MODEL-DRIVEN DESIGN OF SECURE HIGH ASSURANCE SYSTEMS: AN INTRODUCTION TO THE OPEN PLATFORM FROM THE USER PERSPECTIVE

Steve Boßelmann and Johannes Neubauer and Stefan Naujokat and Bernhard Steffen

Chair of Programming Systems
TU Dortmund
{name.surname}@cs.tu-dortmund.de

ABSTRACT

We present DIME, an integrated solution for the rigorous model-driven development of sophisticated web applications based on the Dynamic Web Application (DyWA) Framework, that is designed to flexibly integrate features such as high assurance and security. DIME provides a family of Graphical Domain-Specific Languages (GDSL), each of which tailored towards a specific aspect of typical web applications, including persistent entities (i.e., a data model), data retrieval (i.e., search queries), business logic in form of various types of process models, the structure of the user interface, and security. They are modeled on a high level of abstraction in a simplicity-driven fashion that focuses on describing *what* application is sought, instead of *how* the application is realized. The choice of platform, programming language, and frameworks is moved to the corresponding (full) code generator which may be changed without touching the models leading to high assurance systems.

1. INTRODUCTION

The DIME approach is a consequent refinement of the realization of jABC4 [1] for process modeling and DyWA [2] for data modeling empowering prototype-driven application development. In the spirit of its predecessors DIME follows OTA (One Thing Approach) [3] and XMDD (Extreme Model-Driven Design) [4] and puts the application expert (a potential non-programmer) in the center of the development process. Hence, the different aspects of an application are described with the most adequate form of model, respectively. All these models are interdependently connected shaping the one thing in a very formal yet easy to understand and use manner to the extend that it can be one-click-generated to a running product. DIME can be used to realize a wide range of web applications. We are just starting to explore its potential. Central design goals on this journey are simplicity [5] and agility [6] as well as security and quality assurance.

DIME enables user-level development of sophisticated web applications. The user starts with the designs of various graphical models that cover different aspects of the target application. These models form the input for a subsequent product generation step in which the full target application is assembled from a variety of generated files that contain the respective source code. The target of this product generation is the DyWA framework that fosters the prototype-driven web-application development throughout the whole application life-cycle in a truly service-oriented manner [7]. In short, modeling and code generation is done in DIME whereas DyWA supports the product deployment phase, constitutes the actual runtime environment and manages data persistence. Furthermore DyWA explicitly enables and supports continuous evolution, the sense of continuous model-driven engineering [8], in a rigorous manner, which facilitates ensuing iterations through the product re-design, re-generation and re-deployment cycle.

The DIME approach provides the user with both an early prototype of an up-and-running web application from the very beginning of the development process as well as explicit support for product evolution due to the agile nature of version management regarding data handling and persistency by the DyWA framework. Altogether, the approach has the potential to tremendously push development cycles in an agile but consistent manner which even comprises security aspects.

2. MODELING ENVIRONMENT

For the model design phase, DIME provides a family of Graphical Domain-Specific Languages (GDSL), each of which tailored towards a specific aspect of typical web applications. These span *Data models* for the design of domain models, *GUI models* to specify the structure of (re-usable components of) web pages as well as different types of *Process models*, each of which tailored towards specific aspects of a web application's behavior. The apparent relations between each of these aspects is modeled by means of cross-referencing to create hierarchical model structures. For this purpose, DIME provides SIBs (Service Independent Building Blocks) in terms of basic model components that link to existing models or to atomic components. SIBs are essential for the effective realization of the omnipresent concept of model

