An approach using formally well-founded domain languages for secure coarse-grained IT system modelling in a real-world banking scenario

Christoph Brandt, Thomas Engel

Computer Science and Communications Research Unit
Université du Luxembourg
E-Mail: {christoph.brandt,thomas.engel}@uni.lu

Benjamin Braatz, Frank Hermann, Hartmut Ehrig

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
E-Mail: {bbraatz,frank, ehrig}@cs.tu-berlin.de

Abstract

In this paper we show how distributed coarse-grained IT systems in a real-world banking scenario can be modeled using domain concepts and languages that are standing on top of formal methods. We further show how these methods help to enforce structural security requirements, like firewall placements. In contrast to todays diagrams of IT landscapes, this approach makes use of the full power of formal methods, being at the same time completely transparent to the people using it in the scenario. This is what makes this theoretical approach applicable in a real-world environment where people are highly sensitive to set-up costs and any daily operational overhead.

Keywords

Exogenous component coordination, security, algebraic graph transformation, meta-modelling, domain-specific language, domain concepts

Introduction

The problem addressed in this paper is the question of how to handle structural security issues, like firewall placements, of distributed coarse-grained IT components in a decentralised and global banking organisation. The concrete case that is used as a representative example is the real-world situation of the Credit Suisse Luxembourg, S.A. This case is referred to as "the scenario" in the following.

The scenario encompasses coarse-grain IT components like databases, applications for portfolio management or order processing. It is run by highly skilled technical staff. The average workload is high, and the expertise is well focused. Security issues come in as something holistic and cross-functional, driven by internal and external stakeholders, limited by given restrictions. Internal stakeholders are, for example, local staff or headquarter people enforcing their security policies; external stakeholders show up as, for example, government requirements, like certain public security standards that must be implemented. Limitations naturally arise because security is cross-functional to actual competences, skills and experiences. So, for example, mastering database security and application security for its own does not necessarily result in a secure distributed system, and having security competences in both fields is not realistic for a single expert. It is therefore important to find a solution to help people in the scenario to model, build, administrate, monitor and control such a local IT landscape respecting given restrictions in terms of security set-up time, daily operational workload and available competences. In this sense, the requirements for the solution are derived from the scenario, and the solution itself must prove to be helpful in the scenario later on.

First, from a methodological point of view, the solution proposed in this paper shows how to abstract from the scenario using a mathematical model. It further shows how to handle the building blocks of such a model using a mathematical method. Therefore, the models as well as the modelling process are both based on mathematics. While the coordination model Reo and Abstract Behavior Types (ABTs) are used to model the distributed system, algebraic graph transformation is used as technique for generating and editing these models. Second, from a practical point of view, ways to hide the mathematical issues from the people in the scenario are presented. This is done to reduce the set-up cost before being able to apply this approach, as well as to minimise the daily operational overhead after implementing the solution in the scenario later on.

Because the internal implementation of components in the scenario is not known, the focus has to be on message streams between them. The chosen techniques Reo and ABTs fulfil this requirement, since they allow the modelling of exogenous coordination of components by connectors. Because component and connector models of distributed systems can be interpreted as graphs, algebraic graph transformation applies ideally for specifying complex editing operations controlled by security restrictions. Its mathematical foundation allows a systematic and even automatic analysis of possible states in the modelling process, but it also facilitates inherent safeness of editing operations, as far as desired.

In this paper, we first reflect the current industrial practice of how to address the issue. Then we introduce Reo and Abstract Behavior Types as the formal building blocks during system modelling and algebraic graph transformation as a technique to create system models at the concept level that are in line with structural security requirements. We show further how the formal and the concept layer are related and how the first one can be derived from the second.

Current industrial practice

Before presenting our approach, an overview of the current industrial practice is given. This is done to motivate the concrete need for a better solution. The presented example shows one of the most appropriate available solutions for domain-specific modelling, which is not representative for the use in real-world situations, where most often only informal drawings, natural language and spreadsheets are used to model IT infrastructures and component coordination. But even this state-of-the-art approach clearly demonstrates the need for theoretical solutions to be developed, so that practical questions can be treated in a more satisfying way.

In order to model, administrate, monitor and control structural security, like firewall placements, of a distributed system, respecting cost and time restrictions as well as competence profiles of people available in the scenario, domain-specific modelling is applied, yielding higher abstraction by hiding implementation-specific details to lower levels and, thus, giving people the freedom to focus on domain-specific issues only. The domain level in the scenario is clearly the handling of coarse-grained IT components and their connectors. The underlying assumption is that glue code to connect these components can be generated based on high-level domain-specific specifications. In the best case, the implementation level does not have to be touched by the infrastructure experts in the scenario at all.

Seeking efficiency, industry is primarily looking for tools supporting the ongoing work, seldom for pure methods. So, computer-aided software engineering (CASE) tools, better meta-CASE tools, are the appropriate choice here. Their specific benefit is the separation of concerns: the meta-CASE tool provider is the expert at building CASE tool functionality, the people in the scenario are the experts of their domain. This division of labour is missing from CASE tools for fixed modelling languages such as UML and from development frameworks for (domain-specific) visual language tools. In the first case, either the provider should be an expert of the domain addressed or the domain expert has to encode his terminology into the notions used by the modelling language. In the second case, the people in the scenario must become experts in building CASE tools. Both is not realistic to be assumed.

