A Model-Driven Engineering Framework for Component Models Interoperability*

Ivica Crnković[1], Ivano Malavolta[2], and Henry Muccini[2]

[1] Mälardalen University, Mälardalen Real-Time Research Center
[2] University of L'Aquila, Dipartimento di Informatica
ivica.crnkovic@mdh.se, ivano.malavolta@univaq.it, muccini@univaq.it

Abstract. A multitude of component models exist today, characterized by slightly different conceptual architectural elements, focusing on a specific operational domain, covering different phases of component life-cycle, or supporting analysis of different quality attributes. When dealing with different variants of products and in evolution of systems, there is a need for transformation of system models from one component model to another one. However, it is not obvious that different component models can accurately exchange models, due to their differences in concepts and semantics. This paper demonstrate an approach to achieve that. The paper proposes a generic framework to interchange models among component models. The framework, named **DUALLY** allows for tool and notations interpretability easing the transformation among many different component models. It is automated inside the Eclipse framework, and fully-extensible. The **DUALLY** approach is applied to two different component models for real-time embedded systems and observations are reported.

1 Introduction

A multitude of component models exist today [1]. While they share the same objectives and some common foundational concepts, they all have some specific characteristics that make them different in many ways. For example, the Progress component model (ProCom) [2] supports the engineering process of embedded systems including quality aspects such as response and execution time, while the Palladio Component Model (PalladioCM) [3] contains annotations for predicting software performance. As a result, a proliferation of component models can be noticed today, each characterized by slightly different conceptual architectural elements, different syntax or semantics, focusing on a specific operational domain, or only suitable for specific analysis techniques.

While having domain- or analysis-specific component models allows component-based engineers to focus on specific needs, interchange [1] among component models becomes limited. Many factors, instead, demonstrate the need of interchange. European

* This work has been partly supported by the national FIRB Project ART DECO (Adaptive InfRasTructures for DECentralized Organizations), the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS, and EU FP7 Q-ImPrESS project.

[1] Terminology: to be able to transform application models specified in a modelling language (i.e. a component model) to another modelling language (i.e. another component model) we need to enable interchange between these component models, i.e. interoperability between them.

G.A. Lewis, I. Poernomo, and C. Hofmeister (Eds.): CBSE 2009, LNCS 5582, pp. 36–53, 2009.

projects like Q-Impress[2] or PROGRESS[3] aim to predict performance, reliability, and other quality attributes by integrating analysis features available in existing component models (Klaper, SOFA, PalladioCM, and ProCom in Q-Impress, SaveCCM and ProCom in PROGRESS). Artifacts interchange is required to propagate results and feedbacks from one notation/tool to another. Different component models cover different phases of component life-cycle. An example of such case is the use of SaveCCM and JavaBeans component models; SaveCCM supports modelling of time-related properties, but has different implementations; one of them is achieved by transformation of an application model specified in SaveCCM to JavaBeans, which then are implemented in Java running on a Java platform [4]. Another example of need for interchange is a migration from one component model to another due to technology change. An example, later discussed, is a transformation from SaveCCM to a new generation, ProCom. Interoperability between SaveCCM and Procom models enables reuse of both design models (and their associated modeling tools) and of different quality attributes.

This work focusses on how to tackle interoperability from a model-driven engineering (MDE) perspective. Specifically, considering component models as meta-models (thus without focusing on components' implementation) allows us to apply model transformation techniques to translate one model (conforming to a component model) to something equivalent conforming to a different component model.

Purpose of this work is to show how **D**U**AL**LY [5], a framework for multipoint-to-multipoint transformations initially devised for allowing interoperability among architecture description languages, can be utilized in the context of component models. **D**U**AL**LY transforms a source model into its target by passing through a pivot meta-model (named A_0 in Figure 1) enabling a star architecture of transformations. Order of n transformations (between the reference model and the pivot) are thus sufficient for transforming n notations, compared to the order of n^2 transformations required in traditional point-to-point ad hoc model transformations. While **D**U**AL**LY scales better than traditional point-to-point approaches, the accuracy of transformations is in general lowered by passing through the pivot meta-model. In order to investigate the degree of accuracy we can achieve when transforming a component model into a different one, and to analyze cons' and pros' in using **D**U**AL**LY compared to specialized point-to-point model transformations, we have applied the **D**U**AL**LY approach to automatically build up transformations between two component models (namely, SaveCCM and ProCom, introduced in Section 2) and executed them on specific models.

In the following of this paper, after having provided an overview on **D**U**AL**LY, we focus on the specific component models that exemplify evolution of component models, and we show how **D**U**AL**LY can facilitate their interchange. For this purpose, a demonstration of the approach is illustrated in Section 3: it will show how **D**U**AL**LY semi-automatically provides the means to exchange models between SaveCCM and ProCom and how it may scale to other component models (thanks to its extensibility mechanisms). Considerations are provided in Section 4, while Section 5 discusses related work. Section 6 concludes the paper with final remarks and suggestions for future work.

[2] http://www.q-impress.eu/Q-ImPrESS/CMS/index_html
[3] http://www.mrtc.mdh.se/progress/

2 Technologies Overview

This section introduces **DUALLY**, the generic framework for notations interoperability (Section 2.1), then focusses on two component models for Embedded Real-time systems: the SaveComp Component Model (Section 2.2) and ProCom (Section 2.3).

2.1 The DUALLY Framework

DUALLY provides a mechanism to automatically generate transformations allowing to pass from a notation to another and vice-versa. The configuration of the **DUALLY** framework is depicted in Figure 1.