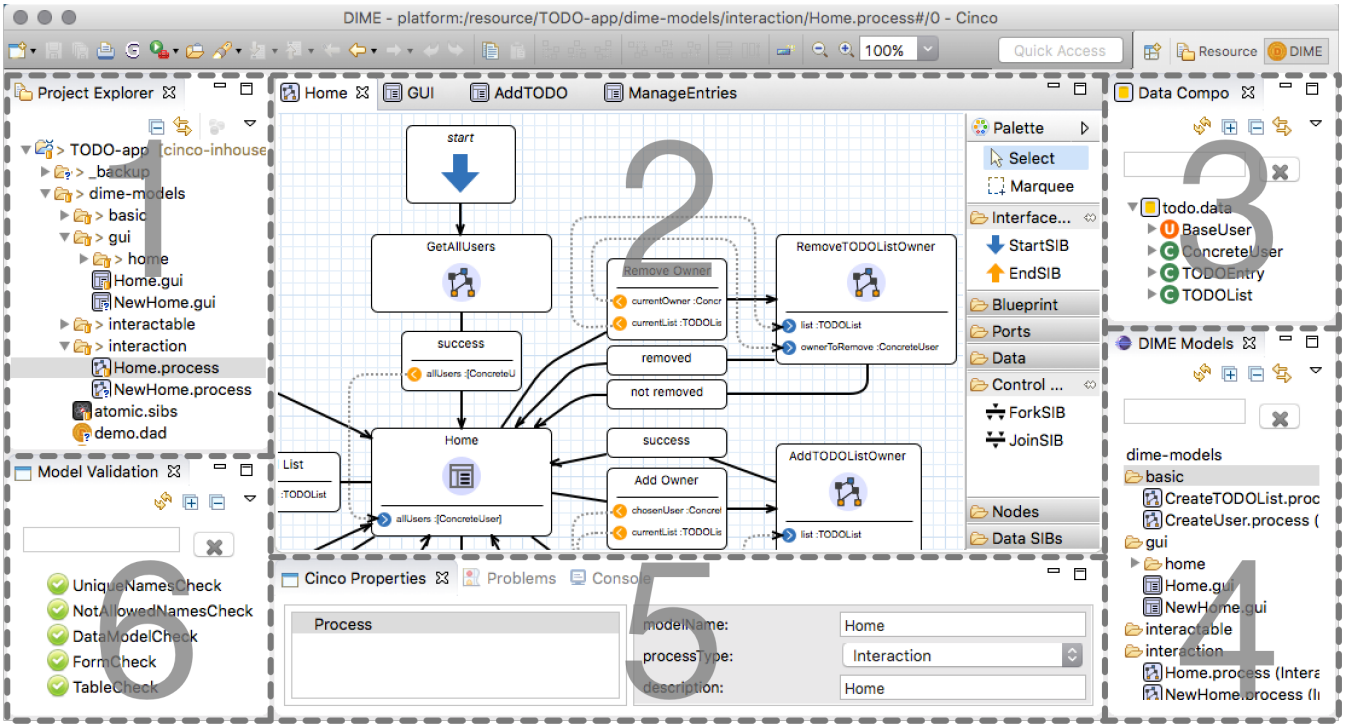

Fig. 1. User interface of DIME with exemplary arrangement of views: (1) Project Explorer. (2) Diagram Editor with Palette. (3) Data View. (4) Models View. (5) Properties View. (6) Model Validation View.

reuse.

The user interface of DIME has been specifically tailored towards supporting the recurrent modeling steps, i.e. to provide guidance for the user by means of quick access to available model entities and relevant properties. The following sections provide a short overview of the DIME user interface as well as more details about available modeling languages and the structural properties of the various models.

2.1. The DIME Application

DIME is a desktop application based on the Eclipse Rich Client Platform (RCP), developed with the CINCO SCCE Meta Tooling Suite [9] that facilitates the development of domain-specific graphical modeling tools in a rigorous model-driven fashion. As RCP application, DIME consists of a set of plugins for the Eclipse framework that provide support for effective model editing and specific views on the current workspace. Models in DIME are foremost graph models formed with nodes and (directed or undirected) edges between them. The provided views take on this inherent structure and provide dense overviews, quick access or other kind of design-relevant information for the user. Fig.1 exemplarily depicts an arrangement of these views constituting the user interface of DIME. Apart from the generic Project Explorer listing files in the workspace, each of the DIME-specific views is shortly introduced in the following.

The **Diagram Editor** in the center of the DIME interface provides the canvas to draw the various graphical models on. Additionally, it provides a palette with basic model components that are specific for the type of the model that is currently opened and shown in the editor.

