare precedence` statement, as shown in the `WaitingLine` aspect of Fig. 3. It is however slightly more fine-grained, as the composition order is defined at the level of pointcuts instead of aspects. This level of granularity is needed for our example, as people should be redirected to a `WaitHere` place before preventing access to the building, in case it is at full capacity. In other words, `moveToWaitingLine` should always be executed before `preventEntrance`. Additionally, these two must be executed before `increaseOnEnter` as well; otherwise the `Counter` place could increase even if this is not allowed.

4. Mapping to Petri nets with guarded transitions

An alternate view on the aspect-oriented Petri net semantics is shown Fig. 6. This view is more suited to understand what our entire example may look like if all advice already are woven into the base Petri net before execution, as a preprocessing step. This results in one large Petri net that is, for the most part, a normal Petri net, but is extended with residual logic. This is similar to an AspectJ program being compiled to a Java program where all advice are inlined together with some residual logic to be able to determine at run-time whether an aspect should be applied at a particular join point shadow or not. In Fig. 6, the residual logic is reflected in certain transitions having guard expressions. For example, consider the `VisitorCounter.decreaseOnExit` advice, which has been applied to the `T2` `«exit»` transition. Because this advice *always* applies to all exits, `T2` has a `[false]` guard expression, such that it can never fire. Instead, the advice's transition (has the `Counter` place as input) gets a `[true]` expression, such that the advice is always executed instead of `T2`.

Applying advice to the entrances is a little more involved, because there are multiple advice on these join point shadows. Because there always is at least one advice applied to entrances, the `T1` transition is disabled with a `[false]` guard. In case the building is not full, meaning that the `Counter` place does not contain 30 tokens, then only the `VisitorCounter.increaseOnEnter` advice should be executed. In case there are 30 tokens in `Counter`, the `[tokens(Counter)==30]` guarded expression is true and the `WaitingLine.moveToWaitingLine` advice is executed. Within this advice, the proceed transition is refined into three options. If no other advice share join points with the `moveToWaitingLine` advice, the proceed transition is just a normal transition. In our example, this never occurs because `increaseOnEnter` always applies, hence the `[false]` guard. If the building is full, `preventEntrance` blocks the entrance. This is indicated by the lack of a transition for this case. In case there is room, the `[tokens(Counter)!=30]` guard is true and executes `increaseOnEnter`.

5. Related work

In terms of related work, the closest to our aspect-oriented Petri nets extension is the work presented in Xu et al. [10], where aspects are used in Petri nets to implement threat mitigations in security design. The join point model of their aspect-oriented extension however is a static one, consisting of places and transitions rather than transition occurences. Such a join point model is arguably not entirely aspect-oriented, as join points are commonly considered as dynamic points in aspect-oriented languages. As a result, the weaving process is implemented as a preprocessing step without any residual logic. While easier to understand, our interests lie more in exploring how characteristic aspect-oriented constructs carry over to different base languages. The feature-oriented Petri nets extension in Muschevechi et al. [6] is used to model software product lines. Its extension to Petri nets adds transitions that are guarded (also called application conditions) by which features are selected in a software product line. The approach is only tested for small examples, so it is currently unclear how the approach scales to larger systems, especially when crosscutting features are involved. It may be worthwhile to use aspect-oriented techniques to support featured-oriented concepts, as this has been done before at the level of programming languages in Apel et al. [2]. Finally, there also is a close connection between our work and hierarchical Petri nets [4], as the use of aspects automatically introduces a notion of modularity and reuse. Our aspect-oriented Petri nets compare to hierarchical Petri nets in a similar manner as AspectJ's advice executions compare to Java method calls. Whereas in hierarchical Petri nets it is indicated explicitly which transitions will be substituted for a subnet, this becomes implicit for aspect-oriented Petri nets, as it implements the inversion of control principle.

Whereas a Java method call is explicit, an advice is executed implicitly, i.e. it implements the inversion of control principle.

The main difference of course is that aspects implement inversion of control, whereas hierarchical Petri nets are explicit. Additionally, introductions make it possible to share information between aspects, and with the base net.

6. Conclusion and future work

This paper has presented an aspect-oriented extension to Petri nets, as an initial step to an aspect-oriented language module. The Petri nets language was chosen because it distinguished itself from a common base language through its non-determinism and implicit state. The running example has shown an interesting use for aspect-oriented Petri nets: Aspects can be used to enforce invariants in a compact and modular manner. The resulting Petri net may then be used for further analysis. One interesting direction of future work however is to study how our addition of aspects affects its use for analysis. For example, if a base Petri net is shown to be free of deadlocks, what happens if an aspect is added? Which characteristics should this aspect have in order to preserve this desired property? Additionally, because Petri nets are such a small language compared to the average programming language, Aspect-oriented Petri nets may also be an interesting use case to study the interactions among aspects, and between aspects and the base system. Another direction of future work is more towards the general-purpose aspect-oriented language module: A variety of other types of base languages can be extended with aspect-oriented features in order to form a more precise idea of what it means for a language to be aspect-oriented, and what this language module may look like. It may as well be interesting to weaken the notion of a join point to include static locations in models, such that even languages that do not define behaviour can have an aspect-oriented extension.

References

[1] CPN tools. http://cpntools.org.

[2] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: on the symbiosis of Feature-Oriented and Aspect-Oriented programming. In *Generative Programming and Component Engineering*, volume 3676, pages 125–140. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[3] S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2), 2004.

[4] P. Huber, K. Jensen, and R. Shapiro. Hierarchies in coloured petri nets. *Advances in Petri Nets 1990*, page 313–341, 1991.

[5] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: modular development of textual domain specific languages. In Richard F. Paige and Bertrand Meyer, editors, *Objects, Components, Models and Patterns*, volume 11, pages 297–315. Springer Berlin Heidelberg, Berlin, Heidelberg.

[6] R. Muschevici, D. Clarke, and J. Proenca. Feature petri nets. In *Proceedings of the 14th International Software Product Line Conference (SPLC 2010)*, volume 2, 2010.

[7] Carlos Noguera, Andy Kellens, Dirk Deridder, and Theo D'Hondt. Tackling pointcut fragility with dynamic annotations. In *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*, RAM-SE '10, page 1:1–1:6, New York, NY, USA, 2010. ACM.

[8] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, page 451–464, New York, NY, USA, 2006. ACM.

[9] Markus Völter and Eelco Visser. Language extension and composition with language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, page 301–304, New York, NY, USA, 2010. ACM.

[10] D. Xu and K. E Nygard. Threat-driven modeling and verification of secure software using aspect-oriented petri nets. *IEEE Transactions on Software Engineering*, 32(4):265– 278, April 2006.