IEEE Software
www.computer.org/software

Habitation: A Domain-Specific Language for Home Automation

Manuel Jiménez, Francisca Rosique, Pedro Sánchez, Bárbara Álvarez and Andrés Iborra

Vol. 26, No. 4
July/August 2009

Habitation: A Domain-Specific Language for Home Automation

Manuel Jiménez, Francisca Rosique, Pedro Sánchez, Bárbara Álvarez, and Andrés Iborra, *Technical University of Cartagena*

Combining a domain-specific language with a model-driven approach can enhance the quality and portability of home automation systems.

The appearance of model-driven engineering (MDE)[1] has invigorated research on domain-specific languages (DSLs)[2] and automatic code generation. MDE uses models to build software, thereby displacing source code as the development process's main feature. DSLs provide easy, intuitive descriptions of the system using graphic models. In this new context, DSLs facilitate work in the first design stages. In addition, MDE helps reduce DSL development costs.[3,4] It therefore represents a synergistic union that can significantly improve software development.

MDE lets developers create tools for managing reactive systems much more effectively.[5,6] These tools are more interoperable than traditional tools and easier to maintain owing to certain MDE qualities, such as a higher abstraction level. Home automation is one example of MDE's application in reactive systems. Home automation systems can interact with their environments, offering management of energy, security, communications, and comfort.[7] Such systems are currently developed using low-level procedures and without a methodology that allows platform-independent inclusion of the system requirements. Home automation application developers must therefore have a high level of specialization. In addition, only minimal reuse of artifacts is possible.

Specific languages that allow platform-independent capture of requirements are practically nonexistent in the home automation field. MDE-based proposals for home automation[6,8] use modeling languages (such as UML[9]), which aren't very intuitive and are far removed from the home automation sphere. For example, UML includes hundreds of elements, but only a few of them are directly relevant to software design. Even using profiles, models would be complex (plenty of tags, stereotypes, and so on). Other proposals correspond to platform-dependent commercial tools; the two best-known are Engineering Tool Software (ETS) and LonMaker, which are specific to the KNX/EIB (European Installation Bus) and LonWorks platforms, respectively.

In light of all this, we introduce Habitation (derived from *development of* h*ome* a*utomation* a*pplications using a model-driven approach*), a new methodology to tackle the complete life cycle of home automation system design. Habitation combines a model-driven approach with DSLs to support these applications' definition. We also offer a platform- and technology-independent graphical tool that uses domain-specific abstractions.

0740-7459/09/$25.00 © 2009 IEEE

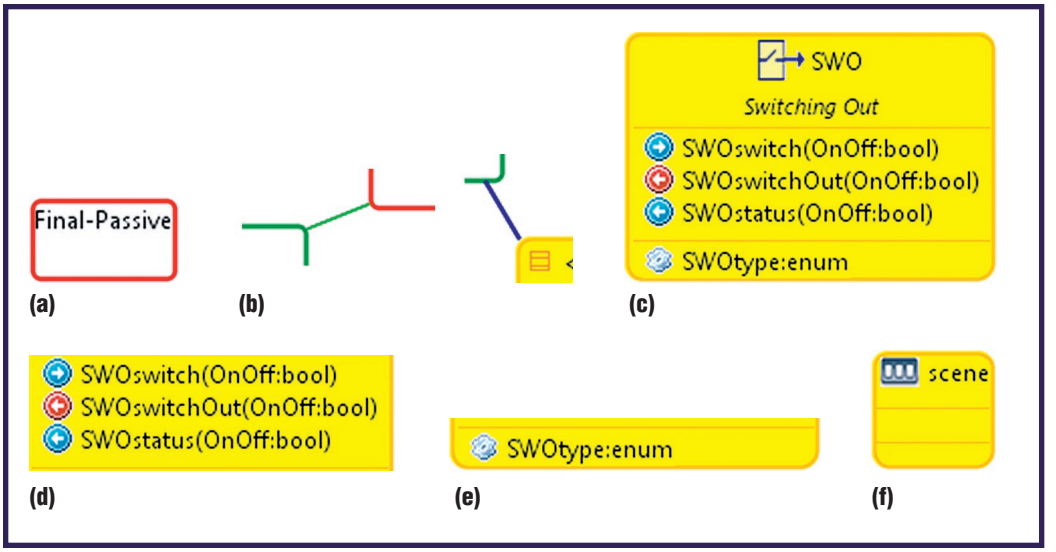


Figure 1. A graphic display of the catalog view. This view's elements include (a) category; (b) links between categories (green) and between functional units and categories (blue); (c) functional unit; (d) services, with the signs indicating whether the service is provided or required (right or left arrow) and whether it's a hardware or software service (red or blue); (e) parameter definition; and (f) scene.