The **Data View** lists data models in the current workspace. In particular, data types and type attributes are enlisted in a tree-based structure according to their inheritance hierarchy. Data types can be dragged and dropped into special data containers of other models, in order to introduce variables for data exchange between the model components.

The **Models View** provides the user with a dense overview of available models in the current workspace. From this view, a model can be dragged and dropped on the canvas in the diagram editor iff the currently edited model supports cross-references to models of that respective type. This action triggers the creation of a new node inside the currently edited model that holds a reference to the existing model in the workspace. This is the essence of model reuse in the context of DIME.

The **Properties View** provides access to attributes and parameters of nodes and edges inside the currently edited model. This is where parameter values for these entities can be changed by the user. Available attributes and parameters differ depending on the type of the respective model component.

The **Model Validation View** lists the results of syntactic and semantic checks that are applied to the currently edited model. These checks are specifically tailored towards the type of the respective model and dynamically evaluated at model design time. It provides guidance for the user and facilitates correctness of the model, by listing warnings and errors with respect to the affected entity or model substructure. Model validation in DIME spans various aspects comprising, for instance, the enforcement of unique names, the correct use of expression languages, identification of missing edges, and various syntactical requirements.

2.2. Graphical Modeling Languages

Each model type in DIME is a well-defined graphical modeling language that relies on nodes and edges as the basic components of graph models, as well as containers that are special nodes that again can contain other nodes. Graphical modeling is done by means of drag-and-drop operations in interaction with the canvas of the Diagram Editor. Basic model components are dragged from the editor's palette and dropped on the canvas. This triggers the creation of a new node in the current graph model representing an instance of the respective component.

Existing nodes may be connected via edges in a drag-and-drop manner, i.e. clicking on the source node and dragging an intermediate line to the target node. If multiple edge types are suitable for connecting the respective nodes, a selection dialog is shown for the user to select the desired type.

Due to limited space of this article, the syntax and the semantics of the DIME's various model types cannot be described in every detail. In the following the most important concepts in this context are introduced.

Service Independent Building Blocks (SIB) are basic model components in DIME that are essential for the effective realization of the omnipresent concept of model reuse by means of cross-referencing and - as a special case of that - the creation of hierarchical model structures [10]. In essence, SIBs either provide a link to an existing model or to an atomic model component. Atomic in this context means that the component is self-contained and integrated in a service-oriented fashion and not based on models within the current workspace.

The modeling operation to create SIBs as nodes in the currently edited graph model is done by means of drag-and-drop operations in interaction with the canvas of the Diagram Editor. It is basically the same operation as described in the context of basic model components, but the respective model is dragged from the Data View or Models View, depending on its actual type. Dropping it on the canvas triggers the creation of a SIB representing the respective model.

In the following the various model types are discussed briefly.

Data models in DIME allow for the graphical design of domain models based on common data modeling concepts in terms of classes and attributes as well as relations (inheritance as well as uni- or bi-directional associations) between them. The structure of Data models reflects the data structures that are manageable by DyWA, as the latter maintains data objects and provides support for persistence at runtime.

GUI models are used to specify the structure of (re-usable components of) web pages that make up the user interface of the target web application. Hence the structure of GUI models reflects the structure of web pages in order to enable user interface design in a familiar manner.

Process models in DIME allow for the graphical definition of the business logic of the target application. In particular, in DIME exist different types of process models. Each of them is tailored towards specific aspects of a web application's behavior. At design time, it depends on the actual types of the involved models whether cross-references are allowed and how they are handled. On the other hand, the syntax of the different types of process models is nearly the same. The common syntactical features are going to be explained in the course of the discussion of process modeling in Sec. 3.

DAD model. The DyWA Application Descriptor (DAD) model is used to specify the entities that are relevant for the application runtime. A suitable configuration comprises the declaration of relevant domain models, an interaction process that provides the landing page for the target web application and an optional startup process to be invoked when the application is started. This configuration is the entry point for the product generation phase in which source code of the target web application is generated.