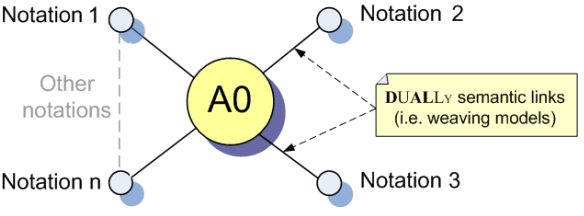

Fig. 1. Notations topology in **DUALLY**

Conceptually, **DUALLY** is a multipoint-to-multipoint transformation framework that permits transformation of elements of a model M1 into semantically equivalent elements in the model M2 (as shown in Figure 2). Each Mi conforms to its MMi that is a meta-model or a UML profile. The semantic mappings (and its corresponding generated transformation) relates MM1 to MM2 (as well as M1 to M2) passing through what we refer to as A_0.

The main purpose of A_0 is to provide a centralized set of semantic elements with respect to which relations must be defined. As clearly shown in Figure 1, for the realization of **DUALLY** we chose a "star" architecture: A_0 is placed in the center of the star, while the transformation engine is in charge of maintaining the transformation

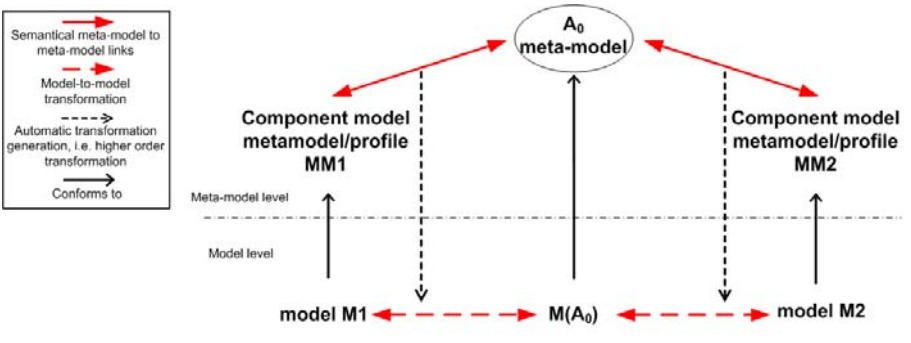

Fig. 2. **DUALLY** Conceptual overview

network. Whenever a model M1 has to be transformed into M2, a two-step process is realized that transforms M1 into M(A_0), and successively M(A_0) to M2. Thanks to the depicted star architecture a linear relationship between the selected language and A_0 is created, thus reducing the number of connections needed. While the star architecture clearly reduces the number of transformations needed, what may happen is that two notations, say M_i and M_j, share some domain specific concepts that are not contemplated in A_0. In general cases, this would strongly reduce the transformation accuracy, since those common concepts could not be transformed (since missing in A_0). However, since **DUALLY**'s A_0 can be extended, accuracy can be improved by including domain specific concepts (an extension to A_0 will be shown in Section 3). A_0 is implemented as a MOF compliant meta-model whose main elements are:

- **`SoftwareArchitecture:`** A collection of components and connectors instantiated in a configuration. It may contain also a set of architectural types, a Behavior and `SAinterfaces` representing points of interaction between the external environment and the architecture being modeled.
- **`SAcomponent:`** A (hierarchically structured) unit of computation with internal state and well-defined interface. It can contain a behavioral model and interacts with other architectural elements either directly or through `SAinterfaces`.
- **`SAconnector:`** Represents a software connector containing communication and coordination facilities. It may be considered as a special kind of `SAcomponents`.
- **`SAinterface:`** Specifies the interaction point between an `SAcomponent` or an `SAconnector` and its environment. It is semantically close to the concept of UML port and can have either input, output or input/output direction.
- **`SArelationship:`** Its purpose is that of delineating general relations between A_0 architectural elements; it can be either bidirectional or unidirectional.
- **`SAchannel:`** A specialization of an `SArelationship` representing a generic communication mean; it supports both unidirectional and bi-directional communication, and both information and control can be exchanged.
- **`SAbinding:`** Relates an `SAinterface` of a component to an `SAinterface` of one of its inner components; it is semantically close to the concept of UML Delegation Connector.
- **`SAtype, SAstructuredType:`** They define architectural types, so any architectural element can potentially be an instance of a particular `SAtype`. Each `SAtype` can contain a set of properties and the behavior of its instances, its internal structure is specified in case it is also a `SAstructuredType`.
- **`Behavior:`** Represents the behavior an architectural element. It is an abstract meta-class that plays the role of a "stub" for possible extensions of the meta-model representing dynamic aspects of the system.
- **`Development:`** Represents the direct relation between the architectural and the technological aspects, such as the process that will be used to develop the system, or the programming languages used to develop a certain component. `Development` is an abstract meta-class and its realization is left to future extensions of the A_0 meta-model.
- **`Business:`** An abstract meta-class to be specialized via future extensions of A_0 and represents the link to business contexts within which software systems and development organizations exist.

– **`Group:`** A logical grouping of any element of A_0, it can contain architectural elements, properties, other groups and so on.

The A_0 meta-model contains other minor concepts like properties, abstract typed elements, generic components; we do not describe them in this work because of their basic nature and for the sake of brevity. For more details about A_0 we kindly refer the reader to [5].

Figure 2 shows that **DUALLY** operates at two levels of abstraction: meta-modeling and modeling. At the meta-modeling level, MDE experts provide the concepts of each notation through either a meta-model or a UML profile. Then, a set of semantic links between $MM1$ (or $MM2$) and the corresponding elements in A_0 are defined. Such links are contained in a weaving model [6], a particular kind of model containing links between either meta-models or models.