For demonstration purposes MetaEdit+ (Tolvanen 2005, Luoma et al. 2004) is taken to be the meta-CASE tool used here. MetaEdit+ is widely accepted and proved its utility in industry. By the help of MetaEdit+ concepts of the domain can be defined. Their visual representation is used further on to model a distributed system like the one in the scenario. Rules and restrictions of how to combine these visually represented concepts are governing structural security issues. Later on, scripts generate the needed glue code. The primary benefit is the one of generation, which cuts costs and reduces set-up as well as maintenance time. The secondary benefit is the one of consistently applying the defined rules and restrictions during the modelling process, as well as the scripts generating the glue code. This yields enforced structural security rules, and constant quality, seen from a technical point of view, in the final code.

However, even practical, this is of little value regarding the structural security issues in the scenario. The main reason for that is a missing mathematical model, which describes the handling of syntax and semantics of the used visual and non-visual languages, as well as their interplay. So, no proofs can be given. For example, the domain concepts are not suitable for model checking. They lack a mathematical foundation. Conflicts between security rules cannot be treated mathematically, because they are not defined using any mathematical model.

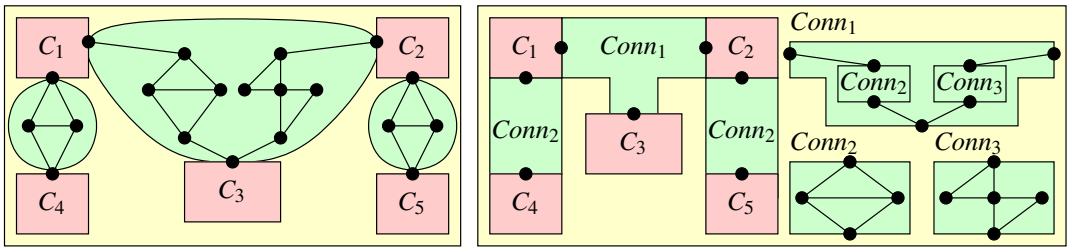
The presented approach in this paper addresses these issues. It shows how to save the benefit of domain-specific modelling using meta-CASE tools with all mentioned advantages like cost cutting and time savings and demonstrates how to apply abstract algebra to build a foundation for proofs that are needed to yield reliable security statements later on.

Coordination model

In this paper we are using Reo (Arbab 2004) as the underlying coordination model, on which the domain-specific business language resides. Reo emphasises the exogeneous coordination of components by glue code and thereby directly supports our requirement of treating them as "black boxes". In contrast to functional, imperative and object-oriented models, e. g. Abstract Data Types, Reo does not rely on interface signatures or synchronous operation calls, but only on message streams. This definition, however, does not prevent the implementation of, for example, interface signatures and several kinds of synchronisation mechanisms on top of the message streams.

Components, connectors and channels

In Reo a system consists of *components*, which are joint using *connectors*. These connectors are themselves constructed out of simpler connectors and *channels*. Channels can be seen as the simplest, atomic kind of connectors. Connectors are represented as graphs, where *edges* correspond to channels and their *channel ends* coincide on the *nodes*. Each channel has exactly two channel ends, each channel end coincides on exactly one node and an arbitrary number of channel ends may coincide on each node. For each channel end x we denote by $\hat{x}$ the unique node on which the channel end coincides. For each node N we denote by $[N]$ the set of channel ends coincident on N. Thus, for all nodes N and channel ends $x \in [N]$ we have $\hat{x} = N$ and for all channel ends x we have $x \in [\hat{x}]$. Figure 1(a) shows an example, where five components $C_1, \ldots, C_5$ are connected using various connectors. In Figure 1(b) we see how connectors may be reused in systems and other connectors by referencing them in the graph. The system has exactly the same topology as the flat system in Figure 1(a), but the presentation with connector references and reuses leads to a more concise arrangement.



(a) Flat system (b) System with connector references

Figure 1: Examples of Reo systems

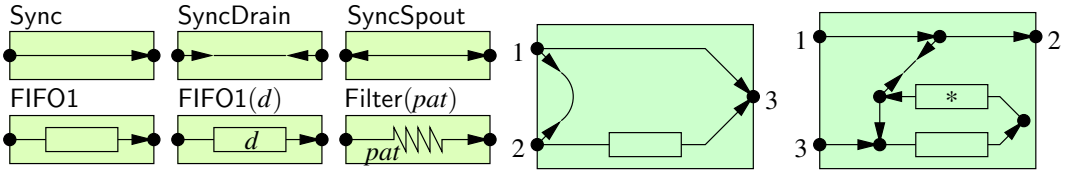
Channels do not have a direction themselves in Reo, but their ends do. A *source* end accepts data into the channel and a *sink* end promotes data out of the channel. Channels may have ends with different or identical kinds. A channel with two source ends is called *drain*, while a channel with two sink ends is a *spout*. In a connector, the channel ends $[N]$ coincident on a node N may be partitioned into disjoint sets $\mathrm{Snk}(N)$ of sink ends and $\mathrm{Src}(N)$ of source ends. A node N is called *sink node* if only sink ends coincide on it ($\mathrm{Snk}(N) = [N] \neq \emptyset$), vice versa it is called *source node* if only source ends coincide ($\mathrm{Src}(N) = [N] \neq \emptyset$). In all other cases ($[N] = \emptyset$ or ($\mathrm{Snk}(N) \neq \emptyset$ and $\mathrm{Src}(N) \neq \emptyset$)) the node is called *mixed node*. Components may only be connected to sink or source nodes, where they may read from sink nodes and write to source nodes, but they may not be connected to mixed nodes.

