Softw Syst Model
DOI 10.1007/s10270-010-0170-z

THEME SECTION

A model-driven approach to automate the propagation of changes among Architecture Description Languages

Romina Eramo · Ivano Malavolta · Henry Muccini · Patrizio Pelliccione · Alfonso Pierantonio

Received: 1 November 2009 / Revised: 30 April 2010 / Accepted: 8 July 2010

Abstract As it is widely recognized, a universal notation accepted by any software architect cannot exist. This caused a proliferation of architecture description languages (ADLs) each focussing on a specific application domain, analysis type, or modelling environment, and with its own specific notations and tools. Therefore, the production of a software architecture description often requires the use of multiple ADLs, each satisfying some stakeholder's concerns. When dealing with multiple notations, suitable techniques are required in order to keep models in a consistent state. Several solutions have been proposed so far but they lack in convergence and scalability. In this paper, we propose a convergent change propagation approach between multiple architectural languages. The approach is generic since it depends neither on the notations to synchronize nor on their corresponding models. It is implemented within the Eclipse modelling framework and we demonstrate its usability and scalability by experimenting it on well known architectural languages.

Keywords Architectural languages interoperability · Model transformation · Model synchronization · Automation · Metamodelling

Communicated by Tony Clark and Jorn Bettin.

R. Eramo · I. Malavolta · H. Muccini · P. Pelliccione (✉) · A. Pierantonio
Dipartimento di Informatica, Università dell'Aquila, Via Vetoio, L'Aquila, Italy
e-mail: patrizio.pelliccione@univaq.it; patrizio.pelliccione@gmail.com

R. Eramo
e-mail: romina.eramo@univaq.it

I. Malavolta
e-mail: ivano.malavolta@univaq.it

H. Muccini
e-mail: henry.muccini@univaq.it

A. Pierantonio
e-mail: alfonso.pierantonio@univaq.it

1 Introduction

Software architectures [35] are today recognized as a crucial means for engineering complex and critical systems since they provide the needed level of abstraction. Architectural descriptions are used for documenting design decisions, for predicting system characteristics before deployment and for driving the coding process. Up to the present day, many architecture description languages (ADLs) and associated tools have been proposed for specifying and analysing software architectures (e.g. [8,12,15,27,32]), each specialized in a specific application domain and characterized by different conceptual architectural elements, syntax, semantics and supported analysis techniques. As remarked in many sources (e.g. [1,9,19,25,30,34,35,37]), the software architecture of a system must be defined taking into account a number of different views for the various uses and stakeholder concerns. Therefore, the most common case in producing an architecture description of a system is to use multiple ADLs, each specialized to one or more concerns. It thus becomes of paramount importance to incorporate descriptions and analyses made with existing ADLs into a coherent framework, enabling the easy addition of new description languages and tools, while enforcing consistency and coherence. This calls for interoperability among architectural descriptions. The need of interoperability at the architectural level is also clearly demonstrated by European

Published online: 31 July 2010

projects like Q-Impress[1] or PROGRESS[2] that aim to predict performance, reliability and other quality attributes by integrating analysis features available in different architectural languages. Artefact interchange is required to propagate results and feedbacks from one notation/tool to another.

A framework for architectural interoperability shall thus not simply relate different models (e.g. by transforming one to another), but keep all those models *synchronized* (thus consistent and coherent) in an effective and automated way. Approaches for architectural descriptions interoperability and change propagation among models have been already proposed (see Sect. 7 for an overview). However, consistently supporting interoperability and change propagation is intrinsically complex and far from being addressed and integrated in a unique framework. Furthermore, the lack of automation does not allow the easy addition of new description languages, and does not guarantee change propagation to multiple models in a finite number of steps.

The goal of this paper was to propose an automated framework for both architectural interoperability and model synchronization that overcomes existing limitations. This framework builds upon two technologies our group developed separately and for different purposes: **D**U**AL**Ly [30], an automated framework that provides interoperability through Model-Driven Engineering (MDE) techniques, and an approach to model transformation and changes propagation based on answer set programming (ASP) [3].

This paper, starting from those two technologies, proposes a *convergent change propagation between multiple architectural languages*, i.e. the proposed (model-driven) approach ensures that when a model has been modified, such modifications are propagated in a finite number of steps to all the other models that are part of the network. It is important to note that the used ASP-based approach for bidirectionality presented in [3] is among the most general ones [18], i.e. it does not require the transformations to be *bijective*, *injective* or *surjective*. This has relevant implications as when from a modified target model it is not possible to reverse the transformation to one or more source models, the engine is anyhow able to deduce a collection of source models that approximate the ideal one from which it is possible to generate the previously modified target. The automated framework proposed in this paper, on the one side enhances **D**U**AL**Ly with change propagation features, and on the other side, enhances the efficiency and efficacy of the ASP-based approach in case of transformations among multiple models.