Each weaving model (WM, in Figure 3) conforms to a specific weaving meta-model WMM provided by **DUALLY** [5]. It's main elements are: correspondences (`Left2Right`, `Right2Left` and `Equivalence`) to relate two or more concepts, feature equivalences to relate attributes or references, bindings to a constant defined by the user, links to a woven element, and the various auxiliary elements referring the meta-classes to relate. **DUALLY** provides a graphical editor for weaving models identification and a mechanism to automatically generate model-to-model transformations that reflect the logic of the semantic links.

Figure 3 highlights also that model-to-model transformations are generated through the execution of higher-order transformations (HOTs): *Left2Right* and *Right2Left*, depending on the direction of the transformation to be generated at the modeling level (i.e. *MM1_2_A0* or *A0_2_MM1*). The input of a HOT is composed of three models: (i) the weaving model WM, (ii) the left meta-model and (iii) the right meta-model (MM1 and A_0 in Figure 3, respectively). The output is a model transformation generated on

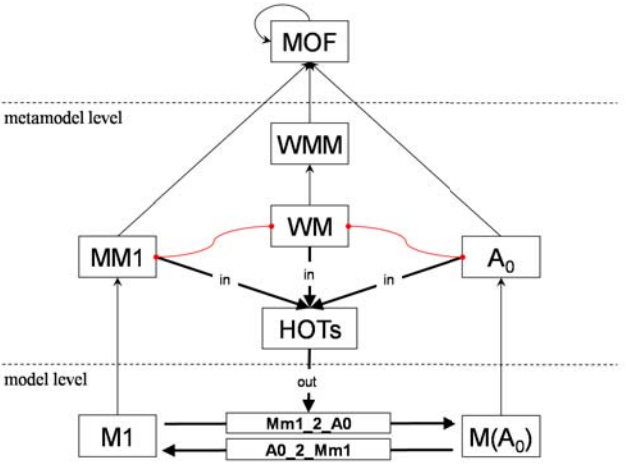

Fig. 3. Higher-order transformations in **DUALLY**

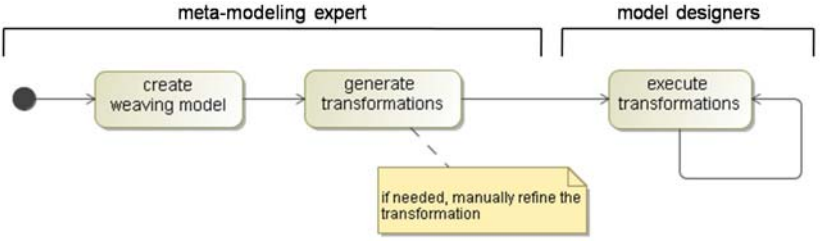

Fig. 4. **DUALLY**-zation process for each meta-model

the basis of the mappings defined into the weaving model. At a high level of abstraction, the current version of **DUALLY** translates each correspondence into an specific transformation rule and each feature equivalence into a binding between the involved attributes or references. The higher-order transformations are meta-model and A_0 independent; this gives the possibility to reuse **DUALLY** with different pivot meta-models, i.e. to reuse its approach in different domains.

At the modeling level, system designers create the model $M1$ and execute the previously generated transformations $t1$ and $t2$. This produces first the intermediate model $M(A_0)$ and then the final model $M2$ in the target notation.

Figure 4 shows the basic steps to **DUALLY**-ze a notation, i.e. to include it in the star topology implied by **DUALLY**. **DUALLY** provides a clear separation between its main users:

- meta-modeling experts play the role of technical stakeholders, they have to know the language to relate and semantically link it to A_0; they can also refine the generated transformations if there is the need for more advanced constructs within them;
- component-based systems designers play the role of final users, they deal with models only and apply the transformations automatically generated by **DUALLY**.

The preliminary steps of determining the pivot meta-model, or in case the extension of A_0, are not part of this process because they are not performed for each **DUALLY**-zation, but rather once for each specific domain the framework is being used in.

DUALLY is implemented as an Eclipse plugin in the context of the ATLAS Model Management Architecture (AMMA) [6]. Weaving models are expressed through the ATLAS model weaver (AMW) [7] and transformations in the Atlas Transformation Language (ATL [8]). Both models and meta-models are expressed via the XML Metadata Interchange (XMI), the interchange standard proposed by the OMG consortium.

2.2 SaveCCM

The SaveComp Component Model (SaveCCM) aims for design of software for embedded systems with constraints on the system resources such as memory and CPU, and requirements of time characteristics such as execution or response time. The graphical notation of the SaveCCM component model is presented in Figure 5.

SaveCCM contains three architectural elements: components, switches and assemblies. The interface of these elements consists of input- and output ports. The model is

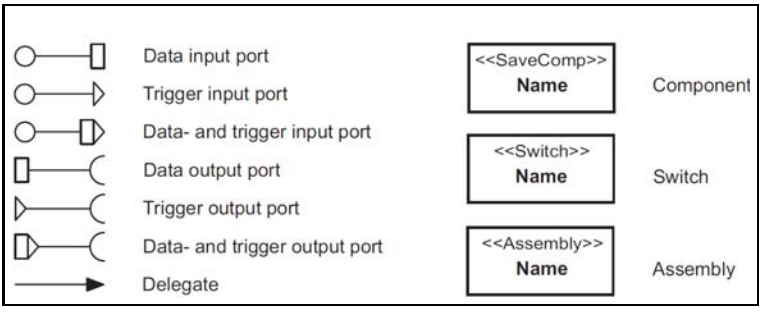

Fig. 5. The graphical notation of SaveCCM

based on the control flow (pipes-and-filters) paradigm, and on the distinction between data transfer and control flow. The former is captured by connections between *data ports* where data of a given type can be written and read, and the latter by *trigger ports* that control the activation of components. A port can also comprise both triggering and data functionality. The explicit notation of control flow makes the design analyzable with respect to temporal behaviour allowing analysis of schedulability, response time, etc., factors which are crucial to the correctness of real-time systems.