Channel types differ in the coordination of writing operations on their source ends and reading operations on their sink ends. Note that Reo allows a user-defined set of primitive channel types to be used. Some examples of possible channel types are given in the next subsection.

Primitive channel types and their composition

Let us consider the following examples of primitive channel types, where their visual representation is given in Figure 2(a). Note that, in contrast to UML and other semi-formal techniques, these visualisations correspond to well-defined entities with a precisely given semantics. Sync facilitates synchronous communication: The writing of a data element at the source end and the deliverance at the sink end happen simultaneously, i. e. data may only flow through the channel if one of the components or channels connected to the source end is ready to write and all of the components or channels connected to the sink end are ready to read. SyncDrain has two source ends and only allows the writing of data elements on both ends simultaneously. SyncSpout non-deterministically produces data elements which can be read from both sink ends simultaneously. FIFO1 allows the entities at the source end to write data into its buffer and entities at the sink end to read this data element from the buffer afterwards. If the buffer is full it must first be emptied through the sink end before another data element may be written through the source end. FIFO1(d) is much like FIFO1, but the buffer is initially filled with the data element d, which must first

be taken through the sink end before the first element can be written to the source end. Filter(pat) is a synchronous channel dropping all data items not conforming to the pattern *pat*.



(a) Channel types (b) Alternator (c) Inhibitor

Figure 2: Channel types and compositions

One of the most outstanding features of Reo is the possibility to create very complex connectors out of a rather small set of primitive channel types. *Alternator-example:* Let us consider the connector in Figure 2(b). It has two source nodes 1 and 2 and a sink node 3. The SyncDrain ensures that data may only be written to the source nodes simultaneously. This can only happen if the entities at sink node 3 are ready to read the data item from source node 1. Moreover, the buffer of the FIFO1 channel must be empty and therefore ready to accept the data item from source node 2. After a write the buffer first has to be emptied by delivering its content to sink node 3 before another write to nodes 1 and 2 can succeed. Thus, this connector delivers data items from source nodes 1 and 2 to sink node 3 in a strictly alternating order starting with an item from source node 1.

Inhibitor-example: The connector in Figure 2(c) allows data to flow synchronously from source node 1 to sink node 2 if the upper FIFO1 buffer is filled and the lower FIFO1 buffer is empty. This is possible since the upper buffer is filled initially with the data element '*'. The SyncDrain synchronises the flow from node 1 to 2 with the data item flowing from the upper buffer to the lower buffer. Some time later this item can flow back to the upper buffer to enable another flow from source node 1 to sink node 2. If a data item is written to source node 3 the lower buffer is filled with this element and all future flows through the connector are inhibited, since the two FIFO1s mutually block each other.

Formal semantics

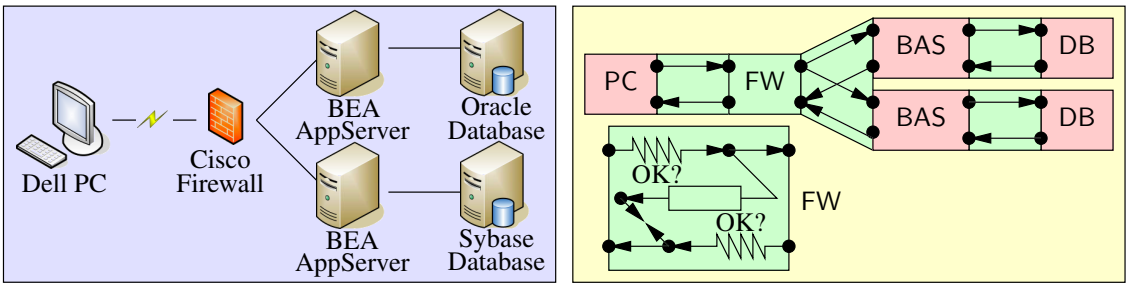
Different kinds of formal semantics have been proposed for Reo. Arbab & Rutten (2003) and Arbab (2005) give a coalgebraic semantics based on *Abstract Behavior Types* (ABTs). ABTs are defined as relations between input and output ports and are used as the semantic models for components and connectors. The messages flowing through these ports are modelled as timed data streams (TDSs) $\langle \alpha, a \rangle$, where α is a possibly infinite stream of data items and a is a strictly increasing stream of real numbers representing the points in time, at which the corresponding data items in α flow through this port. For example, Sync can be modelled as a relation between an input TDS $\langle \alpha, a \rangle$ and an output TDS $\langle \beta, b \rangle$ with $\langle \alpha, a \rangle \mathsf{Sync} \langle \beta, b \rangle \equiv \beta = \alpha \wedge b = a$ and FIFO1 can be modelled as $\langle \alpha, a \rangle \mathsf{FIFO1} \langle \beta, b \rangle \equiv \beta = \alpha \wedge a_i < b_i < a_{i+1}$ for all positions i in the streams.