3. PROCESS MODELING

Process models in DIME comprise both, a control flow aspect as well as a data flow aspect. In the following the main design concepts regarding each of these aspects are described. In this context, we provide figures that contain models of an ongoing exemplary *TODO-app* application that basically manages lists of TODO entries for its users. The interested reader may find further information as well as a detailed tutorial on the DIME website[1].

3.1. Control Flow

Process models contain a single start node and might have multiple end nodes. In between, the control flow is modeled by means of connecting multiple SIBs via directed control flow edges. In this context, SIBs can not only hold references to other Process models (i.e., Process SIBs) but also to GUI models (i.e., GUI SIBs). While integrating Process SIBs fosters model reuse by means of sub-processes, integrating GUI

[1] http://dime.scce.info

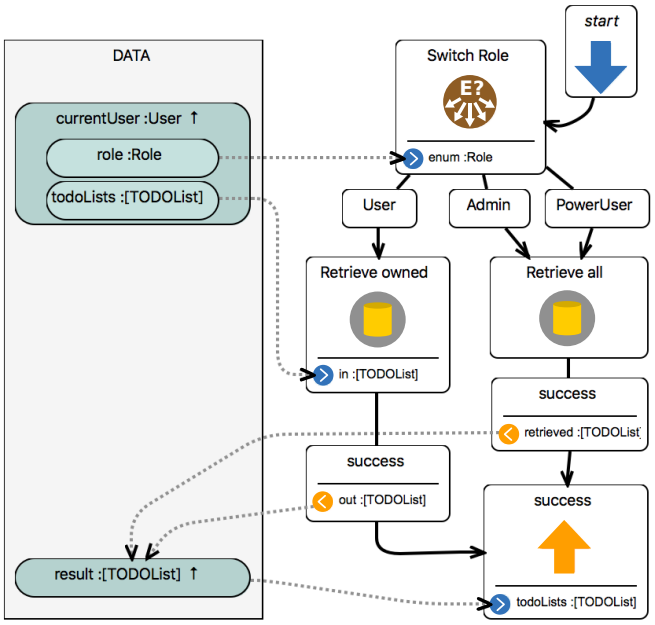

Fig. 2. Process model *GetTODOLists*

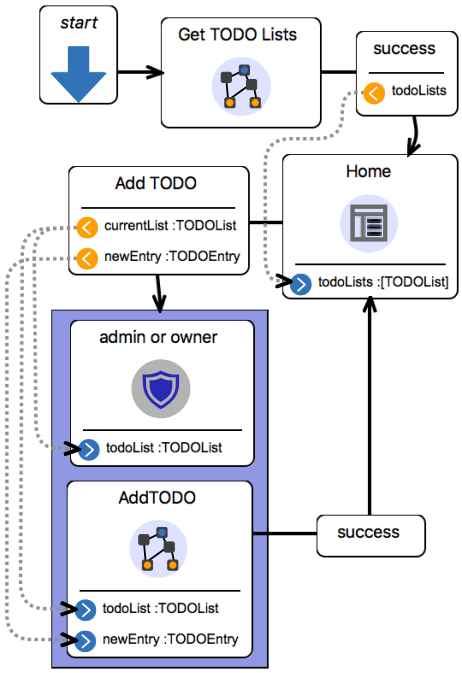

Fig. 3. Process model *AppHome*

SIBs into a process expresses that at runtime throughout the execution of this process an interaction with the user of the application has to take place. In both cases, the subsequent control flow might depend on the actual outcome, be it an execution result of the sub-process or eventual input provided by the user, respectively. In order to reflect this at model design time, the concept of *Branches* is introduced.

SIBs as components of Process models consist of a node and multiple such Branches. While the node represents the actual activity to be executed, each Branch represents one possible outcome of this execution. In particular, the control flow follows only one Branch of a SIB at a time. As an example, Fig. 2 shows a Process model that contains a SIB labeled *Switch Role* with three Branches represented by outgoing edges labeled *User*, *Admin* and *PowerUser*. While in this example the SIB represents an activity that identifies the actual role of the current user of the application, its Branches cover all possible cases. It is also apparent from Fig. 2 that the subsequent control flow depends on the actual Branch taken at runtime. For GUI models each user interaction with the respective web page is interpreted as a branch, e.g., clicking a submit-button in a form or following a hyperlink to another page. As an example, the GUI model depicted at the top of Fig. 5 is integrated into the Process model *AppHome* in Fig. 3 by means of a GUI SIB represented by the node labeled *Home*. Consequently, the button of the form depicted in Fig. 5 bottom is mapped on the single Branch *Add TODO* of the GUI SIB.