Components are the main architectural element in SaveCCM. In addition to an interface a component contains a series of quality attributes (e.g. worst case execution time, reliability estimates), each associated with a value and possibly a confidence measure. Quality attributes are used for analysis, model extraction and for synthesis. Components have a strict"read-execute-write" semantics that ensures once a component is triggered, the execution is functionally independent of any concurrent activity. This facilitates analysis since component execution can be abstracted by a single transfer function from input values and internal state to output values. The switch construct provides the means to change the components interconnection structure, either statically or dynamically. switches specify a number of guarded connection patterns, i.e., partial mappings from input to output ports. To enable increased reusability and design efficiency SaveCCM also defines assemblies. An assembly can be considered as a means for naming a collection of components and hiding its internal structure.

SaveCCM enables time and resource analysis using techniques are the basis in the context of real-time systems and the current version of the SaveCCM development Environment includes two analyzers: LTSA (Labelled Transition System Analyzer) [9] and Times Tool [10].

2.3 ProCom

ProCom [2] is the component model designed as a "new generation of SaveCCM" with extended functionality and somewhat different philosophy. While SaveCCM is designed for small embedded systems, ProCom is aimed for design and modelling of distributed embedded systems of larger complexity. ProCom enables distinguishing of two layers called ProSys and ProSave. The former deals with elements of coarse granularity, like subsystems and channels; the latter contains concepts to internally describe a

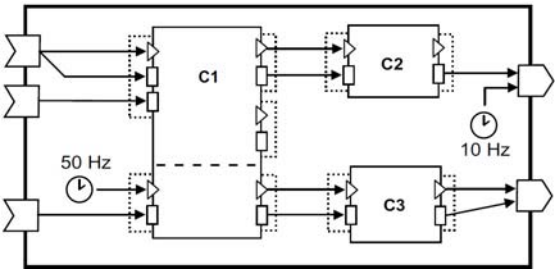

Fig. 6. A subsystem internally modelled by ProSave

ProSys system as a collection of concurrent components. A ProSys subsystem is specified by its input/output message ports, and its external view can optionally include attributes and auxiliary models. A subsystem is always active since it may perform activities periodically or in response to internal events. Subsystems are not directly connected, but rather they communicate through message-channels, which are specific elements for sharing information between senders and receivers of each message.

ProSave concerns the internal design of subsystems through its contained elements. Components in ProSave differ from ProSys subsystems since they are passive and the communication is based on the pipes-and-filters paradigm as in SaveCCM. Similar to SaveCCM, this component model explicitly separates data and control flow that are captured by data and trigger ports respectively. A new feature, compared to SaveCCM, is that ports can be grouped into services. They are part of the component and allow external entities to make use of the component functionality at an higher level of abstraction. Services are triggered independently and can run concurrently. ProSave does not have switches like SaveCCM, but a rich set of predefined connectors that provide more elaborate constructs, such as data/control fork, join, data muxer and demuxer.

ProSave elements are used to decompose ProSys subsystems (as exemplified in Fig. 6).This is is done in a similar way to how composite ProSave components are defined internally (i.e., as a collection of interconnected components and connectors) but with some additional connector types to allow for (i) mapping between message passing (used in ProSys) and trigger/data communication (used in ProSave), and (ii) specifying periodic activation of ProSave components. This is provided by the *clock* connector that repeatedly triggers ProSave elements at a given rate.

3 Demonstration of the Approach

This section shows how we apply the **DUALLY** approach to SaveCCM and ProCom modeling languages. Since both notations have common concepts and **DUALLY**'s A_0 covers many of them, A_0 is still suitable as the pivot meta-model. However, there are also concepts that are contemplated in both SaveCCM and ProCom but not in A_0. This is not a desirable situation, because such concepts will not be preserved during transformation due to A_0's lack of them. The extensible structure of A_0 allows to overcome this issue: we extended A_0 in order to capture such elements an avoid their loss while

Table 1. Elements extending the A_0 pivot meta-model

Added element	A_0 **base element**	**Description**
`System`	`SAcomponent`	Coarse-grained independent component with complex functionality.
`DataSAinterface`	`SAinterface`	Specific interface for data transfer to which typed data may be read or written.
`ControlSAinterface`	`SAinterface`	Specific interface for control flow handling the activation of components.
`Clock`	`SAcomponent`	Component triggering other elements of the system when its given period expires.

passing from either SaveCCM and ProCom to A_0. We discuss this choice and its possible pros and cons in Section 4. Table 1 presents such an extension.

The line of reasoning we followed to create such extension is purely pragmatic: if (i) there are two elements x and y in SaveCCM and ProCom respectively that represent the same semantic concept z, and (ii) there is not a corresponding element in A_0, then extend A_0 with z. During the creation of weaving models, both x and y will be linked to z so that the execution of the generated transformations will preserve z while passing through A_0. An example of a common semantic concept is the data-transfer interface represented by `DataSAinterface` in A_0 (second row in table 1), it corresponds to the concepts of SaveCCM and ProCom `DataPort`.

The remaining of this section is organized so as to reflect the two roles of **DUALLY**'s users. In Subsection 3.1 we act as meta-modeling experts: for each language we follow the steps of the activity diagram in Figure 4 until the generation of model-to-model transformations. In Subsection 3.2 we act as software architects, i.e. we provide an example model conforming to SaveCCM and we apply the transformations generated from the previous steps. This allows us to get first a model conforming to the A_0 meta-model and then the final model conforming to the ProCom meta-model. We evaluate and compare the various models obtained within each phase of the process.