The main contributions of this work can be summarized as follows: (i) the development of a framework that enables both interoperability and synchronization among architectural models allows for an easy addition of new ADLs and guarantees convergent changes propagation, (ii) tool support within the Eclipse Modelling Framework, (iii) discussion on the generalization of the approach to domains different from the software architecture one.

The paper is structured as follows: Sect. 2 sets the context of the paper through a motivating example that is used throughout the paper to demonstrate the approach. Section 3 presents theoretical aspects of our approach and Sect. 4 describes implementation aspects of the proposed solution. Section 5 applies the approach to a case study. Considerations and discussion on advantages and limitations are reported in Sect. 6. Section 7 compares our approach with both ADLs interoperability and change propagation. The paper closes with conclusions and future work in Sect. 8.

2 Motivating example

This section introduces an illustrative and motivating scenario that will be used as running example along each phase of the approach presented in this paper, and that will be detailed in Sect. 5. In this scenario, a software architect performs the description of the system of interest by making use of various ADLs. As introduced before, this is a very recurrent situation since it is not common to find an ADL that perfectly captures all design decisions judged fundamental by the system's stakeholders [30,31].

More specifically, in our explanatory scenario, a company mainly operates in the vehicular system domain and uses a domain-specific component model, the SaveComp Component Model (SaveCCM) [17], to support the development of resource-efficient systems. Systems in this domain are highly critical for the vehicle's functionality and controlling, and both structural and behavioural aspects need to be carefully assessed before development. While SaveCCM allows for structural preventive analysis, a different tool is required for behavioural analysis. Similarly to what happens in the Save project [24], our company decides to use the verification features of LTSA (Labelled Transition System Analyser) [28] in order to increase the dependability of our product. For doing so, a model in the Darwin/FSP ADL [29] is required[3]. While SaveCCM and Darwin/FSP provide domain- and analysis-specific notations and tools, a more commonly known description language is required for sharing architectural knowledge among the different stakeholders involved in the project. For this purpose, we decide to use a UML profile for component-based architectures, called UML_{cc} [20], and UML tools documentation features, like

[1] http://www.q-impress.eu/wordpress/.

[2] http://www.mrtc.mdh.se/progress.

[3] It has to be noted that while FSP provides behavioural information not included in SaveCCM, Darwin contains structural information to be aligned to the SaveCCM model.

the UML MagicDraw reports[4]. This profile contains mechanisms to specify systems via components, connectors and their behaviour. The profile is described in details in [30].

In this specific scenario, and without suitable technologies we are asked to accurately model the system in SaveCCM, then to re-model the system from scratch in Darwin/FSP in order to be able to perform the desired analysis by means of LTSA. Once obtained a satisfactory model, we have to re-model the system from scratch in a UML tool suitable for the UML_{cc} profile. In addition to the overhead required to re-model the system three different times, this scenario opens several potential problems. First of all we are delegating to modellers the responsibility to write "equivalent" models in each involved language and tool. Moreover, at each step models are refined and possible errors are discovered and solved, the propagation of changes requires by-hand re-modelling. Typically, cost and time requirements will prohibit the different model alignment.

To the best of our knowledge, **DUALLY** [30] is the most mature framework to support interoperability among various ADLs. More precisely, **DUALLY** enables the transformation of a model conforming to a specific ADL into corresponding models conforming to other ADLs. In **DUALLY** the interoperability among various ADLs is ensured via model transformation techniques. Instead of creating a point-to-point relationship among all languages, **DUALLY** defines the transformations among ADLs by passing through A_0, which is a core set of architectural concepts defined as generally as possible (to potentially represent and support any kind of architectural representation) and extensible (in order to add domain specificities). In other words, A_0 acts as a bridge among the different architectural languages to be related together. The star architecture of **DUALLY** enables an agile and easy integration of ADLs. The **DUALLY** transformation system is made of a series of low-level model-to-model transformations that enable information migration among architectural models. These model-to-model transformations are constructed automatically by executing higher-order transformations (i.e. transformations taking other transformations as input or producing other transformations as output). Details about **DUALLY** and A_0 can be found in [30]. Unlikely, also **DUALLY** cannot manage at best the change propagation issue. More precisely, **DUALLY** mechanisms to manage the synchronization among architectural models are limited and not directly related to models' evolution and associated change propagation. The default synchronization method, called *"loss in translation"*, is focussed exclusively on managing information that can be lost during the transformations: being A_0 the bridge among the different architectural languages to be related together, elements that have no relations with elements in A_0 could not be translated. In other words, the goal of the loss in translation mechanism is simply to manage and retrieve such information.

Summarizing, the effective and efficient management of changes propagation among various ADLs is still an open problem.

[4] MagicDraw: http://www.magicdraw.com.

3 Dealing with the propagation of changes among different ADLs

In general, changes occurring in an architectural model have a strong impact on all the other interoperating models (each of them possibly conforming to different ADLs). In order to keep models in a consistent state, changes need to be propagated from the updated model to all the others. When dealing with multiple ADLs, propagating changes may be a complex task; such a task is inevitable and requires to be managed by a dedicated approach.