The Home Automation DSL

Home automation application developers mainly use a software tool provided by the device manufacturer (in the case of proprietary systems) or the associations that support the technology (in the case of standardized systems). These tools are usually platform-dependent, code-generation-oriented, integrated environments that do little to raise the abstraction level. Moreover, the concrete syntax they use is rarely intuitive, so users require specialized training and can work only in the solution's immediate context.

The entire application development process is performed by a domain expert who collates the customer's requirements for an installation (elements to be integrated, services required, selection of a concrete technology, and so on) on the basis of the expert's own experience. This expert deploys the devices and then programs them (using a platform-specific tool) to achieve the desired functionality. This manner of working makes it difficult to achieve some of the desired attributes of software systems, such as interoperability, flexibility, reuse, and productivity.

To resolve these shortcomings, we combine a specific visual language with an MDE approach. Our main objective in defining this language is to let designers describe home automation systems using only domain concepts. In this sense, our DSL facilitates the requirements-specification phase visually and intuitively. So, the first constraint is to provide a visual language that's concise and common to the different platforms.

Any home automation system incorporates several elements (*functional units*) that are in all the technologies and standards proper to the domain. These employ different architectures and protocols, but they're identical in capability. To encourage reuse of these functional units and to avoid repeatedly defining the same unit for each application (including several times in a single application), we used a catalog of reusable elements. Once you define such a catalog, you can use it in any application. Functional units have some services through which they can interact with other units. Many of these services are repeated among the functional units, so we created a catalog of services with service definitions that we can reuse in any functional unit.

For this reason, we differentiate two approaches to DSLs. In one, the DSL's purpose is to develop applications, and the user is a developer who might be familiar with the field but isn't necessarily an expert. In the other, the purpose is to develop and implement possible catalog upgrades, and the user should be an expert in the field.

The Catalog View

This view lets the home automation expert model the catalog of functional units and services that developers will later use to create home automation applications. Figure 1 shows the main primitives for modeling a catalog:

- A *category* is a specialization of a catalog element.
- *Links* can be between categories or between functional units and categories.
- A *functional unit* is the smallest element into which a home automation device can be divided. It includes an icon, a name, and the services provided or implemented.
- A *service definition* has a signature that includes the service name and its arguments. The service sign indicates whether the service is

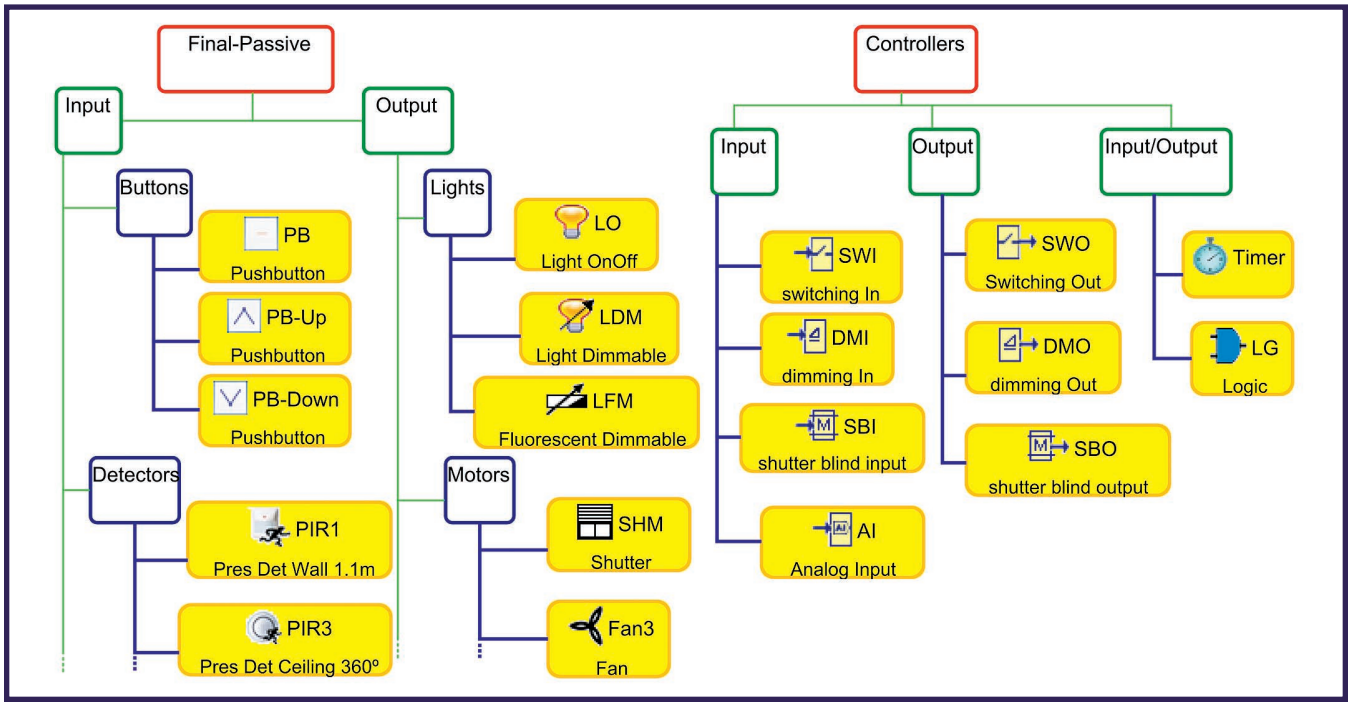


Figure 2. A catalog of functional units and their categories. Final-Passive functional units represent nonprogrammable elements; Controller functional units represent programmable elements.

provided or required (right or left arrow) and whether it's a hardware or software service (red or blue). A service catalog serves as a compartment containing the service definitions.