3.1 Semantic Links Definition

The first step is to import the SaveCCM and ProCom meta-models into the **DUALLY** framework; in our case it is straightforward since such meta-models have been developed in the context of Eclipse. Next step is the creation of the weaving models. Since a weaving model defines the links between a notation and A_0, we need two weaving models: $SaveCCM\_A_0$, which contains the semantic links between SaveCCM and A_0 and $ProCom\_A_0$ that relates ProCom concepts to A_0 elements. Due to space restrictions, in this paper we describe each weaving model in an informal way, abstracting from the technical details and presenting only its basic semantics.

In $SaveCCM\_A_0$ a SaveCCM `System` is mapped into an A_0 `System`. If the source `System` element is the root of the model, an A_0 SoftwareArchitecture element is also created (it will contain all the A_0 target elements). SaveCCM `Component` and `Connection` are mapped to `SAComponent` and `SAchannel`, respectively.

`Assembly` and `Composite` components are both mapped to `SAcomponent` annotating the type of its corresponding source element in the *description* field. By doing this we will not lose which kind of element the `SAcomponent` was generated from during the translation to an A_0 model. A SaveCCM `Clock` corresponds to the `Clock` entity specified in the extension of A_0, the `Delay` element is mapped into a generic `SAcomponent`, `Switch` is mapped into an `SAConnector`. `DataPort` and `TriggerPort` correspond to `DataSainterface` and `ControlSAinterface` respectively, their *direction* attribute is set accordingly to the type of the SaveCCM ports (i.e. whether they are input or output ports) and vice-versa. We applied a specific mechanism to manage combined SaveCCM ports: a combined port is splitted to a `DataSainterface` and a `ControlSAinterface` with the same name and direction; doing this, the generated $SaveCCM2A_0$ transformation splits combined SaveCCM ports to two A_0 interfaces, while the $A_02SaveCCM$ transformation merges two A_0 interfaces with the same name and direction to a single combined SaveCCM port. Both generic and SaveCCM-specific attributes (e.g. "delay" and "precision" attributes in `Delay`) correspond to A_0 properties.

$ProCom\_A_0$ contains links between concepts of A_0 ProCom at both system (i.e. ProSys) and component (i.e. ProSave) levels. At the system level, `Subsystems` correspond to `SAtypes` and `SubsystemInstances` are related to A_0 `Systems`. ProSys `Connections` are mapped to `SAChannels`, while `MessagePorts` correspond to generic `SAPorts`. The `MessageChannel` entity is linked to `SAconnector`. At the component level, `CompositeComponent` and `PrimitiveComponent` are mapped to `SAtype`. The concept of `SubcomponentInstance` is linked to the concept of `SAcomponent`; `Sensor`, `ActiveSensor` and `Actuator` also relate to `SAcomponent` along with the corresponding real-time attributes, in this cases the type of the source ProCom entity is annotated in the *description* attribute of the generated `SAcomponent`. A_0 `Clocks` are mapped to ProCom `Clocks`, and the corresponding period attribute is set. Each kind of ProCom port is mapped into $SAPort$, while the corresponding `Services` and port groupings are not matched because both A_0 and SaveCCM do not have entities semantically close to them; such elements are lost during the translation to A_0 and can be restored when "going back" to ProCom thanks to **DUALLY**'s lost-in-translation mechanism [5].Furthermore, $ProSave$ `Connections` are mapped into `SAchannels` and every kind of `Connector` (e.g. `DataFork`, `ControlJoin`) corresponds to an `SAConnector`; the specific type of the $ProSave$ `Connector` can be inferred by analysing its own ports and connections pattern, e.g. a connector with one input data port and n output data ports can be considered a `DataFork` connector, a connector with n input control ports and a single output control port is a `ControlJoin` connector, etc. A number of low-level correspondences have been abstracted to make the description of the weaving models as readable as possible and complete at the same time. Most of them are contained into the $ProCom\_A_0$ weaving model; for example there is a system of correspondences and conditions to correctly arrange $ProSave$ Components and their internal realization or the mechanism to infer the type of $ProSave$ Connectors sketched above.

The next step is the execution of the **DUALLY** HOTs in order to get the model-to-model transformations. We generate two transformations from each weaving model:

(i) $Save2A_0$ and A_02Save from the weaving model between SaveCCM and A_0; (ii) $ProCom2A_0$ and $A_02ProCom$ from the $ProCom\_A_0$ weaving model. The logic of the generated transformations reflects that of the links in the weaving models. Since obtained transformations suit well with the models that we use, at the moment there is no need to refine them, i.e. the generated transformations are ready to be used by software architects.

3.2 Model Transformations Execution

After the generation of model transformations, designers can use them to translate models into other notations. Generated transformations can be executed in any order, in this work we focus only on how to produce ProCom models from SaveCCM specifications; this process is composed of three main phases:

1. development of a model conforming to SaveCCM;
2. execution of the $Save2A_0$ transformation that produces an intermediate A_0 model;
3. execution the $A_02ProCom$ transformation in order to translate the A_0 model into a ProCom specification.

The initial SaveCCM model is a specification of an Adaptive Cruise Controller (ACC), an enhanced version of the traditional vehicular Cruise Control. It has the basic function to actuate brake or throttle controls in order to keep the speed of the vehicle constant; it is geared also with a set of sensors (mainly radars and cameras) that allow the controller to (i) check if there is another vehicle in the lane, allowing the driver to maintain a safe distance with the preceding vehicle and (ii) check the presence of speed road signs and adapt the actual speed according to them. Figure 7 presents a simplified version of the ACC system developed using Save-IDE [11], the SaveCCM dedicated modeling toolkit.

The ACC system contains two main components, $Sensors$ and $ACC\_application$. The former represents a group of sensors that periodically provides data about the distance towards the preceding vehicle, actual speed, status of the whole controller and the degree of pressure on the brake pedal by the driver. Such sensors have been grouped into a single component in order to leave the model as simple as possible. The core of the system is the latter component, it performs three main tasks: (i) analyze information provided by sensors and return which actions forward to the actuators; (ii) log the status of the system; (iii) provide data to the Human Machine Interface (HMI).

