A Transformation-Driven Approach to the Verification of Security Policies in Web Designs

Esther Guerra, Daniel Sanz, Paloma Díaz, and Ignacio Aedo

Computer Science Department,
Universidad Carlos III de Madrid (Spain)
{eguerra,dsanz,pdp}@inf.uc3m.es, aedo@ia.uc3m.es

Abstract. In this paper, we present a verification framework for security policies of Web designs. The framework is based on the transformation of the models that conform the system design into a formalism where further analysis can be performed. The transformation is specified as a triple graph transformation system, which in addition creates mappings between the elements in the source and target models. This allows the back-annotation of the analysis results to the original model by means of triple graphical patterns. The verification mechanisms are provided by the designer of the Web design language, together with the language specification. However, the complexities of the formalisms are hidden to the developer who uses the language.

As case study, we apply these ideas to Labyrinth, a domain specific language oriented to the design of Web applications. The analysis is done by a transformation into the Petri nets formalism, and then performing model checking on the coverability graph. The framework is supported by the meta-modelling tool AToM^3.

1 Introduction

Domain Specific Languages (DSLs) are becoming popular due to its capability to capture high-level, powerful abstractions of well-studied application domains, and have the potential to increase the user productivity for modelling tasks. Since they are close to the application domain, they are less error-prone than other general-purpose languages and easier to learn, also because the semantic gap between the user's mental model and the design model is smaller. The Web is a typical domain where the use of DSLs is successful [8,5,15], as there are domain specific terms (e.g. node, link) not provided by general-purpose languages.

Due to the fact that Web systems provide specialized services that cannot be performed by all the system users, correctness of access control policy for Web becomes a crucial issue. Access control is used extensively in information systems as a security mechanism for protecting sensitive information and resources from unauthorized access. The access policy requires to be expressed during the design stage, using the same abstraction level as the one used to capture the system description, instead of delaying access control to the implementation phase. In addition, this integrated, abstract expression allows to validate the access policy at design time, so that inconsistencies can be detected and corrected.

L. Baresi, P. Fraternali, and G.-J. Houben (Eds.): ICWE 2007, LNCS 4607, pp. 269–284, 2007.

In order to validate system designs, a common approach is transforming the models into semantic domains (e.g. Petri nets, process algebra or logic) for analysis. Formal methods [4,6] are techniques based on mathematics or logics that help to specify and verify systems. They are usually applied on the early phases of the development, when the cost of fixing errors is lower [4]. Although they offer significant benefits in terms of improved quality, they are not broadly used due to several reasons, among them their high cost and the need of expert personnel in a certain formal method. This expert knowledge is seldom found among the average software engineers.

In this paper, we follow a transformation-based approach for the verification of security policies in Web designs that hides the complexities of the underlying formal methods from the developers, who specify the Web system in a well-known DSL. Internally, these models are transformed into a semantic domain for further analysis, and feedback is given back to them. The verification process is responsibility of the DSL designer. He specifies the DSL by using meta-modelling, and for the verification, he defines triple graph transformation systems that perform the transformation into the semantic domain and create mappings between the elements in both models. Besides, he can define graphical triple patterns in order to specify how the results of the analysis are shown back to the user in the original notation, that is the Web DSL.

The paper follows an example-driven approach by applying the verification framework to Labyrinth [7], a DSL oriented to the design of hypermedia and Web systems. In particular, we have designed a transformation from Labyrinth into Petri nets [20] in order to analyse system properties, such as the availability of navigational paths or hypermedia objects, taking into account the applied role-based access control (RBAC) model [2]. The approach can be adapted to other Web-oriented notations, as it lies on a meta-modelling framework that does not depend on the DSL.

The paper is organized as follows. Section 2 introduces Labyrinth, which is used as case study throughout the paper. Its use is illustrated through the modelling of ARCE, a Web system for emergency management. Section 3 presents our approach to the verification of security policies with back-annotation of results, and how these ideas have been implemented and applied to ARCE. Section 4 compares with related research. Finally, section 5 ends with the conclusions and future work.

2 Security Modelling in Web Systems. A Case Study: Labyrinth

For the purposes of this work we use the Ariadne Development Method (ADM) [8], a Web engineering method that provides a set of meta-models to specify the information structure, navigation paths, interaction mechanisms, presentation features and access control policies. The method comprises three phases: *Conceptual Design*, that is concerned with the identification of abstract types of components, relationships and functions; *Detailed Design*, where system features, processes and behaviours are specified in detail; and *Evaluation*, where a set of criteria are used to assess system usability from the evaluation of prototypes. Our work focuses on those meta-models of the *Conceptual Design* related with the definition of the access control policy.