- A *parameter definition* indicates a functional unit's parameters.
- A *scene* is a specialization of the functional unit. We define the steps that constitute a scene later.

These elements are available in the developed tool's palette.

Figure 2 shows a snapshot of a catalog, which includes categories and functional units that any home automation application developer would use. We'll upgrade and enlarge this catalog by incorporating new functional-unit definitions inside the existing categories. The Final-Passive functional units represent unprogrammable elements (for example, lights and push buttons). The Controller functional units represent programmable elements.

The Application View

This view is used by the developer who designs new applications, who needn't be a home automation expert. The developer can use the catalog to specify an application using these primitives:

- Developers configure *instances of functional units*, which are defined in the catalog, by adding the necessary values to their parameters.
- The *links between functional units* indicate, through services, how these units will interact with the rest of the system. The links can act as channels when a functional unit involved is passive (as such, it's modeled as a hardware-level connection) or as a normal link when neither unit is passive.
- Developers can use *scenes* to configure the sequential execution of several services from functional units within a single action. For example, a developer could define a "Presentation" scene using three steps: lower the blinds, dim the lights, and lower the projection screen. The user could push a button to trigger this scene.

Figure 3 shows these elements, which are included in the developed tool's palette.

Model-Driven Methodology and Tools

Our methodology uses the Object Management Group's model-driven architecture (MDA),[10] which organizes software development in three layers:

- a *computation-independent model* (CIM), which in our case represents the syntax and part of the semantics of the defined DSL;
- a *platform-independent model* (PIM), which in our methodology is a simplification of the UML metamodel for reactive systems,[5] and consid-

ers components, activities, and state chart diagrams; and
- a *platform-specific model* (PSM), for which we've defined a metamodel for the KNX/EIB home automation technology. This metamodel considers the domain object model used by ETS.

In the CIM layer (see Figure 4), the developer elicits requirements through the DSL. Models from this level are automatically transformed into architectural components in the PIM layer. Our tool then transforms the components into executable PSMs for each platform.

This methodology requires that we use the DSL in the first development phase (CIM) so that the user can interact easily with the tool, relying on the methodology's underlying precision. The PIM level is a junction point for different reactive systems (wireless sensor networks, robotic systems, artificial vision, and so on). Consequently, the elements of home automation systems designed in this manner can be integrated as components of a more complex reactive system.

The tool we developed to support our methodology uses the Eclipse (www.eclipse.org) development environment. Eclipse provides a working framework in which the user can manage models. It incorporates various MDE-related projects, making it possible to perform modeling, model transformation, verification, graphic environment generation, code generation, and other such tasks.

The Eclipse Modeling Framework is a plug-in in the Eclipse development environment. EMF lets you create model editors and supplies the basis for interoperability with other tools. We used the Eclipse Graphical Modeling Framework (GMF), which automatically generates graphic editors as Eclipse plug-ins from models.

Our DSL tool has three parts:

- a drawing area in which to build graphic models for the catalog and applications,
- a graphic palette containing the elements (see Figures 1 and 3) that can be dragged to the drawing area, and
- an area in which the available properties (attributes, parameters, and so on) are displayed and can be modified for the selected element.

The tool, which lets us create logical models that describe applications in terms of functional units and links between their services, is now fully operational. A demonstration of its use in our case study example is available at http://hdl.handle.net/10317/854. The CIM metamodel supports an additional floor-plan view, but its implementation in the tool is still under development.