As different outcomes of a SIB might convey different provided data, there is also a data flow aspect in the context of Branches to be discussed.

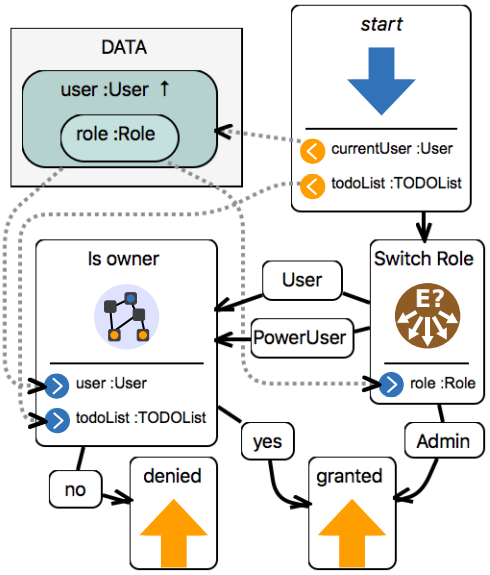

Fig. 4. Security Process model *AdminOrOwner*

3.2. Data Flow

Data flow in the context of Process models too is modeled in a graphical manner. For this purpose, the concepts of *ports* as well as the *data context* are introduced.

For modeling access to the runtime context of the application, Process models provide a specific *DATA* container that holds data nodes representing instances of data types specified in Data models. In the following, this container is referred to as data context of the Process model. In order to enable modeling of the actual data flow, SIBs in the context of Process models can have so-called *Input Ports* while Branches in turn can have *Output Ports*. In Fig 2, each of the *success*-Branches of the retrieval activities *Retrieve owned* and *Retrieve all* have one Output Port representing the retrieved TODO lists. Connecting any of these ports with the data node *result* in the data context via a data flow edge expresses that at runtime the represented data object is provided by the Output Port of the respective Branch. As both the data output of the Branches as well as the data object in the data context are typed as `[TODOList]` (i.e. a list of `TODOList` objects) the data flow edge is valid. In turn, invalid data flow edges are recognized and prevented by the DIME framework. Additionally, Fig. 2 shows how the data object represented by the *result* node is provided as data for the Input Port *todoLists* of the end node *success* of the Process model via a data flow edge. Altogether, the process depicted in Fig. 2 describes the activity of retrieving the TODO lists owned by the current application user, or all TODO lists from a database in case that this user has a role with special privileges.

If Process models are integrated into another model by means of a Process SIB, each of its end nodes is mapped to a separate Branch of this Process SIB. Furthermore, each Branch of the Process SIB would have an Output Port for each Input Port of the corresponding end node. As an example, Fig. 3 shows the Process model *GetTODOLists* as shown in Fig. 2 integrated into the Process model *AppHome* by means of a Process SIB labeled *Get TODO Lists* in the figure. Note that this Process SIB has a Branch *success* according to the end node of the Process model in Fig. 2 and this Branch has an Output Port *todoLists* related to the Input Port of the respective end node with the same name.

Though not depicted in this figure, the same concept applies for Output Ports of start nodes in relation to Input Ports of corresponding Process SIBs. These are referred to as model parameters. The overall approach is directly related to the concept of formal and actual parameters of functions in programming languages.

In the context of GUI SIBs, data objects in the corresponding GUI model are related to Output Ports of the SIBs' Branches. As an example, the data objects connected via edges labeled *Submit* to the button at the bottom of Fig. 5 match the Output Ports of the corresponding Branch *Add TODO* of the GUI SIB in Fig. 3

3.3. Security Guards

Security guards are a concept in DIME to restrict access to models based on a special type of Process models named *Guard Process*. The task of a Guard Process is to decide upon whether the current user of the target application fulfils specific criteria in relation to the respective input of the process. Hence, models of this type follow a predefined structure that requires two end nodes *granted* and *denied*, as well as a model parameter named *currentUser* with respective user type. Fig. 4 shows an example of a Guard Process that realizes the decision upon whether the current user is admin or owner of a TODO list specified as model parameter. The depicted process in particular follows the structural requirements introduced above.