For the sake of simplicity, we do not show the internal structure of $ACC\_application$ and the external context of the whole system. The system is also composed of two clock components that periodically trigger the corresponding components. More specifically, $clk50hz$ triggers $Sensors$ to gain the available current data and $ACC\_application$ to set its status according to the newly available informations. The Splitter switch periodically receives a trigger generated by the clock at 50Hz and splits it triggering simultaneously the $Sensors$ and $ACC\_application$ components. $clk10hz$ triggers $ACC\_application$ to log the status of the systems and to provide data to the HMI; $clk10hz$ operates at a lower rate since it is not related to safety-critical activities.

Once the ACC model has been created, we give it as input to $SaveCCM2A_0$ (the transformation automatically generated by **D**U**AL**L**Y**). The produced model is composed of a main `SoftwareArchitecture` containing all the elements of the ACC

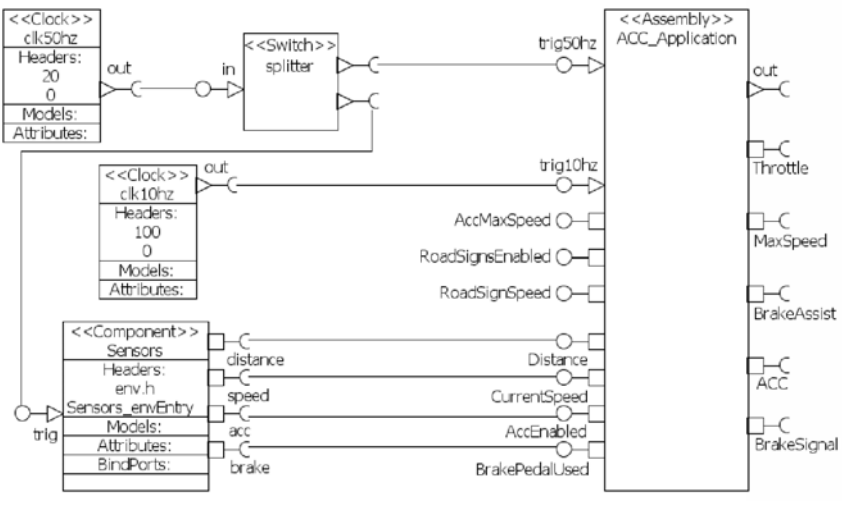

Fig. 7. ACC system designed using SaveCCM

system. Each SaveCCM component has been transformed into an `SAcomponent`, two clock entities have been created from the 10Hz and 50Hz SaveCCM clocks and their *period* attribute is represented as an A_0 `Property`. Data and Trigger ports have been translated into `DataSAinterfaces` and `ControlSAinterface`, respectively. All the $Save_CCM$ `Connections` have been translated into `SAchannels`, while the *Splitter* switch is transformed into an `SAconnector` with a single input `ControlSAinterface` and two output `ControlSAinterfaces`. Attributes specific to the SaveCCM language (e.g. the attribute specifying the implementation in C programming language of $ACC\_application$) are preserved in the A_0 model through generic properties; they will be looked up when passing to the ProCom specification.

At this point, we execute the $A_0 2 ProCom$ transformation on the A_0 model that we just obtained. The produced specification is a complete ProCom model containing both a $ProSys$ and $ProSave$ layer. The $ProSys$ part of the model is made of only the ACC `System` containing its internal $ProSave$ specification (depicted in Figure 8).

The obtained ProCom model, even though we passed through A_0, is similar to the initial SaveCCM specification, e.g. the topography of the two models remains the same. We mapped every `A0channel` to a $ProSave$ `Connection` since the connected elements are only components (in case they were systems, $ProSys$ `Connections` were generated); the two `Clock` components have been translated and their period attribute has been preserved. The thick dot shown in 8 represents a `ControlFork` connector that splits the control flow into two paths. The various data/control ports have been translated and their direction and type are also preserved; the only concern that we did not manage to map is grouping $ProSave$ ports into services. Actually we expected it since there is no concept of service neither in SaveCCM nor in A_0. This is the typical case in which model designers have to manually arrange obtained models. A possible alternative solution to this issue is to generate services applying some heuristics, e.g.

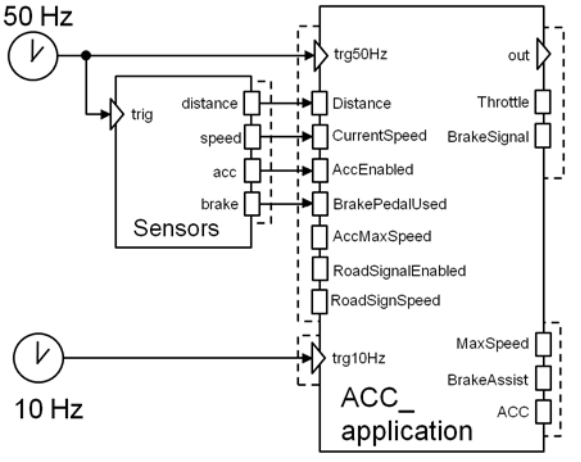

Fig. 8. The ACC specification obtained from the execution of $A_0 2ProCom$

analyze ports using a dictionary or an ontology so that ports whose names represent a common semantics may be grouped into a single service.

In conclusion, in this section we showed how **DUALLY** can be used to pass from a real-time notation (SaveCCM) to another (ProCom) applying a model-driven approach; designers need to specify only semantic links at the meta-modeling level and modellers execute automatically generated transformations to migrate models. The results of such executions strongly depend on the accuracy of the semantic links between meta-models; this and other issues are discussed and evaluated in the next section.