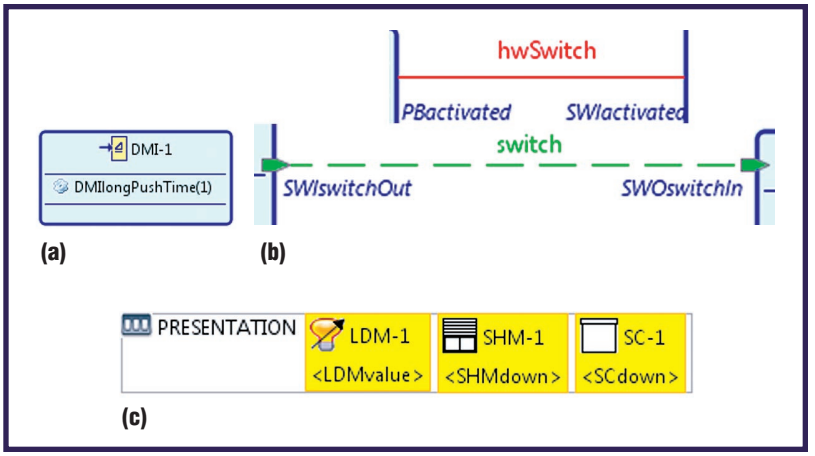


Figure 3. A graphic display of the application view. (a) An instance of a functional unit includes the unit's parameters and their values. (b) Links between functional units can act as channels (red line) or other link types (discontinuous green line with end points). In each link, the top and bottom labels indicate services that participate in the link; the center label is the service's definition. (c) Scenes contain the steps to be performed. A step shows the service to be performed and the icon of the functional unit to which it belongs.

The transformations between the CIM and PIM layers are completely defined using a graph-grammar-based approach[11]—in particular, the EMF Model Transformation (EMT) plug-in.[12] Because models are usually represented by graphs, graph grammar is more attractive than other approaches. For instance, transformation rules expressed through graphs are easier to understand and trace.

Transformation is expressed with rules. Each rule has a left-hand side (LHS) and a right-hand side (RHS), both of which are graphs. A rule might also have a negative application condition (NAC), which must not be satisfied to apply it. To apply a rule to a host graph (the graph to be transformed), a subgraph isomorphism from the LHS to the host graph must exist. After the application, there must be a subgraph isomorphism from the RHS to the result graph.

Consider the model-to-model transformation (from DSL to PIM) in Figure 5. Figure 5a represents a push button (PB-1) that switches a light (LO-1) on and off. Elements SWI-1 and SWO-1 symbolize the controllers providing the desired functionality. Figure 5b shows a graph transformation rule. The rule states that when a service (LHS) is found, it must be transformed into ports, interfaces, and services of the target component model (RHS). However, this rule isn't applied if the transformation has