Baier et al. (2006) use *constraint automata* to specify accepted languages of TDS tuples. The transitions in constraint automata are labelled with the names of the ports on which messages are observed simultaneously and constraints on the data items in these messages. Constraint automata can also be used to specify requirements and provisions of components and connectors. Since they are close to ordinary automata and labelled transition systems this semantic foundation of Reo is suitable to provide model-checking techniques.

Domain-specific business language

The scenario observed in the modelling of the coarse-grained IT infrastructure at Credit Suisse shows that the usage of formal techniques like Reo is not practicable in such an environment. In fact, we need a domain-specific language (DSL) tailored to the skills of IT infrastructure architects. Such a DSL should be close to the informal, intuitive drawings used today, so that these users are not coerced into the steep learning curve necessary to introduce a formal technique. In order to avoid sacrificing the formal foundation, this DSL should, however, have relations to underlying Reo models.

An example of a model in such a language can be found in Figure 3(a), where illustrative images are used as symbols for components and connectors. In Figure 3(b) the underlying Reo model with its connectors and components is given. We observe that a single connection in the DSL model corresponds to a whole connector in Reo with multiple source and sink nodes for the different directions of bidirectional connections. In order to formalise the relation between Reo and the DSL and the manipulation of models in both languages we give an introduction to the theory of typed graphs and algebraic graph transformation in the next section.



(a) IT infrastructure in DSL (b) IT infrastructure in Reo

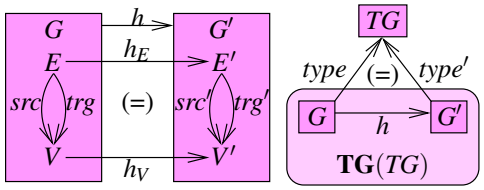
Figure 3: DSL model and underlying Reo model

Typed graphs and algebraic graph transformation

In this section we are using the theory of typed graphs and algebraic graph transformation, as it is presented e. g. by Ehrig et al. (2006), to define on the one hand a formal relation between Reo and a prototypical domain-specific language and show on the other hand how to model transformations such as refactorings or automatic modifications and translations. Graph transformation was chosen because it has a sound mathematical foundation with a long history of research and a wealth of theoretical results.

Typed graphs and language families

A *graph* $G = (N, E, src, trg)$ consists of a set N of *nodes* and a set E of *edges* together with functions $src, trg: E \to N$ assigning to each edge its source and target node. A *graph homomorphism* $h: G \to G' = (h_N, h_E)$ consists of two functions $h_N: N \to N'$ and $h_E: E \to E'$, which are compatible with the source and target functions, i. e. $src' \circ h_E = h_N \circ src$ and $trg' \circ h_E = h_N \circ trg$ (see Figure 4(a)). Graph homomorphisms may translate, merge and embed nodes and edges of the source graph in the target graph. Given a graph TG, called *type graph*, a *typed graph* $(G, type)$ consists of a graph G and a *typing morphism* $type: G \to TG$. Hence, every node of G is labelled with a node type (a node of TG) and every edge with an edge type (an edge of TG). A *typed graph morphisms* is a homomorphism $h: G \to G'$ compatible with the typings, i. e. $type' \circ h = type$ (see Figure 4(b)). The category of all typed graphs and their morphisms for a given type graph TG is denoted by $\mathbf{TG}(TG)$.



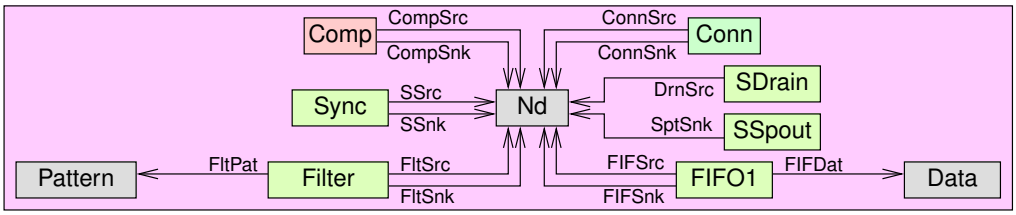
(a) Graph homomorphism (b) Typed morphism

Figure 4: Morphisms for graphs and typed graphs

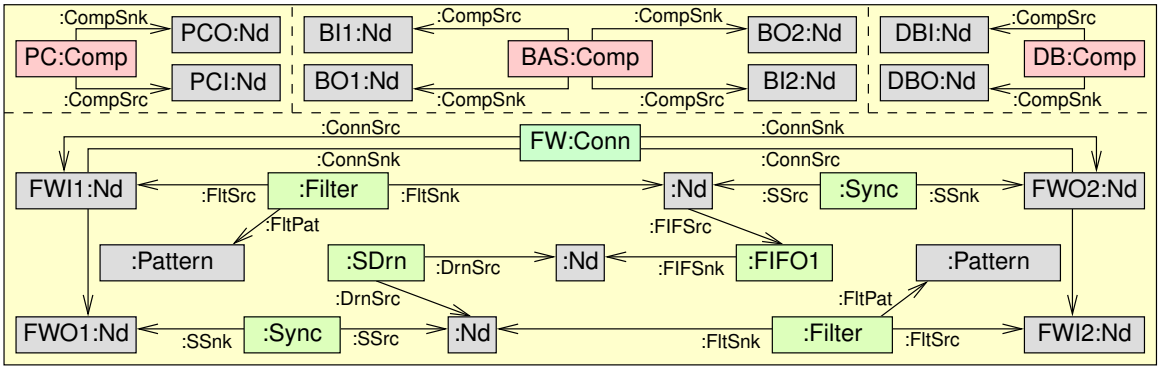
These algebraic notions can be used as a simple formalisation for meta-modelling, where the type graphs are the meta-models or schemas defining an abstract syntax for a class of models, represented by the typed graphs as instances. Meta-models in the sense of MOF (OMG 2006) are, however, much more expressive than type graphs. They can e. g. declare multiplicities, inheritance and attributes of model elements. Some extensions that provide similar features for graphs are given by Ehrig et al. (2006). In this paper, however, we will confine ourselves to the case of simple typed graphs without additional features.