4 Evaluation and Considerations

Purpose of this paper has been to investigate the feasibility in utilizing **DUALLY** as a generic transformation framework in the context of component models and its advantages and limitations in comparison to manual and specialized model-to-model transformations. For this purpose and based on the experience reported in the previous section, we identified a set of generic dimensions to reason about which strategy best suites our needs: i) number of required transformations (`NTrans`) for achieving interchange among n notations, ii) number of additional transformations (`AddTrans`) required when a new notation is added, iii) accuracy (`Acc`) of the transformations results. Other (secondary) dimensions are also analyzed. Table 2 summarizes the main findings.

From the first row, we can see that the point-to-point strategy requires the development of order of n^2 transformations, while the pivot-based solution requires order of n transformations. Since many component models exist in real-time embedded systems, the traditional point-to-point strategy would require possibly too many transformations. Specifically, since our intent is to link several additional notations such as AADL, UML

Table 2. Comparison between point-to-point and **DUALLY**

	point-to-point transformations	**DUALLY**
`NTrans`	n*$(n-1)$	n
`AddTrans`	n	2
`Acc`	semantic loss	semantic loss + pivot inaccuracy
automation	manually coded transformation	automatically generated transformations
steps	one-step transformation	two-step transformation

models using the MARTE profile or Timed Automata, the pivot meta-model proposed in **DUALLY** seems to be more appropriate.

Moreover, the second row points out that adding a further notation to the interoperating network of notations requires n new transformations implemented from scratch in the point-to-point strategy. The pivot meta-model strategy in **DUALLY**, instead, requires only one new transformation between the added notation and A_0. More importantly, the full-mesh strategy requires a deep knowledge of the n existing notations, while the pivot-based solution requires knowledge on A_0 only.

As outlined in the third row, loss of information may happen in both the point-to-point and pivot-based strategies, since the expressiveness of the notations to relate can vary (i.e., if a notation is more expressive than the other, the extra expressiveness cannot be typically mapped). Apart from this, the pivot-based strategy is in principle less accurate than the point-to-point solution, since the (domain-specific but generic) intermediate model can increase the probability to loose concepts during the translation. Even a well designed pivot meta-model might be less accurate than a point-to-point transformation. Heterogeneity between notations to relate exacerbates this issue. In order to limit such an issue, **DUALLY** provides extensibility mechanisms as a way to minimize the loss of information. As shown in Section 3 we created an ad hoc extension including all those elements in SaveCCM and ProCom not contemplated in A_0. Specifically, we added the concepts of *system* (meant as coarse-grained component), *data/control interfaces* and *clock* components. This allowed us to not lose such elements while passing through A_0; this was an undesirable issue because elements would have been lost only because of the pivot meta-model and not for a real mismatch between SaveCCM and ProCom. For example, SaveCCM *data port* would have been mapped to A_0 *SAinterface*, but when passing to ProCom we did not know to which kind of ProCom port we had to translate it; so, extending A_0 allowed us to map SaveCCM *data ports* to ProCom *data ports* passing through A_0's `DataSAinterfaces` without losing their semantics. As illustrated in Section 3, this activity has not required a big effort, due to the similar expressive power and conceptual elements exposed by the two notations and the similarity between the concepts already in A_0 and those in component models. Indeed, when deciding to **DUALLY**ze notations in a completely different domain, extending A_0 might not be enough. We are currently evaluating how to deal with such a situation.

The fourth row points out that while traditional point-to-point transformations require ad hoc and manually coded transformations in some transformation language

Table 3. Efforts to relate SaveCCM and ProCom

	SaveCCM	**ProCom**
import meta-model	0	0
learn meta-model	4 (7,2%)	10 (18,1%)
create weaving model	9 (16,2%)	20 (36,2%)
refine transformation	0	0
develop models	2 (3,6%)	0
	15 (29,8%)	30 (56,1%)

(e.g., ATL [8] or QVT [12]), **DUALLY** automatically generates the transformation code out of the weaving models.

In the fifth row, we put in evidence that **DUALLY** uses a two-steps transformation, while a single-step is required in point-to-point strategies. The main issue about the two strategies is that of accuracy, i.e. passing through a pivot meta-model could lower down the quality of the models produced by the transformations; we managed to overcome this problem extending A_0 so that no common element is lost because of A_0's expressivity. So, in this specific case, extending A_0 and applying the **DUALLY** approach leads to the same degree of accuracy as if a point-to-point strategy is applied. In addition, applying the pivot-based strategy gives us also other benefits in the context of future work; for example it is possible to reuse the **DUALLY**-zations of both SaveCCM and ProCom while relating other notations to A_0, or quantifying how many elements SaveCCM and ProCom have in common with respect to other **DUALLY**-zed notations.

Table 3 quantify the efforts needed to **DUALLY**-ze SaveCCM and ProCom. Each notation has a dedicated column, while the rows describe specific activities performed during our experience. The value of a cell is both represented in terms of person-hour, (i.e. the amount of work carried on by an average worker) and in percentage with respect to the whole process. All the percentage values do not sum to one hundred since we leaved out around eight person-hours to define which topology adopt to relate the notations and to specify which elements form the extension of the A_0 meta-model.

Since each meta-model is already in the XMI format, the effort related to the import phase can be considered equal to zero. The second row represents the time to "learn" each notation, that is how much it took to understand its constructs and manage its models. Table 3 highlights that creating weaving models is the activity that requires much effort, since the resulting weaving model must be very accurate and the generation of well-formed transformations directly depends on it. This specific experiment did not require to manually refine the generated transformations.