The ADM lies on a meta-model called Labyrinth [7] that defines the hypermedia abstractions used in the different diagrams of the ADM. A simplified version of its

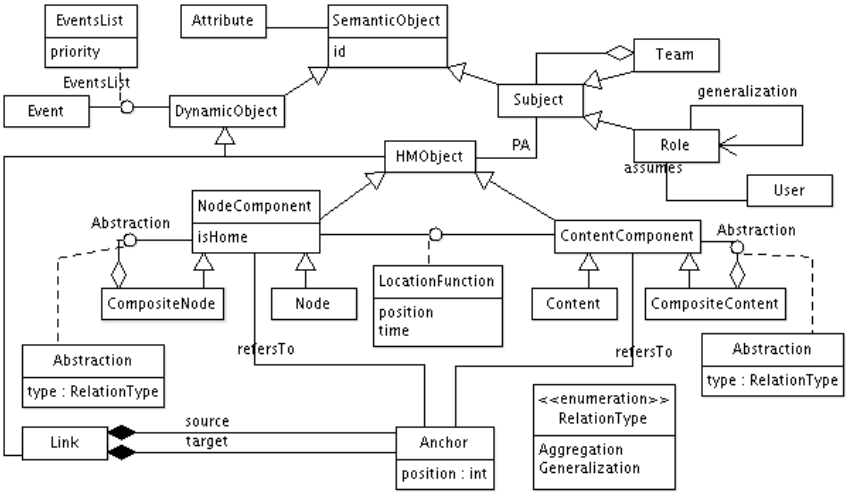

Fig. 1. Labyrinth Meta-model

meta-model with its most salient features is shown in Fig. 1. A complete description of Labyrinth and its contributions can be found in [7]. Here we give a brief description.

Structural features are specified through *nodes* and *contents*. A node is an abstract container where information contents are placed by means of the *location function*, which promotes separation of structure and content. Nodes and contents support composition mechanisms that allow to create complex information structures by using aggregation and generalization. Navigation features are expressed through *anchors* and *links*. An anchor is a link endpoint (whether source or target), and can refer to a content or a node and be shared between links. User modelling is based on the RBAC paradigm [2], and includes *users*, *roles* and *teams*. A role represents a job function within an organization, gathering a set of permissions, and a set of users allowed to exercise them. Roles decouple privileges from users, facilitating the management of authorizations due to the fact that roles are more stable than users in terms of responsibilities. Relationship *assumes* relates users with the roles that are allowed to hold, while the permission assignment (relationship *PA*) relates subjects to the nodes and contents that the role is allowed to visit. Changes in user privileges are managed through role memberships. As some roles can be more general than others, we define a generalization relation that implies inheritance: permissions assigned to the more general roles are inherited by more specific roles. A team represents a collection of heterogeneous roles and/or teams, and captures groups of interest or collaborative teams. Users cannot be assigned to teams, but instead participate in a team through role membership. A permission assigned to a team is propagated to all team components. Note that it is not necessary to explicitly define all permissions in the system, as they are propagated throughout the roles and teams structure as described in [1]. Anchors get the permission of the node or content to which are tied, while links are available for a role if at least one anchor of each end is available, that is, the role has access to the source and target link ends.

2.1 Modelling Example: The ARCE System

This section presents an example that will be used to illustrate the features of the verification framework. The ARCE system is aimed at resource management for international cooperation in case of disaster[1]. When an emergency happens, the Civil Protection Department of the affected country uses ARCE to create an emergency report. If international cooperation is needed, the affected country can ask for resources to other countries, which in their turn may offer a contribution. Here we will focus on emergency report management, resource requests and resource contributions.

2.2 ARCE System Design

Part of the nodes making the Web application, as well as their structural relationships, are specified in the *structural diagram* shown in Fig. 2. Composite node `Home` acts as root of the system, while abstract nodes `Requests` and `Contributions` group nodes related to each activity respectively. Each leaf node corresponds to a Web page that includes the contents required to do a particular function in ARCE.

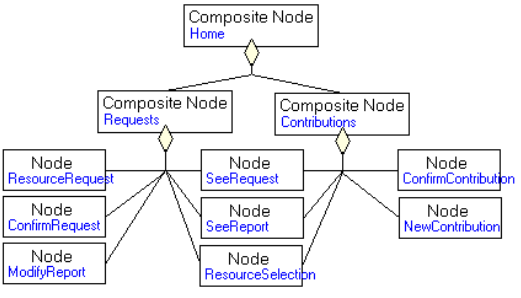

Fig. 2. Excerpt of the ARCE Structural Diagram in ADM

The *navigation diagram* in Fig. 3 captures the navigation paths by means of nodes and links. To request a resource, the user navigates from `Home` to `Resource Request`. To add a resource, the user goes to the `ResourceSelection` node, picks the desired resource and indicates the quantity, and finally returns back to the `ResourceRequest` node, where an editable list of resources is shown. When the request is ready, the user confirms it and returns `Home`. Optionally, the user may see or modify the emergency report during the request. To make a contribution, the user navigates from `Home` to `NewContribution`, where some details of the offer are filled and the list of resources is prepared. To add a resource to the offer, the users goes to node `ResourceSelection`, which in this case shows the contributor's available resources, picks one, selects the quantity and returns to the `NewContribution` node.

For each node, an *internal diagram* describes its internal structure made by the contents and spatial-temporal relationships that define the presentation aesthetics. For example, the internal diagram for the `SeeReport` node is shown in the window to the

[1] http://arce.dei.inf.uc3m.es

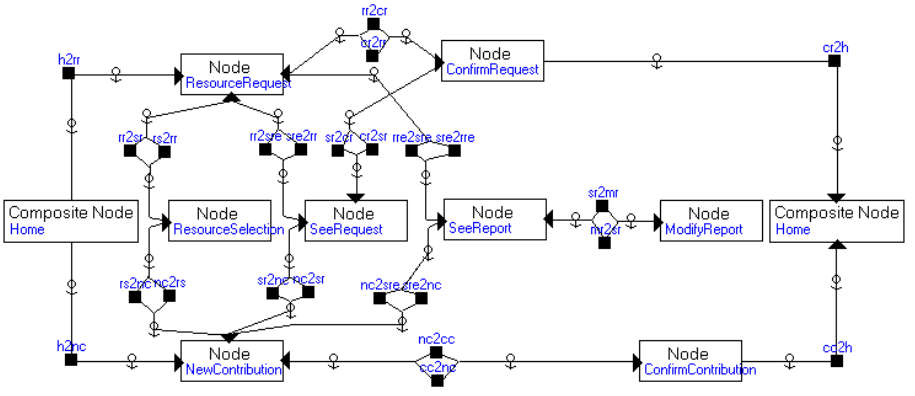

Fig. 3. ARCE Navigation Diagram in ADM

left of Fig. 8. This node includes three contents: `ReportHeader`, `GeneralData` and `TechnicalData`. The contents are available according to the role that accesses the node, which is defined by the access policy.