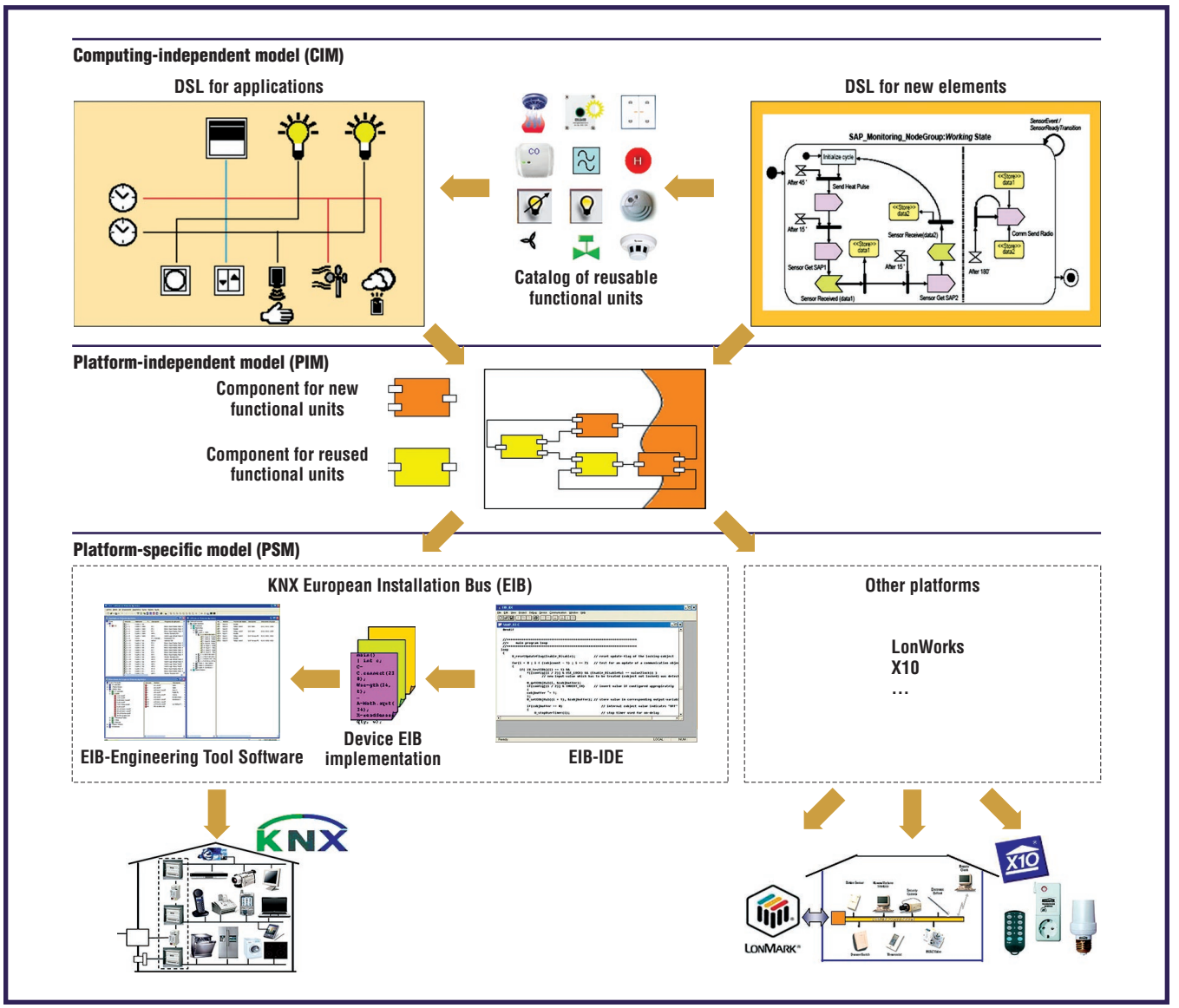


Figure 4. The proposed methodology for developing home automation applications. The CIM level captures user requirements using the defined domain-specific languages (DSLs). The PIM level considers a UML-like component model. At the PSM level, we provide models for different specific platforms and several code generation strategies.

already been performed (NAC). Applying all the rules results in a component model (see Figure 5c). A complete description of the CIM-to-PIM graph transformation rules and the considered metamodels is available elsewhere.[13]

To generate code, the developer must first select a target platform. Doing this involves two key considerations:

- The technology must be supported by international standards.
- The tools for programming the devices must be available and able to be interfaced externally.

The two leading home automation technologies—KNX and LonWorks—fulfill these requirements. Because our research group has wide experience in KNX, we selected this technology for the first platform-specific infrastructure. Currently, the rules for transforming PIM models into PSM models (conforming to the defined KNX/EIB metamodel) are informally defined. We're working to formalize these rules using the graph-grammar notation. PSM models are independent of specific commercial tools and serve as a source for model-to-text transformations. To implement these transformations, we chose the Java Emitter Template tool (JET; www.eclipse.org/modeling/m2t/?project=jet) and the ITTools plug-in. This plug-in lets us interface with the manufacturer