This Guard Process can be used to decide whether the current user is allowed to manipulate a specifc TODO list. Fig. 3 depicts its integration into the *AppHome* process. It is used to restrict the manipulation of a TODO list by means of adding new TODO entries. The Process SIB *AddTODO* is contained in a so-called *Guard Container* together with the Guard Process *AdminOrOwner* to express that the execution of the first needs to be guarded by the latter. The underlying security concept is discussed in the following section.

4. SECURE HIGH ASSURANCE SYSTEMS

There are two different views from which we consider the security and assurance aspects of applications built with DIME: model level and platform level.

4.1. Model Level

We follow a thorough modeling approach with DIME. Its interdependent models of various types are each tailored to the different aspects of a web application. Altogether they build one coherent specification with all the necessary information to generate a fully operational web application in a service-oriented way on any platform chosen arbitrarily from the set of adequate frameworks. Since we obtain a coherent description of our application, the barriers between the layers of an implementation of such an application diminish. Like in aspect-oriented programming we are able to describe a property of our application only once and the analysis during code-generation ensures that this property holds for all layers, beginning with the JavaScript code in the browser, reaching to the business logic and persistence layer in an application server.

This way, error-prone replication of check code to all the layers becomes unnecessary. Therefore, we gain high assurance that the system behaves as expected. In addition, this approach can be permeated to dynamic access control rules leading to secure systems. Of course, the generators may have flaws, but on the one hand the generators can be reused for many different applications, and therefore it is much more

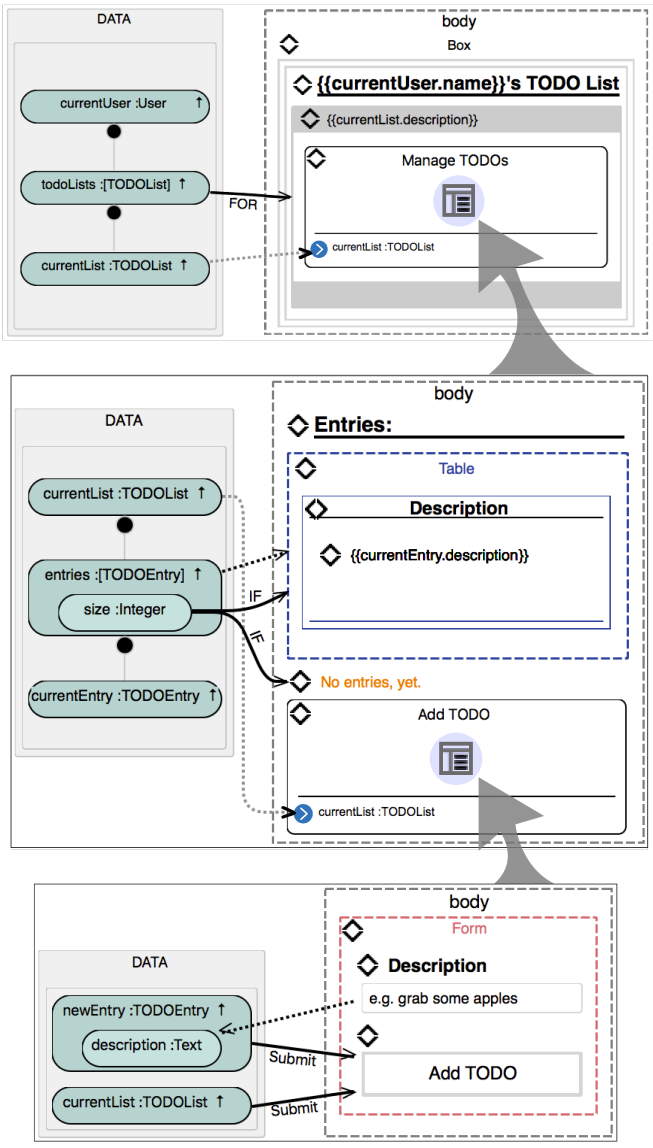

Fig. 5. Hierarchical GUI models of the TODO-app

likely that these flaws are revealed than in a single manual implementation. On the other hand, fixing such an issue can be carried out to all applications built with DIME, just by generating the applications again.