Fig. 4 shows the *user diagram* of ARCE. The team `ARCE User` aggregates teams `Requester` and `Contributor`. Role `ROperator` can request resources, but cannot confirm them. He can see all the report contents. Role `RExpert` is an expert operator who can do the same as a `ROperator` and, in addition, modify the report. `Emergency Manager` is a directive role that confirms the requests and only can see the general data of the report. Role `COperator` can make a contribution but not confirm it. He can access the general data of the report, but not modify it. Role `CExpert` does the same as `COperator`, but he also can read the technical data of the record. Finally, `Contributor Manager` is a directive role that can see the contributions, confirm them and read the general data of the report.

Access policy, expressed as permissions on nodes and contents, is depicted in Table 1. The • symbol denotes the assignment of a permission to a role/team for accessing a node/content, and the ▽ symbol represents an inherited permission. The Web designer only specifies the direct assignments, and the inherited ones are derived.

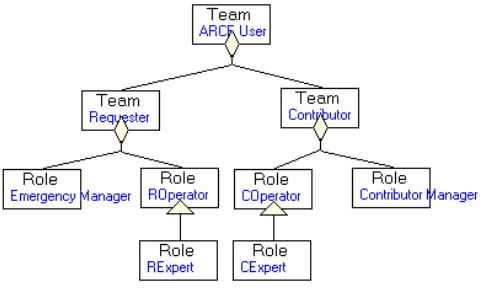

Fig. 4. ARCE User Diagram in ADM

Table 1. Access Policy for ARCE System Nodes and Contents

		Subjects								
		ARCEUsr	Requ	ROp	RExp	EMgr	Contr	CExp	CMgr	COp
Nodes	Home	●	▽	▽	▽	▽	▽	▽	▽	▽
	ResourceSelection			●	▽					
	ResourceRequest		●	▽	▽	▽				
	SeeRequest	●	▽	▽	▽	▽	▽	▽	▽	▽
	ConfirmRequest					●				
	SeeReport	●	▽	▽	▽	▽	▽	▽	▽	▽
	ModifyReport				●					
	NewContribution						●	▽	▽	▽
	ConfirmContribution								●	
Contents	ReportHeader	●	▽	▽	▽	▽	▽	▽	▽	▽
	GeneralData	●	▽	▽	▽	▽	▽	▽	▽	▽
	TechnicalData				●			●		

With this access policy, for example, role `Emergency Manager` can access to contents `ReportHeader` and `GeneralData` (permissions inherited from team `ARCE User`) but not to `TechnicalData`.

3 A Transformation-Driven Approach to Security Analysis

When modelling security for software systems, there is a need in verifying the correctness of the access policy in terms of the availability of different system entities. In the case of Web systems incorporating RBAC policies (e.g. Labyrinth), we are particularly interested in verifying the availability of navigation paths, nodes and contents for different subjects, the absence of nodes or contents that are not available to any subject, which contents are shown to each subject, and checking if it is possible for a subject to reach a node from which no link to any other node is available (i.e. there is a *deadlock*).

In order to be able to answer these questions, we have used a common technique that consists of expressing the operational semantics of the model(s) to be analysed by using a formalism [14,19,23,24]. The used formalism must provide the necessary tools to answer such questions. In this case, we transform the Labyrinth designs into Petri nets [20], which provide analysis techniques to investigate system properties such as deadlock, reachability of states and invariants, which are the kind of properties we are interested in. The transformation is performed by a triple graph transformation system (TGTS) [13] that builds, from a Labyrinth model, the equivalent Petri net for a system subject. Once the net is obtained, we use analysis techniques based on the reachability/coverability graph, as well as model checking, in order to verify system properties. Finally, the results are back-annotated and shown to the user in the original notation, which is the one he knows. Next subsections explain this process in detail.

3.1 Transformation from Labyrinth into Petri Nets

Graph transformation [9] is an abstract, declarative, visual, formal and high-level technique to express computations on graphs (and therefore on models). Roughly, graph transformation systems are made of rules having graphs in their left and right hand sides (LHS and RHS respectively). In order to apply a rule to a *host graph*, a morphism (an occurrence or match) of the LHS has to be found in it. Then, the rule is applied by substituting the match by the rule's RHS. The execution of a graph grammar consists of a non-deterministic application of its rules to an initial graph, until no rule is applicable. In addition, rules can be equipped with application conditions that restrict their applicability. One of the more used is the so-called Negative Application Condition (NAC). This is a graph that, if found in the host graph, forbids the rule application. Finally, we can combine meta-modelling and graph transformation allowing abstract graph nodes to appear in rules [10]. In this way, nodes can be matched to instances of any subclass, greatly improving the expressive power of rules.