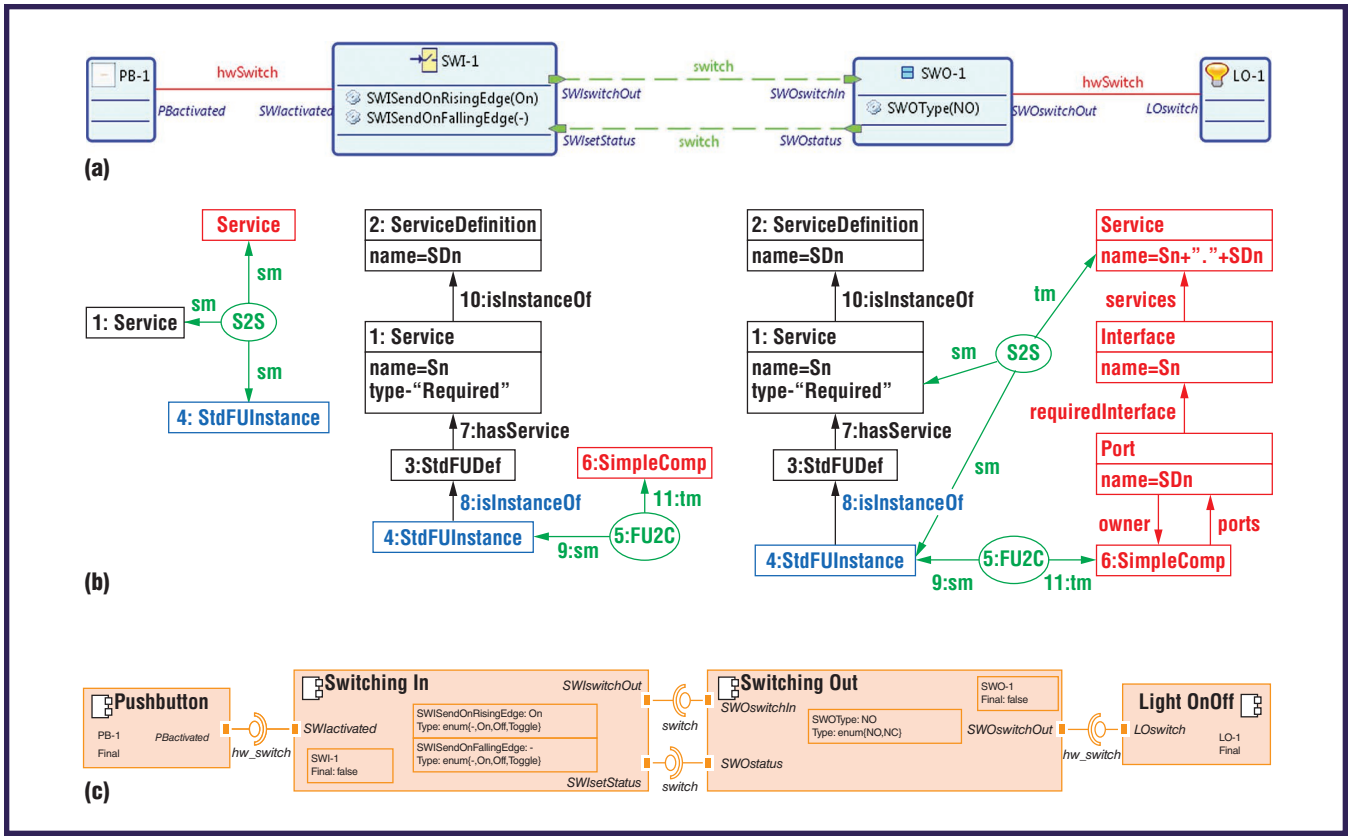


Figure 5. Obtaining a component diagram from a DSL model. (a) A DSL example with the source model at the CIM level. (b) A graph transformation rule for a transform from the CIM to the PIM level. Black indicates a catalog view (DSL) instance, red indicates a target metamodel (PIM) instance, blue indicates an application view (DSL) instance, and green indicates a transformation metamodel instance. (c) A component model diagram with the target model at the PIM level.

environment (ETS) using the VBScript programming language. In this way, we promote reuse of platform-specific tools.

Case Study

Our sample case study involves a system that controls and manages a meeting room used for meetings, seminars, and presentations of various kinds (see Figure 6).[13] This case study has let us validate the DSL's functionality in a real application and establish a starting point from which to apply our methodology. It aims to achieve various objectives regarding energy, security, comfort, and communications.

We used the DSL application view to formally display the system requirements. The complete specification is given by the model's graphical view plus the parameterization of the corresponding properties. Links between functional units establish a kind of activity diagram that starts from the events triggered at the input units. Parameters set up the controllers' internal behavior.

Figure 6 shows the application model for lighting management, including the deployment layout and some correspondences with the DSL elements. Two push buttons control six lighting points in the room. One push button (PB-1) switches and dims the lights next to the meeting room window (LDM-1 to LDM-4). Another push button (PB-2) controls the lights next to the door (LDM-5 and LDM-6). To model this behavior, we connect each push button to a dimming input controller (DMI-1 and DMI-2) using a channel link (red lines) that binds required (PBactivated) and provided (DMI-activated) services (both services must be instances of the same service definition). At the same time, lights are linked to their controllers (DMO-1 to DMO-6), which switch and dim the lights. Finally, controller services are associated with logical links (dashed green lines).

To achieve energy saving and comfort, the system activates a power-off function when it detects no presence in the room after five minutes. Presence detectors (PIR3-1 and PIR3-2) are interconnected through channels to the functional units acting as controllers (SWI-1 and SWI-2). These