The *develop models* row specifies how much time it took to create a model in each notation. The *ProCom* column value is equal to zero because the ProCom model has been automatically derived by executing the $A_0 2 ProCom$ transformation. The efforts associated to the meta-modeling expert represent almost the 80% of the whole process but are performed only once for each notation. Further, **DUALLY** hides most of the complexity to the final users, since they deal with models development and transformations execution only.

5 Related Work

Related work mainly regards model-based tool integration, automatic derivation of model transformations and semantic integration. The authors of [13] present ModelCVS, a framework similar to **D**U**AL**LY in which meta-models are lifted to ontologies and semantic links are defined between such ontologies, they will serve as a basis for the generation of model transformations; this approach manages also concurrent modeling through a CVS versioning tool. **D**U**AL**LY is somewhat different because it implies the A_0-centered star topology (it scales more when dealing with multiple notations) and the preliminary step of meta-model lifting is not performed.

The role of **D**U**AL**LY's A_0 is similar to the Klaper language in the field of performability. Grassi et al. in [14] propose the Klaper modeling language as a pivot meta-model within a star topology; however the Klaper-based methodology is different from **D**U**AL**LY's approach since model transformations are not "horizontal" (Klaper is designed as a mean between design-level and analysis-oriented notations). Moreover model transformations are not derived from semantic bindings, they must be manually developed.

In the field of software architectures, the ACME initiative [15] is famed for being one of the very first technologies to tackle the interoperability problem. It benefits from both a good tooling set and a good level of expressivity, but it is neither MOF compliant nor automatized; further on, a programming effort (rather than graphically designed semantic links) is needed every time a notation must be related to the ACME language.

Finally, the Eclipse project named Model Driven Development integration (MDDi[4]) presents an interesting approach based on the concepts of model bus and semantic bindings, but it is still in a draft proposal state.

6 Conclusion and Future Work

In model-driven engineering transformations of models are crucial activities in the development process. In component-based development, along with increasing support for specification, analysis and verification of quality attributes, and evolution of the component-based systems, transformation models between different component models becomes increasingly important. We have analyzed possibilities of increasing interchange between component models by applying a principle multipoint-to-multipoint transformation engine and using a pivot-metamodel, both implemented in **D**U**AL**LY. Although **D**U**AL**LY was originally designed for transformation of different architecture description languages, we have demonstrated feasibility of this approach. The component models SaveCCM and ProCom, used in our investigations, are examples of evolution of component models; for this reason they have many similar elements, but also elements that are quite different. We have shown that it is possible to find a common core, in spite of these differences. Actually we have demonstrated that the pivot-metamodel expressed in A_0 is sufficient. The interoperability between component models and A_0, depends of course on similarities of the component models. We have addressed the structural part of the architectural interoperability, which does not cover the

[4] http://www.eclipse.org/proposals/eclipse-mddi/

complete interoperability; for example we have not addressed behavioral models, and models of quality attributes. Our future work is focused on (i) adding new component models , and (ii) adding behavioral models. Architectural interoperability is however the most important since it is used as a common reference point for other models both functional and non-functional. Further it is the only part that is included in most of the component models.

So far in the Component-Based Development approach, interoperability between component-based applications have been addressed in research in order to increase reusability of existing components or component-based systems. We have investigated interoperability between models which increases reusability of models, this is crucial for analysis and verification of components and component-based systems.

References

1. Crnkovic, I., Larsson, M. (eds.): Building Reliable Component-based Software Systems. Artech House (2002)
2. Sentilles, S., Vulgarakis, A., Bures, T., Carlson, J., Crnkovic, I.: A component model for control-intensive distributed embedded systems. In: Chaudron, M.R.V., Szyperski, C., Reussner, R. (eds.) CBSE 2008. LNCS, vol. 5282, pp. 310–317. Springer, Heidelberg (2008)
3. Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. Journal of Systems and Software 82, 3–22 (2009)
4. Feljan, J., Carlson, J., Zagar, M.: Realizing a domain specific component model with javabeans. In: 8th Conference on Software Engineering Research and Practice in Sweden (SERPS 2008) (2008)
5. Malavolta, I., Muccini, H., Pelliccione, P., Tamburri, D.A.: Providing architectural languages and tools interoperability through model transformation technologies. IEEE Transactions on Software Engineering (to appear, 2009)
6. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the large and modeling in the small. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 33–46. Springer, Heidelberg (2005)
7. Didonet Del Fabro, M., Bézivin, J., Jouault, F., Breton, E., Gueltas, G.: AMW: a generic model weaver. In: Proc. of 1ére Journée sur l'Ing. Dírigée par les Modéles, Paris, France, pp. 105–114 (2005)
8. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
9. Magee, J., Kramer, J., Giannakopoulou, D.: Behaviour Analysis of Software Architectures. In: First Working IFIP Conference on Software Architecture, WICSA1 (1999)
10. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Times: a tool for schedulability analysis and code generation of real-time systems. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 60–72. Springer, Heidelberg (2004)
11. Sentilles, S., Håkansson, J., Pettersson, P., Crnkovic, I.: Save-ide an integrated development environment for building predictable component-based embedded systems. In: Proc. of the 23rd IEEE/ACM Int. Conf. on Automated Software Engineering (2008)
12. OMG: MOF QVT Final Adopted Specification. Object Modeling Group (2005)
13. Kappel, G., Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W., Schwinger, W., Wimmer, M.: Lifting metamodels to ontologies - a step to the semantic integration of modeling languages. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 528–542. Springer, Heidelberg (2006)

14. Grassi, V., Mirandola, R., Randazzo, E., Sabetta, A.: KLAPER: An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example. LNCS, vol. 5153, pp. 327–356. Springer, Heidelberg (2008)
15. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, pp. 47–68. Cambridge University Press, Cambridge (2000)