Referring to the TODO-App example application in Fig. 3 the process *AddTODO* is secured with the guard process *admin or owner*. This is the only place where this property has to be set. The analysis during code generation can follow the control flow back to the branch *Add TODO* of the GUI-SIB *Home*, and then traverse the cross-references (see Fig. 5 top to bottom) to the corresponding GUI model *Add TODO GUI* (cf. Fig. 5 bottom). It may then generate a case differentiation for the user interface, which will disable the elements of the corresponding form (see red rectangle with label "Form" in Fig. 5 bottom) or omit it completely, if the guard process

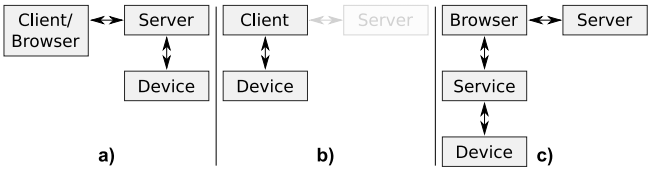

Fig. 6. Variants of server-side encryption via a dedicated hardware device.

evaluates to *denied* for the respective TODO-list. Since the TODO-Lists are rendered in a `FOR`-loop (cf. Fig. 5 top), a backend service can be generated, that evaluates the guard process *admin or owner* for the TODO-lists to be shown all at once and returns which may be edited and which not. Further on, the guard process will be called each time before the process *Add TODO* is executed and the TODO is added only, if the guard evaluates to *granted*. This means, even if someone calls the process directly it is assured, that the access control rules are satisfied.

4.2. Platform Level

Classical client-server architectures often use server-side encryption of content. This may be supported by a hardware device as shown in Fig. 6 a) or done completely in software. Supporting such a solution with high assurance entails the need for some expertise in cryptography in the development team and for each new application there is the potential to make mistakes. Using a generative approach can lower this risk and reduce the responsibility of a development team.

Since the DIME approach is service-oriented, not everything is generated down to the last statement. Instead, a base system (i.e., DyWA) is used, which is a manually implemented full-fledged web application right from the start. Only the data structures, business logic (including access control), and the structure of the user interface are generated on top of modern technologies like *Angular 2*[2] and *Java EE*[3]. The modeling level is completely independent from the environment it is generated to. Hence, it is very easy to integrate support for cryptography on the server side and reuse this for every DIME application.

Furthermore, in the last years the interest to *protect content against the provider of a service* has grown tremendously. For sure, several uncoverings of data privacy violations, by employees of providers, hackers, and even governments contributed to this trend. Several cloud services and instant messaging services therefore introduce *end-to-end encryption*, so that the provider will not be able to access the data – as it would be the case with server-side encryption – neither willingly nor under pressure, even, if it is stored on their servers.

End-to-end encryption has a weak spot: the client. This

[2]https://angular.io
[3]https://www.oracle.com/java/technologies/java-ee.html

can be both the user in front of a device who in general is not an expert in cryptography and the device itself. There are several approaches to increase trustability of devices on the software level [11, 12] as well as on the hardware level [13], leading to architectures as shown in Fig. 6 b).

Securing web applications with a desktop client via end-to-end encryption is already challenging, but using hardware-supported encryption in a web application in combination with a browser client is even harder. This is ironically due to limitations of JavaScript that have been introduced for security reasons in the first place. The JavaScript interpreter lives in a so-called sandbox and is not allowed to (freely) access devices on the local machine like the file system or the hardware security device.