As an example of type graphs and typed graphs consider Figure 5. In Figure 5(a) a simple meta-model for Reo is given by the type graph TG_{Reo}, where node types are defined for Reo nodes, components, connectors and channels, and their connections are given by the edge types represented as arrows. In Figure 5(b) the abstract syntax of the components PC, BAS and DB and the connector FW from the Reo model in Figure 3(b) is given by an instance graph $Reo_{NetDSL} \in \mathbf{TG}(TG_{Reo})$. We give the typing morphism by annotating the types of nodes and edges after a potential identifier and a colon. Note that the connections between these building blocks are not formalised in this model, because they will be specified by domain-specific models later on.

Homomorphisms between type graphs can be used to relate a family of languages of typed graphs. An overview of the proposed language family for domain-specific languages based on Reo is given in Figure 6, where also the



(a) Meta-model TG_{Reo}



(b) Reo model Reo_{NetDSL}

Figure 5: Meta-model for Reo with instance model

intended users of the different sublanguages are visualised. All arrows are graph homomorphisms, where dashed arrows are the typing morphisms between instances and there schemas, while solid arrows are type graph and instance morphisms relating Reo and the domain-specific language by "part of"-relationships.

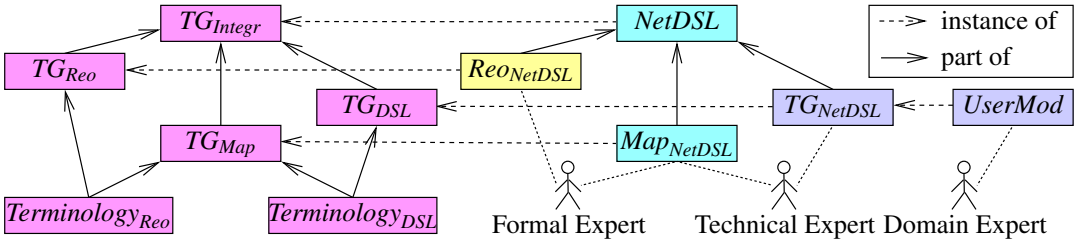
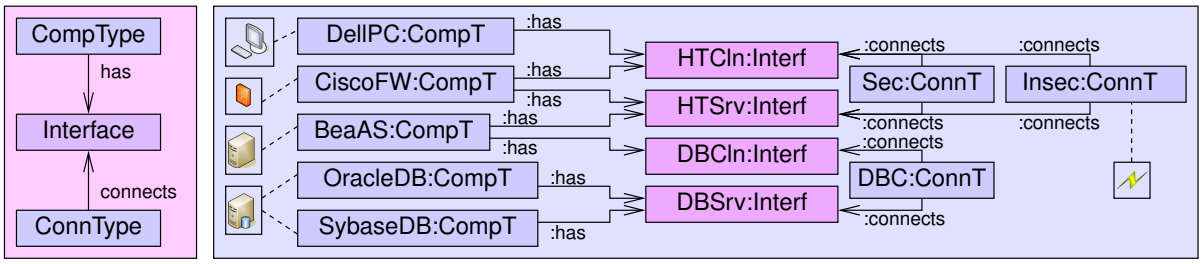


Figure 6: Language family for distributed systems

Experts for formal methods, Reo experts in particular, use the language defined by the type graph TG_{Reo} to create Reo models like Reo_{NetDSL}, but using the visual notion as in Figure 3(b). Independently, technical experts define meta-models for domain-specific languages, which are instances of the "meta-meta-model" TG_{DSL} given in Figure 7(a). Schemas for domain-specific languages consist of node types for components, connections and interfaces. In our example the meta-model TG_{NetDSL} in Figure 7(b) defines a very simple modelling language for network infrastructures, which has component types for clients, servers and firewalls and connection types for HTTP and database connections; e. g., the component type BeaAS for application servers has interfaces HTSrv and DBCln, where HTSrv can be used to connect its instances to some other component with a HTCln interface via a secure or insecure HTTP connection with type Sec or Insec, and DBCln can be connected to a database. TG_{NetDSL} serves as a bridge between two language levels: On the one hand it is an instance of the schema TG_{DSL}, on the other hand it is itself a schema for the user models, network infrastructures in our example, manipulated by domain experts.

The domain experts as end users in our scenario can now use the domain-specific language without explicit skills in the underlying formalisms. As an example the DSL model *UserMod*, whose visual representation was given in Figure 3(a), is shown as instance of TG_{NetDSL} in Figure 8.