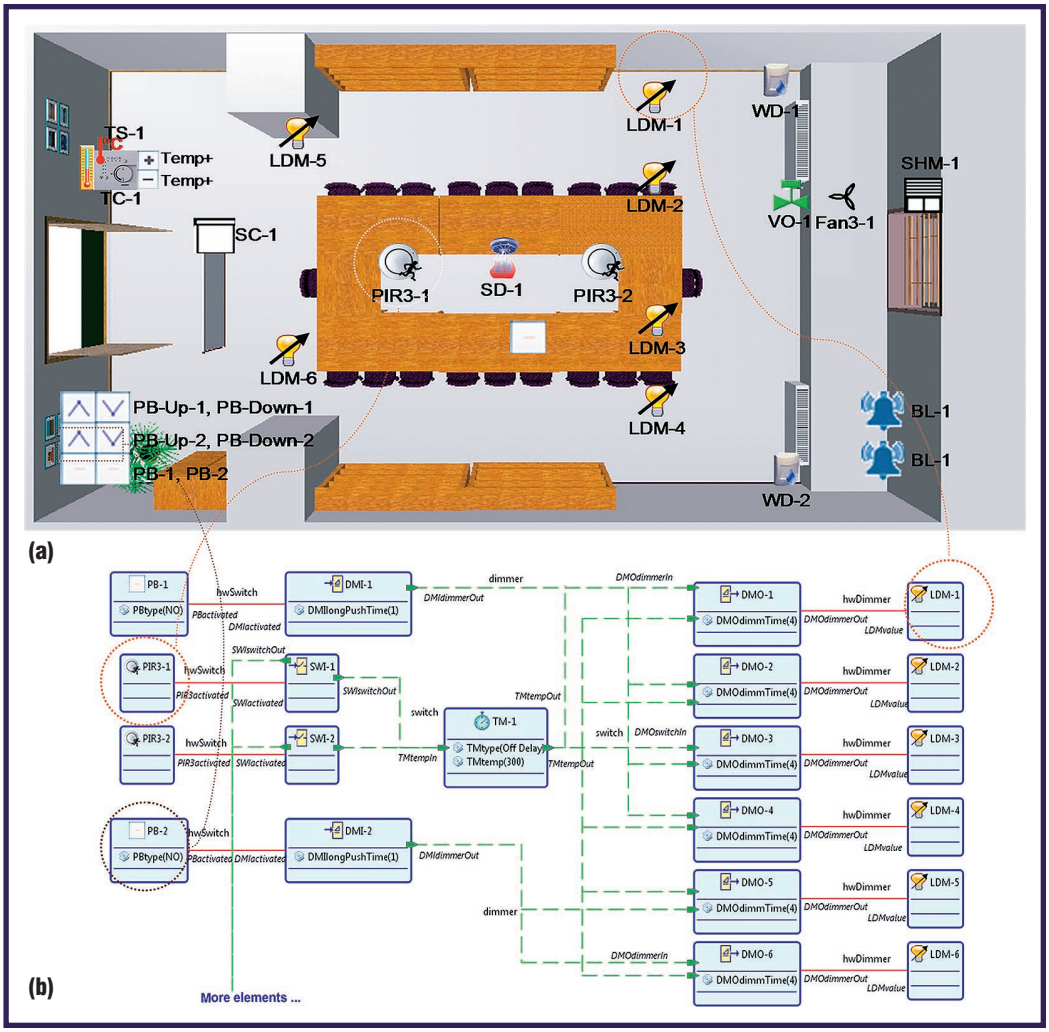


Figure 6. The home automation system in our case study: (a) floor plan and element layout and (b) lighting model in the Habitation DSL. Dashed lines show correspondences between the physical devices and the Final-Passive functional units. The remaining DSL elements are integrated either as part of these devices or as separate controllers.

controllers call the TMtempIn service from the timer (TM-1) every time the system detects a presence. So, the timer switches the lights on when the system detects a presence and switches them off after 300 seconds of detecting no presence. It's possible to use presence detectors simultaneously for both lighting and security management. The functional-unit icons suggest their meanings. Execution platforms (KNX/EIB, LonWorks, and so on) usually integrate controllers (in the case study example, DMI-1, DMO-1, and so on) into devices following a specific criterion. So, Figure 6 doesn't include correspondences between these controllers and floor-plan devices.

Evaluation

Software products should be evaluated for each relevant quality factor using widely accepted metrics.[14] To validate our proposal and our DSL's possible benefits, we conducted an experiment involving a group of students in an electronic-engineering master's course on home automation. We offered participants, none of whom had previous knowledge of home automation technology, three training sessions before beginning the evaluation. The first involved training in the home automation field. The two subsequent sessions provided training in the use of a commercial tool and in the use of the DSL. We then presented participants with a case study in which they were to use both tools.

The experiment mainly concerned usability under specific conditions.[15] It evaluated six usability quality factors: ease of understanding, ease of learning, operability, flexibility, accordance, and attractiveness. For each quality factor, we asked participants to perform an action using the tools and then complete a questionnaire rating their experience (using a 1- to 5-point scale, where 5 is the highest quality rating). We also tracked the time needed to complete the actions and to respond to the questionnaire. We obtained final valuations for each factor using an arithmetic mean of the results of each questionnaire. Table 1 lists the results.

Table 1

Usability results from our case study

Quality factor	Home automation DSL			Commercial tool		
	Score (1 to 5)	Time needed (min.)		Score (1 to 5)	Time needed (min.)	
		Action	Questionnaire		Action	Questionnaire
Ease of understanding	3.6	20.0	5	2.4	24.6	4
Ease of learning	4.2	30.0	3	2.0	90.0	8
Operability	3.6	20.0	6	3.0	25.0	3
Flexibility	2.8	36.8	6	3.6	23.4	4
Accordance	3.8	12.2	5	3.4	10.0	3
Attractiveness	4.0	10.0	3	2.0	20.0	4

Study participants mostly rated the DSL tool higher than the commercial tool (ETS). The exception was flexibility. The largest differences were in ease of learning, ease of understanding, and attractiveness. For example, to questions such as, "Do you need help to remember the concepts represented by each primitive in the palette/tool bar?" most students answered "not at all" (score 5) in the DSL questionnaire. Fewer students did so for the commercial tool.

We're completing the code-generation implementation for ETS. We're also working to integrate other home automation platforms and advanced capabilities, such as requirements traceability, in the process.

Incorporating the GMF plug-in offers multiple possibilities for managing models. However, the training time required to effectively use it is long. With this in mind, we're exploring other modeling tools that require less training time, such as MetaEdit+, which offers a fully integrated modeling, metamodeling, and code generation environment.[16]

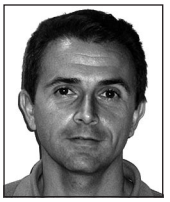
Acknowledgments

The Spanish Interministerial Commission of Science and Technology's MEDWSA (a conceptual and technological framework for the development of reactive software systems) project (TIN2006-15175-C05-02) and the Technical University of Cartagena partially supported this work.