In model-to-model transformation, it can be useful to manipulate triple graphs instead of the usual ones. Triple graphs are made of three different graphs: the source graph (of the transformation), the target one, and a correspondence graph that relates the elements in the other two graphs. Similarly to graph grammars, TGTSs [13] are used in order to manipulate triple graphs.

In this paper, we have defined a TGTS that builds a Petri net from a Labyrinth model. By using a TGTS we create correspondences between the Labyrinth and the resulting Petri net elements that facilitate the back-annotation of the analysis results in terms of the Labyrinth notation. Fig. 5 shows some rules of this TGTS. The source graph (shown in the upper part of the rules) corresponds to Labyrinth, while the target graph (lower part of the rules) is the Petri nets formalism. We use a compact notation for the rules, in which the LHS and the RHS are presented together. The elements to be added by the rule application (i.e. those that belong to the RHS but not to the LHS) are shown in a gray area and labelled as "new". NACs are omitted for clarity, and in all the rules are equal to the RHS. The general idea of the transformation is creating a net that simulates the behaviour of a subject (i.e. a role or team). For this purpose, a Petri net place is created for each node and content for which the subject is granted, and links between them are transformed into Petri net transitions. The subject is represented as a token; therefore, if a token is in a certain place, that means that the subject is accessing to the node or content(s) that the place represents. Note that visiting a node implies visiting all authorized contents for the subject, and thus, the Petri net marking gives the set of nodes and contents accessed in a given moment by the subject.

The three rules to the left in Fig. 5 perform the flattening of the subject's hierarchy. Rule *Flattening1* creates a correspondence element *CElem* with a morphism to the subject for which the Petri net is calculated. The rule receives such subject as parameter. This is an abstract rule: therefore, the rule is applicable to any subclass of subject (i.e. classes *Team* and *Role*). Then, rules *Flattening2* and *Flattening3* traverse the subject's hierarchy creating correspondence elements with morphisms to each subject's ancestor.

Rule *HMObject2Place* in Fig. 5 creates a place for each hypermedia object (i.e. node or content) to which the subject or its ancestors have permission to access (i.e. a correspondence element to the subject was created by the execution of the flattening

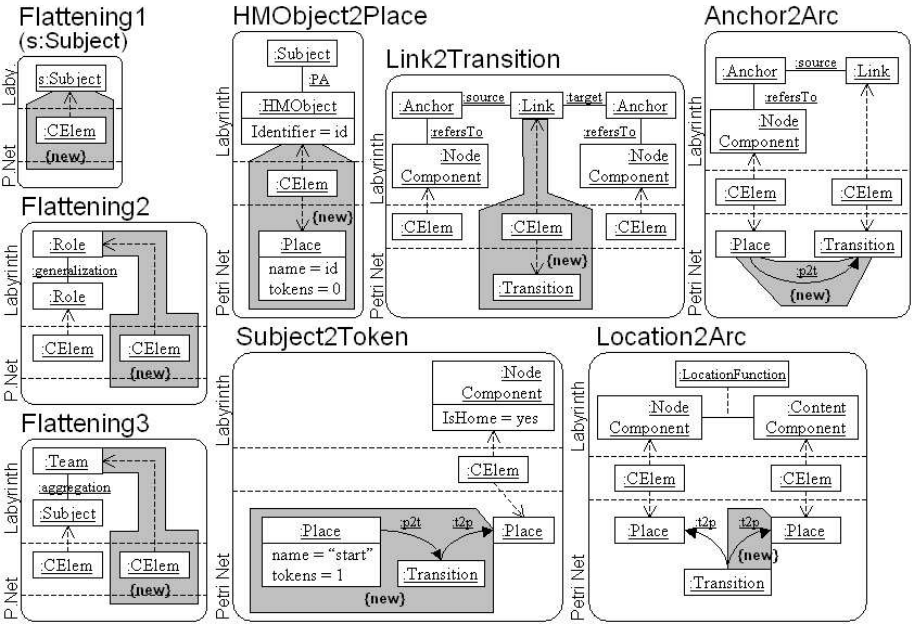

Fig. 5. Some Triple Rules of the Transformation from Labyrinth to Petri Nets

rules). Similarly, rule *Link2Transition* creates a Petri net transition for each link. We only transform those links between hypermedia objects for which the subject has access, that is, objects that have a correspondence element created by previous executions of rule *HMObject2Place*. Another similar rule is used for the case in which the source of the link is a content component instead of a node component. Rule *Anchor2Arc* creates an incoming arc from a place to a transition if the corresponding node is source of the corresponding link. Two similar rules create an arc when the source is a content component, or when the node is target of the link (creating in that case an outgoing arc from the transition). Rule *Subject2Token* creates an extra place with a token (the subject) and a transition from it to the place related to the home page. This transition models the first access of the subject to the system. Finally, each time a transition is fired (i.e. each time a link is traversed), a token must be placed not only in the target node, but also in the target node contents for which the subject is granted. For this purpose, rule *Location2Arc* creates the appropriate arcs to such contents. Similarly, leaving a node implies leaving its contents. Again, two similar rules create the necessary arcs.

Note that some system information is lost in the proposed transformation (e.g. the position of anchors), however, such information is useless for our analysis purposes. Thus, we require from the transformation to preserve the properties under investigation (availability of navigation paths depending on the security policy), as we do in this case.