The formal and the technical experts work together in creating the mapping from domain-specific types in TG_{DSL}-instances to connector definitions in Reo models. These mappings are instances of TG_{Map} shown in Figure 9(a). The edge types between CompT from TG_{DSL} and Comp and Conn from TG_{Reo} are used to assign underlying Reo components and connectors to the domain-specific component types. The intermediate node types Link, In and Out and the corresponding edge types are used to relate domain-specific interfaces with corresponding source and sink



(a) Meta-model TG_{DSL} (b) Meta-model TG_{NetDSL}

Figure 7: Meta-model for DSLs with meta-model for *NetDSL*

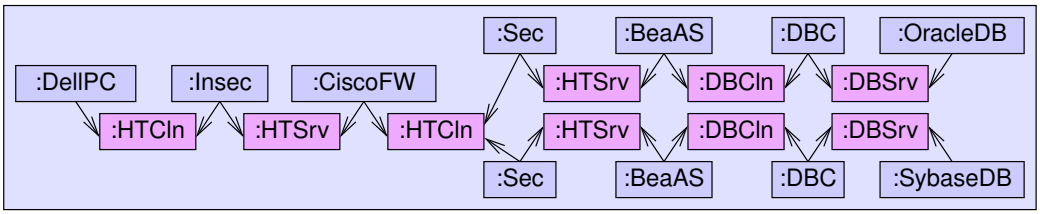
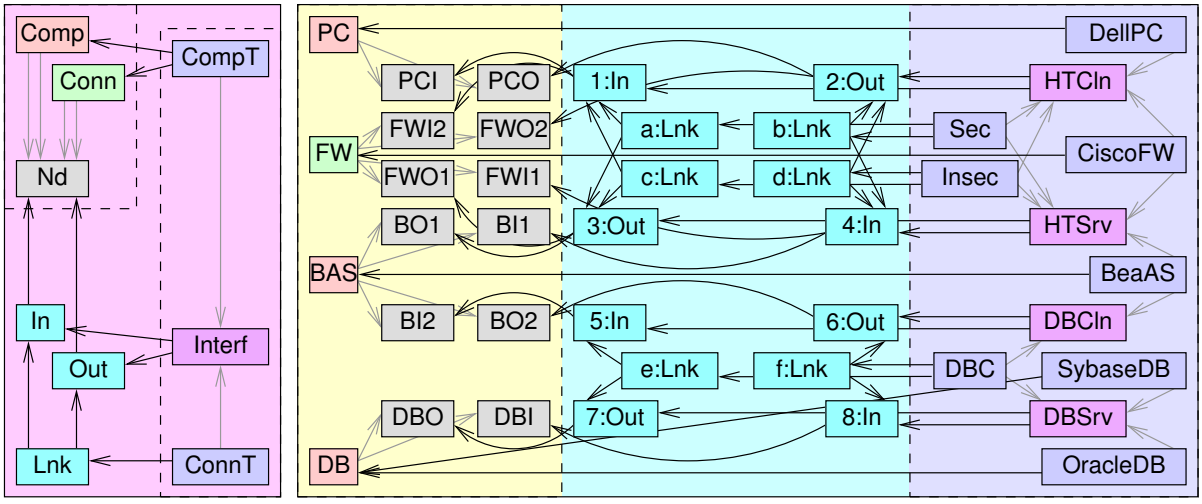


Figure 8: Instance model *UserMod* for IT infrastructure

nodes in Reo and domain-specific connections with links between these ports. The mapping Map_{NetDSL} from the connector, component and connection types of TG_{NetDSL} to the corresponding Reo connectors, components, source and sink nodes is given in Figure 9(b). For example, the Reo component PC is assigned to the domain-specific component type DellPC, which has an interface HTCln consisting of an Out node for request and an In node for responses. These nodes are realised by the corresponding Reo nodes of PC, but also by Reo nodes of the connector FW, which is assigned to another domain-specific component type (CiscoFW) having an HTCln interface. The domain-specific connection type Sec consists of two Links in the mapping, one for each direction of the secure HTTP connection.



(a) Meta-model TG_{Map} (b) Mapping Map_{NetDSL}

Figure 9: Meta-model for mappings with mapping for *NetDSL*

Such a mapping can be used to facilitate the automatic construction of a complete Reo model from *UserMod* or some other domain-specific model. More precisely, the building blocks from Reo_{NetDSL} are copied for each occurrence of a component with corresponding component type and these copies are connected using synchronous channels according to the links specified for the connection types in Map_{NetDSL}. This complete Reo model can then be model checked or otherwise validated.

Note that the integrated type graph TG_{Integr} does not have to be given explicitly to define the relationships between Reo and the DSLs. It can rather be constructed as a *colimit* of the meta-models for Reo, DSL type graphs and mappings over their respective common terminologies $Terminology_{Reo}$ and $Terminology_{DSL}$. Here, the category

theoretical notion of "colimit" can be understood as disjoint unions over common parts. Likewise, the whole definition *NetDSL* of our example DSL can be understood as an *amalgamation* of its constituent parts Reo_{NetDSL}, TG_{NetDSL} and Map_{NetDSL}.