References

1. B. Selic, "The Pragmatics of Model-Driven Development," *IEEE Software*, vol. 20, no. 5, 2003, pp. 46–51.
2. M. Mernik et al., "When and How to Develop Domain-Specific Languages," *ACM Computing Surveys*, vol. 37, no. 4, 2005, pp. 316–344.

About the Authors

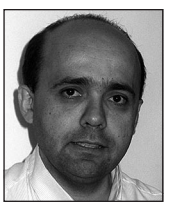


Manuel Jiménez is an associate professor of industrial electronics at the Technical University of Cartagena and is a member of the university's DSIE (Division of Systems and Electronic Engineering) research group. His research interests include electronics and model-driven engineering for reactive systems. Jiménez has a PhD in computer science from the Technical University of Cartagena. Contact him at manual.jimenez@upct.es.

Francisca Rosique is an assistant professor and a PhD student in computer science at the Technical University of Cartagena and a member of the university's DSIE (Division of Systems and Electronic Engineering) research group. Her research interests include model-driven engineering and home automation systems. Rosique has a master's in telecommunication engineering from the Technical University of Cartagena. Contact her at paqui.rosique@upct.es.

Pedro Sánchez is an associate professor of computer science at the Technical University of Cartagena and a member of the university's DSIE (Division of Systems and Electronic Engineering) research group. His research interests include model-driven engineering for real-time systems. Sánchez has a PhD in computer science from the Technical University of Valencia. Contact him at pedro.sanchez@upct.es.

Bárbara Álvarez is an associate professor in computer science at the Technical University of Cartagena and a member of the university's DSIE (Division of Systems and Electronic Engineering) research group. Her research interests include real-time systems and software architectures for teleoperation. Alvarez has a PhD in telecommunication engineering from the Technical University of Madrid. Contact her at balvarez@upct.es.



Andrés Iborra is full professor and head of the Electronics Technology Department at the Technical University of Cartagena and a member of the university's DSIE (Division of Systems and Electronic Engineering) research group. His research interests include computer vision and robotics. Iborra has a PhD in industrial engineering from the Technical University of Madrid. Contact him at andres.iborra@upct.es.

3. T. Clark et al., *Applied Metamodeling: A Foundation for Language Driven Development*, Ceteva, 2008; http://itcentre.tvu.ac.uk/~clark/Publications.html.
4. J.P. Tolvanen and S. Kelly, "Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences," *Proc. 9th Int'l Conf. Software Product Lines* (SPLC 05), LNCS 3714, Springer, 2005, pp. 198–209.
5. D. Alonso et al., "V3Studio: A Component-Based Architecture Modeling Language," *Proc. 15th Ann. IEEE Int'l Conf. Workshop Eng. Computer-Based Systems* (ECBS 08), IEEE Press, 2008, pp. 346–355.
6. M. Voelter and I. Groher, "Product Line Implementation Using Aspect-Oriented and Model-Driven Software Development," *Proc. 11th Int'l Software Product Line Conf.* (SPLC 07), IEEE CS Press, 2007, pp. 233–242.
7. J.L. Ryan, "Home Automation," *Electronics & Comm. Eng. J.*, vol. 1, no. 4, 1989, pp. 185–192.
8. E. Serral et al., "A Model Driven Development Method for Developing Context-Aware Pervasive Systems," *Proc. 5th Int'l Conf. Ubiquitous Intelligence Computing* (UIC 08), LNCS 5061, Springer, 2008, pp. 662–676.
9. *Unified Modeling Language (UML) Specification v2.1.2*, Object Management Group, Nov. 2007; www.omg.org/spec/UML/2.1.2.
10. S. Mellor et al., *MDA Distilled*, Addison-Wesley Professional, 2004.
11. G. Rozenberg, *Handbook of Graph Grammars and Computing by Graph Transformation*, World Scientific, 1997.
12. E. Biermann et al., "Tiger EMF Model Transformation Framework (EMT)," 2006; http://user.cs.tu-berlin.de/~emftrans/papers/userdoc.pdf.
13. M. Jiménez, "Development of Home Automation Applications Following a Model Driven Approach," PhD thesis, Electronics Technology Dept., Tech. Univ. of Cartagena, 2009; http://hdl.handle.net/10317/846 (in Spanish).
14. C. Wohlin and P. Runeson, *Experimentation in Software Engineering: An Introduction*, Springer, 2000.
15. A. Seffah et al., "Usability Measurement and Metrics: A Consolidated Model," *Software Quality J.*, vol. 14, no. 2, 2006, pp. 159–178.
16. J.P. Tolvanen and M. Rossi, "MetaEdit+: Defining and Using Domain-Specific Modeling Languages and Code Generators," *Proc. 18th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 03), ACM Press, 2003, pp. 92–93.