Fig. 6 shows the Petri net resulting from applying the presented TGTS to the role `Emergency Manager` in the ARCE Web example.

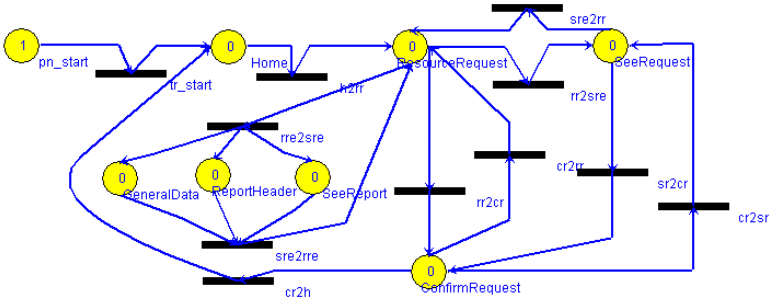

Fig. 6. Resulting Petri Net for Role `Emergency Manager`

3.2 Analysis and Back-Annotation

In order to verify a property, we first apply to the Labyrinth model the presented TGTS and create the Petri net for a certain subject. Then, we obtain the net's coverability graph (an approximation of the possibly infinite net state space), and apply model checking [20] on the graph in order to verify the property. Properties are expressed in Computational Tree Logic (CTL). CTL formulae are made of atomic propositions, the names of the places of the Petri net, which are true for a given net state if the place contains at least one token. Propositions can be combined with boolean connectors ($\wedge$, $\vee$, $\neg$), path quantifier operators that express if predicates are fulfilled starting from a certain state, and temporal quantifier operators that describe the properties of a branch in the computation tree. Valid path quantifiers are E (exists a path) and A (for all the paths). Valid temporal quantifiers are X (in next state) and U (until). Other quantifiers, such as F (in some state in the future) and G (always), can be expressed in terms of X and U. The result of checking a property on a model is the set of states satisfying the given property.

In the case of Labyrinth, we are interested in verifying the following properties:

1. *A specific navigational path is allowed for a given subject*. Let s be a subject, and $Np =< node_1, node_2, ..., node_N >$ a navigational path where subindex i specifies the order in which nodes are visited. Np can be expressed as the recursive function *next(i) = node*$_i \wedge$ *EX (next(i+1))* if $i < N$, and $next(i) = node_i$ if $i = N$. Thus, this property can be written as the CTL expression *E True U (next(1))*, which is evaluated on the coverability graph of the Petri net obtained for subject s. This property allows checking if a subject can perform a task that implies traversing certain navigation path. For example, in order to validate a contribution, the role `Contributor Manager` should be able to go from node `Home` to node `ConfirmContribution` and then return `Home`.
2. *A specific node or content is never shown to a given subject*. Let s be a subject and hmo a node or content. Then, this property can be expressed as $\neg$ *(E True U hmo)*, which allows detecting elements that should be available for a subject but are not,

as well as checking if a subject has more permissions than required. For example, as no other role than `RExpert` can modify a report, the node `Modifyreport` must not be shown to any other subject.

3. *A specific node or content is never shown*. Let S be the set of system subjects, and hmo a node or content. Then, this property can be expressed as $\bigwedge_{s \in S} \neg$ *(E True U hmo)*. This is an extension of property number 2 to the set of subjects of the system.
4. *A specific node or content is shown in each navigational path for a given subject*. Let s be a subject, and hmo a node or content. Then, this property can be expressed with the CTL expression *A True U hmo*, which checks that all the possible navigation paths followed by a subject will show the specified object. For example, node `Home` should be accessed in any possible path.
5. *A subject does not reach a deadlock state*. In other words, all nodes define at least one outgoing link. Let s be a subject, then the property can be written as $\neg$ *(E True U deadlock)*, where predicate $deadlock$ becomes $true$ in states with no successor.
6. *A specific node shows at least one content for a subject*. Let s be a subject and n a node. Then the expression n gives the Petri net markings that satisfy the expression. Note that if the node has a token, then its contents for which the subject is granted have also one and belong to the marking. This property allows detecting nodes that are empty for certain subjects due to a bad design of the access policy.

In order to hide the analysis process to the Web designer (who is proficient in the Web domain and the used DSL) we back-annotate the results to the Labyrinth model. This is possible since we maintain in the correspondence graph the relations between the elements in the source and target models. The elements to back-annotate can be specified as a triple pattern that receives as parameters the Petri net states or transitions to back-annotate, and as output the Labyrinth elements resulting from the back-annotation. For example, Fig. 7 shows to the left the triple pattern used to specify how results are shown to the user in the case of property type number 6. The pattern is executed for each place (the input) obtained as result of the analysis. The output is the set of contents related to those places. Similarly, the pattern to the right in the same figure is used for the back-annotation of properties number 2, 3 and 4. The analysis method used for these properties returns the sequence of firing transitions that leads from the system initial state to the analysed node or content. These transitions are the input of the pattern. The output is the set of links related to the transitions, together with their sources, targets and corresponding anchors. Thus, in the Labyrinth model will be shown each possible navigational path leading to the hypermedia object under study.