Graph transformation, grammars and refactoring

Given a type graph TG a *transformation rule* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ is defined by three typed graphs $L, K, R \in \mathbf{TG}(TG)$ and two typed graph morphisms $l: K \to L$ and $r: K \to R$, where l is injective. The *application* of a rule p to a *host graph* G by an injective *match* $m: L \to G$ is given by a double-pushout (DPO) as in Figure 10 resulting in a graph G'. Such an application is denoted by $G \overset{p,m}{\Longrightarrow} G'$. Intuitively, an application can be described as the removal of all elements in L that are not reached from K resulting in the context graph C and the addition of the elements in R and not reached from K. If r is non-injective the transformation merges identified elements. A *transformation* $G_0 \Longrightarrow^* G_n$ is given by a sequence $G_0 \Longrightarrow \cdots \Longrightarrow G_n$ of rule applications.

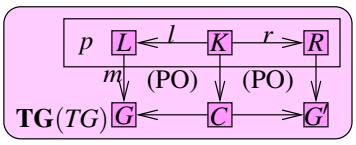
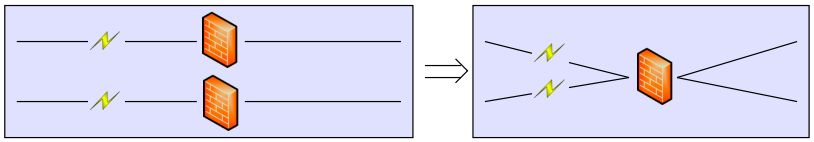


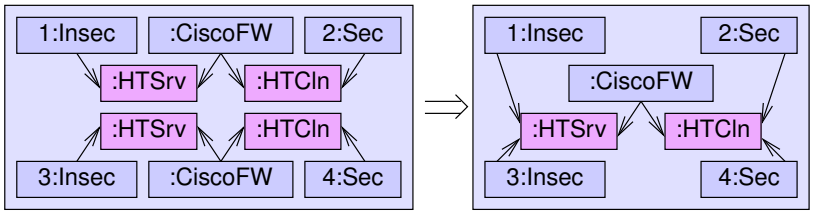
Figure 10: DPO transformations

Transformation rules may be used as *productions* in *graph grammars*. Those grammars are quite similar to Chomsky grammars for textual languages; a type graph TG, a start graph $S \in \mathbf{TG}(TG)$ and a set of productions define a language $L = \{G \in \mathbf{TG}(TG) \mid S \Longrightarrow^* G\}$ of typed graphs. Since in most cases not all instances of a type graph are desired as possible models, grammars should be given for all considered model languages in the scenario: Reo, the domain-specific type graphs, the mappings between them and the domain-specific languages themselves. Grammars can also be used to provide syntax-directed editing, e. g. by using a tool like Tiger (Ehrig et al. 2005) which generates visual editors based on graph grammars, where the rules can also be edited in the visual notation of the DSL.

Rules can also be used to define *refactorings* of typed graphs. A refactoring modifies the structure of a model without changing relevant properties, where the choice of "relevance" may vary. If a refactoring is built from the productions of a grammar it is guaranteed to produce syntactically correct models. As an example of such a rule consider Figure 11, where a domain-specific model is refactored by merging multiple firewalls. The left-hand and right-hand graphs L and R of the rule are explicitly depicted, while the preserved part K is given by the interface nodes 1, 2, 3 and 4 contained in both graphs.



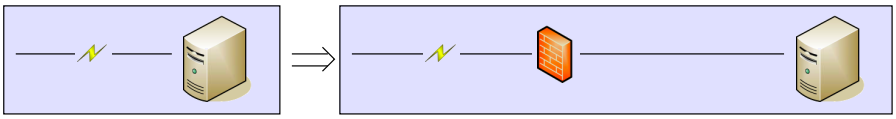
(a) Visual notation



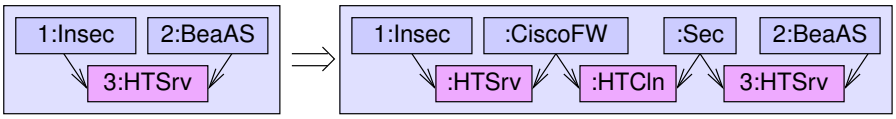
(b) *NetDSL* instance graphs

Figure 11: Refactoring rule for domain-specific models

Moreover, rules may be designed to serve as automatic *model transformations*. Such rules are applied to a model as long as possible using all valid matches. This mechanism is useful for tasks like the translation of models or to perform some cleaning operations, but also to enforce structural patterns. For example, the rule in Figure 12 protects each application server with a firewall.



(a) Visual notatation



(b) *NetDSL* instance graphs

Figure 12: Insertion of a firewall

Negative application conditions and conflict detection

While such model transformations can be used to "clean up" a model and enforce structural requirements ex post, it is also possible to prevent the occurrence of forbidden structures in advance by *negative application conditions* (NACs). A NAC prohibits the application of a rule if certain additional structures are found in the host graph. More precisely, we claim that security requirements can be modelled by *graph constraints* on the domain-specific model. Due to a general correspondence between graph constraints and application conditions (Ehrig et al. 2006) this allows to assure security properties by applying productions with such NACs.