Since we describe only *what* an application is sought to do, we can transparently add a web service to our setup (see Fig. 6 c)) running on the local machine, which is able to encrypt and decrypt arbitrary content using a hardware device like the SEcube [14] and thereby guaranteeing end-to-end encryption of web applications. The code generator then adds (JavaScript) code to the browser client calling this web service to encrypt the content (e.g., the title and description of a TODO entry) before it is sent to the server and to decrypt it before it is presented in the web page to the user. This way, the sandbox limitations are circumvented in a clean and simple way.

The server stores the encrypted content in the database and returns it on demand. There is no need that the server is aware of the encryption. In more complex scenarios with multiple receivers, e.g. a shared TODO-list, a protocol for the exchange of the respective public keys needs to be generated into the application in advance.

The setup of the service and the hardware device can be integrated into the authentication process of the web application. The service itself can be shipped as an executable on a mass storage device integrated into the security hardware, digitally signed with the private key in the hardware itself. The complete communication between the JavaScript client in the browser and both the local encryption service and the server side of the web application will be secured via HTTPS to prevent man-in-the-middle attacks for getting one's hand on the meta data which cannot be end-to-end encrypted, since the server needs to be able to process it. This way, the security of a DIME application can be increased even when accessed from public computers (e.g., in an internet cafe).

5. SUMMARY AND OUTLOOK

We have presented DIME, an integrated solution for the rigorous model-driven development of sophisticated web applications that is designed to flexibly integrate features such as high assurance and security via a family of Graphical Domain-Specific Languages (GDSL), each of which tailored towards a specific aspect of typical web applications, including persistent entities, data retrieval, business logic, the structure of the user interface, dynamic access control, and security. DIME's simplicity-driven modeling approach makes the choice of platform, programming language, and (security) frameworks transparent, by moving them to the underlying (full) code generator which may be changed without touching the models.

6. LITERATURE

[1] Bernhard Steffen et al. "Model-Driven Development with the jABC". In: *HVC 2006, Haifa, Israel*. Vol. 4383. LNCS. Springer, 2007, pp. 92–108.

[2] Johannes Neubauer et al. "Prototype-Driven Development of Web Applications with DyWA". In: *Proc. of 6th ISoLA*. LNCS 8802. Springer, 2014, pp. 56–72.

[3] Tiziana Margaria and Bernhard Steffen. "Business Process Modelling in the jABC: The One-Thing-Approach". In: *Handbook of Research on Business Process Modeling*. IGI Global, 2009.

[4] Tiziana Margaria and Bernhard Steffen. "Service-Orientation: Conquering Complexity with XMDD". In: *Conquering Complexity*. Springer, 2012, pp. 217–236.

[5] Maik Merten and Bernhard Steffen. "Simplicity driven application development". In: *Journal of Integrated Design and Process Science (SDPS)* 16 (2013).

[6] Tiziana Margaria and Bernhard Steffen. "Simplicity as a Driver for Agile Innovation". In: *IEEE Computer* 43.6 (2010), pp. 90–92.

[7] Tiziana Margaria, Bernhard Steffen, and Manfred Reitenspieß. "Service-Oriented Design: The Roots". In: *Proc. of 3rd ICSOC*. Vol. 3826. LNCS. Springer, 2005, pp. 450–464.

[8] Tiziana Margaria and Bernhard Steffen. "Continuous Model-Driven Engineering". In: *IEEE Computer* 42.10 (2009), pp. 106–109.

[9] Stefan Naujokat et al. "CINCO: A Simplicity-Driven Approach to Full Generation of Domain-Specific Graphical Modeling Tools". To appear in STTT (2016).

[10] Bernhard Steffen et al. "Hierarchical Service Definition". In: *Annual Review of COMMUN ACM* 51 (1997), pp. 847–856.

[11] Giorgio Di Natale et al. "Model driven design of crypto primitives and processes". In: *This volume*. 2016.

[12] Roberto Baldoni and Luca Montanari. "Italian National Cyber Security Framework". In: *This volume*. 2016.

[13] Antonio Varriale et al. "SEcube^{TM}: Data at Rest & Data in Motion protection". In: *This volume*. 2016.

[14] Antonio Varriale et al. "SEcube^{TM}: An open security platform: General Approach and Strategies". In: *This volume*. 2016.