3.3 Tool Support

The whole verification framework has been implemented in the AToM^3 tool [17], which allows the specification of visual languages by means of meta-modelling, and the manipulation of graphs by means of graph transformation. Recently, the tool has been enhanced with the possibility of expressing multi-view visual languages [12], which are languages made of different diagram types, such as the presented ADM (or the general purpose UML). Thus, we have defined the whole Labyrinth meta-model in AToM^3, and then the different diagrams types as subsets of it. The tool provides syntactic and static

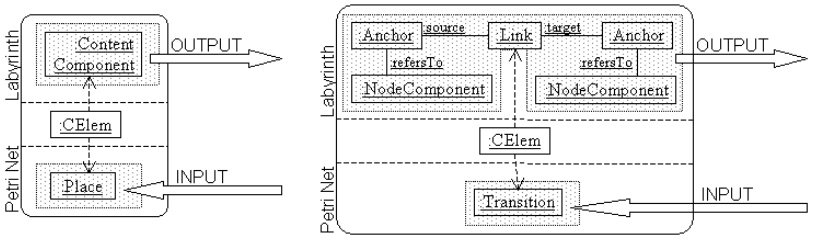

Fig. 7. Some Back-annotation Triple Patterns

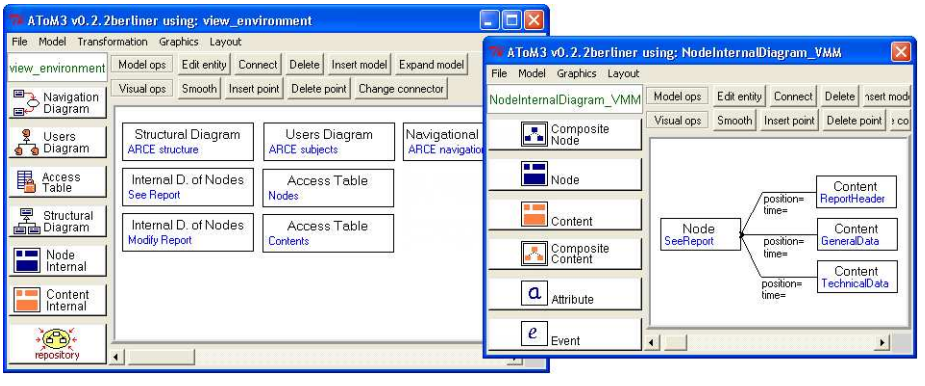

Fig. 8. Generated Environment for Labyrinth

semantics consistency between diagrams by means of automatically generated TGTSs that build a *repository* made of the gluing of the different diagrams. For the analysis of the dynamics, it is possible to define semantic views to be generated by means of TGTSs from the models. In the case of Labyrinth, we have defined the presented TGTS to transform the repository into Petri nets. In addition, each property to be verified in the semantic view can be specified with: (i) a pre-process method where the request of data required for the analysis (e.g. the name of the subject for which the property is checked) is specified; (ii) an analysis method call; and (iii) a back-annotation mechanism specified either procedurally or by triple patterns. For the analysis of properties in Labyrinth, method calls use analysis functions that calculate the coverability graph in AToM^3 and use a model checker implemented in the tool as well. It is up to future work the use of external Petri net tools to perform the analysis.

Starting from this definition, AToM^3 generated a modelling environment that allows specifying instances of the different diagram types. This environment is shown to the left of Fig. 8, where the different diagrams of ARCE have been defined. The figure shows to the right the editing of the internal diagram of node `SeeReport`.

The environment automatically creates a repository with the gluing of the system diagrams. In the repository interface (background window in Fig. 9), a button is generated for each Labyrinth concrete class and for each analysis property. Buttons derived

from classes allow adding new entities to the repository. Buttons derived from properties allow checking properties, and the result is shown according to the defined back-annotation mechanism, hiding the internals of the analysis process. If the back-annotation is specified by a pattern, the output elements obtained from its application are highlighted in the original model, as well as summarized in a dialog window. Fig. 9 shows the checking of a property in the Labyrinth repository and the back-annotation of the results. In addition, a button is generated that allows showing the result of executing the TGTS. This can be used for simulation purposes.

3.4 Verifying ARCE Access Policy

The generated environment has been used for the modelling of ARCE. The verification of the availability of navigation paths for different roles (property of type number 1 in subsection 3.2) implied just clicking on the button *Check NavigationPath* in the repository interface, which is shown in Fig. 9. The name of the subject and the sequence of nodes in the navigation path to be checked are requested to the Web designer. Then, the environment internally builds the corresponding CTL expression, performs the analysis, and the result is shown in a dialog window.

Similarly, checking to which contents of a node a subject could access (property number 6) was done by clicking on the button *Check Node Contents*. The name of the subject and node are requested to the user. Then, the node contents are shown highlighted (by the execution of the left pattern in Fig. 7 to the analysis result), as well as summarized in a dialog window. Fig. 9 shows the result obtained after checking the property for role `Emergency Manager` and node `SeeReport`. Note how content `TechnicalData`, although was specified in the internal diagram of the node, is not accessible for this role due to the specified access policy.