For example, the editing rule creating insecure connections could be equipped with a NAC prohibiting direct connections to application servers, such that the structure repaired by the rule in Figure 12 cannot arise in the first place. Such a behaviour is desired in the scenario if production systems are concerned, which shall never be in a state that violates security requirements.

Another benefit of the theory of algebraic graph transformation is the possibility of automatic conflict detection based on *critical pairs*. Critical pairs are minimal situations in which two rules can be applied, such that the application of one rule prevents the application of the other one. This may become relevant in the scenario if different developers modify the model concurrently and later want to merge their work. If, e. g., one developer removes an element that the other one has duplicated, the question arises which operation should precede. The main result of the theory is that there is a finite set of critical pairs describing all possibilities for such conflicts. At least for some of them sensible resolutions can be designed once and for all on the critical pairs, thereby aiding in distributed concurrent development.

Discussion and future work

The current industrial practice, presented first, illustrated how to cut down costs and development time by leveraging the level of abstraction using domain-specific modelling supported by the tool MetaEdit+. However, the notion of concepts used in MetaEdit+ and similar tools to implement such languages is not able to provide a sound basis for formal reasoning. Only productivity gains were realised, and quality issues were able to be treated only on a technical level.

In contrast to that, the approach of using the Reo coordination model and Abstract Behaviour Types as well as algebraic graph transformation in addition to a domain-specific language yields to similar productivity gains and provides a sound basis for formal reasoning. So, people can use the domain language as intuitively as in the first case to model, administrate, monitor and control their environment of distributed coarse-grained IT components. And they do get as an additional benefit security guarantees out of applicable formal methods like graph transformation or model checking.

Future work will have to handle the mathematically sound definition of a complete domain-specific language in this scenario, like it has been done e. g. for UML state machines and sequence diagrams (Hermann et al. 2006), such that its semantics can be defined by a model transformation to underlying formal techniques, e. g. Reo which is used in this paper. It will also regard the interaction with model checkers and theorem provers. Model checkers should check for such things as tunnelling through a system. Theorem provers should help checking the consistency and completeness of security requirements assigned to components and connectors.

References

Arbab, F. (2004), 'Reo: A channel-based coordination model for component composition', *Mathematical Structures in Computer Science* **14**(3), 329–366. Preprint available at http://homepages.cwi.nl/~farhad/MSCS03Reo.pdf, doi:10.1017/S0960129504004153.

Arbab, F. (2005), 'Abstract behavior types: A foundation model for components and their composition', *Science of Computer Programming* **55**, 3–52. Preprint available at http://ftp.cwi.nl/CWIreports/SEN/SEN-R0305.pdf, doi:10.1016/j.scico.2004.05.010.

Arbab, F. & Rutten, J. J. M. M. (2003), A coinductive calculus of component connectors, *in* D. Pattinson, M. Wirsing & R. Hennicker, eds, 'Recent Trends in Algebraic Development Techniques, Proceedings of the 16th International Workshop on Algebraic Development Techniques (WADT 2002)', Vol. 2755 of *Lecture Notes in Computer Science*, Springer, pp. 35–56. Preprint available at http://ftp.cwi.nl/CWIreports/SEN/SEN-R0216.pdf, doi:10.1007/b94458.

Baier, C., Sirjani, M., Arbab, F. & Rutten, J. J. M. M. (2006), 'Modeling component connectors in Reo by constraint automata', *Science of Computer Programming* **61**(2), 75–113. doi:10.1016/j.scico.2005.10.008.

Ehrig, H., Ehrig, K., Prange, U. & Taentzer, G. (2006), *Fundamentals of Algebraic Graph Transformation*, EATCS Monographs in Theoretical Computer Science, Springer.

Ehrig, K., Ermel, C., Hänsgen, S. & Taentzer, G. (2005), Generation of visual editors as Eclipse plug-ins, *in* 'Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)', IEEE Computer Society. Preprint available at http://tfs.cs.tu-berlin.de/publikationen/Papers05/EEHT05.pdf.

Hermann, F., Ehrig, H. & Taentzer, G. (2006), A typed attributed graph grammar with inheritance for the abstract syntax of UML class and sequence diagrams, *in* D. Varro & R. Bruni, eds, 'Proceedings of the 5th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)'. To appear in Electronic Notes in Theoretical Computer Science, preprint available at http://tfs.cs.tu-berlin.de/publikationen/Papers06/HET06.pdf.

Luoma, J., Kelly, S. & Tolvanen, J.-P. (2004), Defining domain-specific modeling languages: Collected experiences, *in* J.-P. Tolvanen, J. Sprinkle & M. Rossi, eds, 'Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM 2004)', number TR-33 *in* 'Computer Science and Information System Reports', University of Jyväskylä. Available at http://www.dsmforum.org/events/DSM04/luoma.pdf.

OMG (2006), *Meta Object Facility (MOF) Core, v2.0*. Available at http://www.omg.org/cgi-bin/doc?formal/2006-01-01.

Tolvanen, J.-P. (2005), 'Domain-specific modeling for full code generation', *Methods & Tools* **13**(3). Available at http://www.methodsandtools.com/archive/archive.php?id=26.

Acknowledgements

This work is co-sponsored by the Credit Suisse Luxembourg, S.A., and the Ministre de la Culture, de l'Enseignement supérieur et de la Recherche of Luxembourg.

Copyright