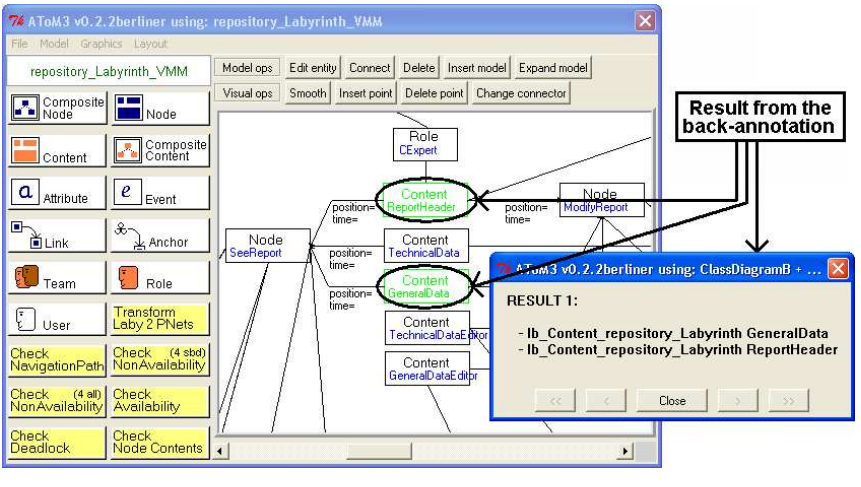

Fig. 9. Back-annotation of Property Checking. The result is highlighted in the model and summarized in a dialog window.

Table 2. Some Properties Verified in ARCE

Prop. Type	Verified Property		Expected Result	Observed Result
	Subject	CTL Expression		
1	ROp	*E True U (Home ∧ EX (ResourceRequest ∧ EX (ResourceSelection ∧ EX (ResourceRequest ∧ EX Home))))*	true	true
	EMgr	*E True U (Home ∧ EX (SeeReport ∧ EX (SeeRequest ∧ EX (ConfirmRequest ∧ EX Home))))*	true	false
	COp	*E True U (Home ∧ EX (NewContribution ∧ EX (SeeReport ∧ EX (ModifyReport ∧ EX (SeeReport ∧ EX (NewContribution ∧ EX (ResourceSelection ∧ EX (NewContribution ∧ EX Home))))))))*	false	false
	CMgr	*E True U (Home ∧ EX (ConfirmContribution ∧ EX Home))*	true	false
2	CExp	*¬ (E True U ModifyReport)*	no	no
	RExp	*¬ (E True U ModifyReport)*	yes	yes
	COp	*¬ (E True U SeeReport)*	yes	yes
	CMgr	*¬ (E True U NewContribution)*	no	no
3	–	$\bigwedge_{s \in \{ARCEUsr,Requ,Contr,ROp,RExp,EMgr,COp,CExp,CMng\}} \neg$ *(E True U SeeReport)*	no	no
4	ROp	*A True U ModifyReport*	no	no
	ROp	*A True U Home*	yes	yes
5	ROp	*neg (E True U deadlock)*	true	true
	CMgr	*neg (E True U deadlock)*	true	true
6	EMgr	*SeeReport*	yes	yes

Table 2 summarizes some properties that were checked in the ARCE design by using the presented approach and environment. For each kind of property described in section 3.2, we provide some example concrete CTL formulae, together with the expected and observed results. Results for properties 2, 3, 4 and 6 are back-annotated to the original model. For the rest of properties, the result is given as a true/false dialog window.

4 Related Work

The use of Petri nets for the formal specification, simulation and analysis of software systems (among them Web and security systems) is spread. For example, in [11] navigational paths are modelled by using Petri nets, where temporal links are also considered. [3] presents a formal XML firewall security model using RBAC based on Petri nets. In [22] security analysis of extended role based access control systems is modelled by using coloured Petri nets. In all these cases, the designer models the system directly as a Petri net, where verification is performed. In this paper we also use Petri nets for system verification, but propose the use of DSLs that include concepts especially suitable for the domain to be modelled (e.g. node, link, anchor), which makes system specification easier for the Web designer. Verification is provided by translating the specific domain models into Petri nets and then performing model checking on the net's coverability graphs. However, the transformation and analysis are hidden to the Web designer, as the results are back-annotated and shown in the original notation.

Approaches to model transformation for the analysis of systems by its translation into a semantic domain are frequent, mainly oriented to the validation of UML models [14,19,23,24], some of them providing support for back-annotations as well. Quite similar to ours is the approach followed in [23], where reference models are used to interrelate the elements of the source and target models in a single graph, allowing the back-annotation of analysis results. The reference model is similar to the notion of correspondence graph in triple graphs that we are using in this work, though our approach maintains the two graphs cleanly separated (i.e. we have separate models and meta-models for Labyrinth and Petri nets). In addition, triple graphs are more flexible as no additional structure is needed in the models in order to maintain the correspondences.

Validation techniques have been also applied to RBAC in works like [16,18] in order to check inconsistencies in terms of (non-)existence of permissions or verification of the RBAC model itself. These approaches are general or domain-independent, in the sense that the RBAC and system meta-models are separated. On the contrary, we base on a DSL that includes elements for the modelling of access policies in its meta-model. These elements are specially suite for the Web domain. Other works, such as [25,21], also allow to include domain-dependent modelling entities in the verification process. However, these inclusions seem to be done by hand, without an automatic mechanism able to transform a system model to the chosen formalism, so that back-annotations are hard to implement.

5 Conclusions and Future Work

In this paper we have presented a formal verification framework for security policies on Web systems that hides the complexity of the formalisms from the Web designer. The framework has been illustrated by its application to Labyrinth, a DSL oriented to the design of Web applications. We have designed a transformation from Labyrinth into the Petri nets formalism, which allows checking model properties such as reachability or deadlocks. The analysis of properties is made by performing model-checking of the coverability graph by using temporal logic formulae. Analysis results are back-annotated by using triple patterns.

The present work uses a simplified version of the complete Labyrinth meta-model. It is up to future work the extension to the complete meta-model, which includes for example categorization of permissions. We are also studying the transformation into Timed Petri nets that include temporal constraints (e.g. temporal anchors and security constraints). The transformation into coloured Petri nets would possibly allow to generate a single net for the whole system instead of for each subject.

Acknowledgements. Work sponsored by projects TIN2006-09678 of the Spanish Ministry of Education and Science (MEC); TSI2004-03394 of the MEC and an agreement between Univ. Carlos III (UC3M) and Dir. Gral. de Protección Civil y Emergencias (DGPCE); and ARCE, supported by an agreement between UC3M and DGPCE.

References

1. Aedo, I., Díaz, P., Montero, S.: A methodological approach for hypermedia security modelling. Inform. Software Technol. 45(5), 229–239 (2003)
2. ANSI INCITS 359-2004. Role Based Access Control (2004)
3. Ayachit, M.M., Xu, H.: A Petri Net based XML Firewall Security Model for Web Services Invocation. In: Proc (547) Communication, Network, and Information Security (2006)
4. Berry, D.M.: Formal methods: the very idea. Some thoughts about why they work when they work. Science of Computer Programming 42(1), 11–27 (2002)
5. Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., Matera, M.: Designing Data-Intensive Web Applications. Elsevier Science (2003)
6. Clarke, E.M., Grumberg, O., Long, D.: "Model Checking". In: Proceedings of the International Summer School on Deductive Program Design Marktoberdorf, Germany, 1994. In M. Broy, "Deductive Program Design", NATO ASI Series F, vol. 152, Springer, Heidelberg (1996)
7. Díaz, P., Aedo, I., Panetsos, F.: Labyrinth, an abstract model for hypermedia applications. description of its static components. Information Systems 22(8), 447–464 (1997)
8. Díaz, P., Montero, S., Aedo, I.: Modelling hypermedia and web applications: the Ariadne Development Method. Information Systems 30(8), 649–673 (2005)
9. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation. vol. 1. World Scientific (1997)
10. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. In: Monographs in Theoretical Computer Science, Springer, Heidelberg (2006)
11. Furuta, R., Na, J.: Applying programmable browsing semantics within the context of the world-wide web. In: Proc. of Hypertext 02, pp. 23–24. ACM Press, New York (2002)
12. Guerra, E., Díaz, P., de Lara, J. (eds.): A formal approach to the generation of visual language environments supporting multiple views. In: Proc. VL/HCC'05, pp. 284–286. IEEE Computer Society Press, Los Alamitos (2005)
13. Guerra, E., de Lara, J.: Attributed typed triple graph transformation with inheritance in the double pushout approach. Tech. Report UC3M-TR-CS-06-01, Universidad Carlos III (2006)
14. Guerra, E., de Lara, J.: Model View Management with Triple Graph Transformation Systems. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 351–366. Springer, Heidelberg (2006)
15. Koch, N., Kraus, A.: Towards a Common Metamodel for the Development of Web Applications. In: Lovelle, J.M.C., Rodríguez, B.M.G., Gayo, J.E.L., Ruiz, M.d.P.P., Aguilar, L.J. (eds.) ICWE 2003. LNCS, vol. 2722, pp. 497–506. Springer, Heidelberg (2003)
16. Koch, M., Parisi-Presicce, F.: Visual Specifications of Policies and Their Verification. In: Pezzé, M. (ed.) ETAPS 2003 and FASE 2003. LNCS, vol. 2621, pp. 278–293. Springer, Heidelberg (2003)
17. de Lara, J., Vangheluwe, H.: $ATo M^3$: A tool for multi-formalism modelling and meta-modelling. In: Kutsche, R.-D., Weber, H. (eds.) ETAPS 2002 and FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002)
18. Li, N., Tripunitara, M.V.: Security Analysis in Role-Based Access Control. ACM Transactions on Information and System Security 9(4), 391–420 (2006)
19. Machado, R.J., Lassen, K.B., Oliveira, S., Couto, M., Pinto, P.: Execution of UML Models with CPN Tools for Workflow Requirements Validation. In: Proc. of the Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools. pp. 231–250 (2005)
20. Murata, T.: Petri nets: Properties, analysis and applications. In: Proceedings of the IEEE, vol. 77(4) pp. 541–580 (1989)

21. Schaad, A., Lotz, V., Sohr, K.: A Model-checking Approach to Analysing Organisational Controls in a Loan Origination Process SACMAT 2006. pp. 139–149 (2006)
22. Shin, W., Lee, J.G., Kim, H.K., Sakurai, K.: Procedural Constraints in the Extended RBAC and the Coloured Petri Net Modeling. IEICE Trans. on Fundamentals 88(1), 327–330 (2005)
23. Varró, D., Varró, G., Pataricza, A.: Designing the automatic transformation of visual languages. Sci. Comp. Programming 44(2), 205–227 (2002)
24. Xie, F., Levin, V., Browne, J.C.: ObjectCheck: A Model Checking Tool for Executable Object-oriented Software System Designs. In: Kutsche, R.-D., Weber, H. (eds.) ETAPS 2002 and FASE 2002. LNCS, vol. 2306, pp. 331–335. Springer, Heidelberg (2002)
25. Zhang, N., Ryan, M., Guelev, D.P.: Evaluating Access Control Policies Through Model Checking. In: Zhou, J., Lopez, J., Deng, R.H., Bao, F. (eds.) ISC 2005. LNCS, vol. 3650, pp. 446–460. Springer, Heidelberg (2005)