In this section we illustrate how it is possible to support the propagation of changes among architectural models by exploiting (i) a framework for supporting ADLs interoperability (Sect. 3.1) and (ii) the ASP point-to-point approach for bidirectional model transformations (Sect. 3.2).

3.1 The proposed process of change propagation

While **DUALLY** transforms a model into any other by passing first through an A_0 model, model changes are propagated accordingly first to the A_0 model and successively forwarded to any other architectural model (it has to be noted that the obtained result is independent from the order followed in the forwarding).

Under the assumption that concurrent modifications to different models cannot apply, the **DUALLY** architecture ensures the *convergence* of the change propagation process, that is, it ensures by construction that a modification of a model within the network is propagated to all the other models in a finite number of steps (future work in order to manage concurrent modifications is discussed in Sect. 8).

Figure 1 shows two possible change propagation processes: the change propagation in a generic process and the change propagation as regulated by A_0.

By referring to Fig. 1a (i.e. the generic changes propagation process) once model D is modified to D', the changes must be propagated to A, B, and C. However, as soon as B is modified due to the modification in D, the changes in B must be propagated to A, C, and even D. The same holds for changes caused in A and C. This simple example provides the intuition that the change propagation output strongly depends on the order in which models are updated after a change, and that the propagation process could easily diverge.

By referring to Fig. 1b, the changes propagation regulated by A_0 fixes these problems. Once a model D has been

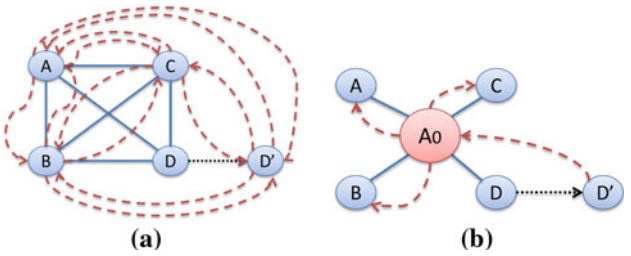


Fig. 1 Changes propagation: generic process (**a**) and through A_0 (**b**)

modified to D', changes are propagated to A_0, and successively from A_0 to A, B, and C, following any order. The changes propagation output is independent from the followed order. This is possible because of the star topology in which notations are arranged: once D's changes are propagated to A_0, the updated A_0 model is considered as "frozen" and the propagation of changes to each peripheral model (A, B, and C in the figure) is performed independently. Under the assumption that no concurrent changes are allowed, after applying those changes propagation, new modifications can be performed.

As aforementioned, the approach we are presenting is able to deal also with *non bijective transformations*. In a non-bijective transformation, an element can be transformed into many different elements or vice versa. Thus, when a modification applies, the changes propagation algorithm may produce several alternative models. The user has to pick one of such alternatives to become the reference model. It is in fact infeasible to keep working with many alternative models, since each of those shall be successively propagated to any architectural model in **DUALLY** creating an unmanageable net of alternatives.

Summarizing, once a model is modified, the changes propagation process that we propose is as follows:

1. Propagate the changes to A_0;
2. Select one of the possible alternatives obtained in A_0;
3. For each ADL in the **DUALLY** network, propagate the changes of A_0 to all the other models (and again select one alternative in case of many results).

The number of alternatives that we may obtain depends on the number of non-bijective transformations, on the number of alternatives that each non-bijective transformation has, and on the number of model elements that match the transformation. In other words, the number of alternatives depends on the degree of *non-determinism* of the involved model transformations. More precisely, let T_{NB} be the set of non-bijective transformations. $|T_{NB}|$ is the number of non-bijective transformations. Let a_t be the number of alternatives that each $t \in T_{NB}$ has. Finally, let e_t be the number of model instance elements that match with t. The number of possible models that can be generated is

$$\prod_{t \in T_{NB}} a_t^{e_t}$$

It is important to note that the number of possible alternatives that can be generated is not a side effect of our approach, but instead depends on how transformations are defined.

When dealing with total and bijective model transformations the inverse of a transformation can be used to re-obtain a (unique) source model starting from the target model. The scenario changes when dealing, instead, with a transformation that is neither total nor injective (please consider this is the most common case). Since some elements of the source model do not have a correspondence in the target model and vice versa, alternative models are typically generated. Model transformations shall be defined avoiding as much as possible ambiguities and with constraints and guards that help their disambiguation. Often, model transformations have ambiguities only on one direction. When the round-trip starts from the side that has no ambiguities, the transformation can be instrumented with information useful to remember, for each involved element, how the transformation should be disambiguated when going back to the initial side; this mechanism is supported by the ASP-based transformation approach used in this paper. Conversely, when the round-trip starts from the side with ambiguities, a bidirectional model transformation should produce all the possible target models and allow the designer to choose the most adequate target model; this is exactly what the ASP-based transformation approach does.

Referring to the running example introduced in Sect. 2, the changes propagation process, as shown in Fig. 2, will be organized in five main steps. The preliminary steps consist in considering Save-CCM and to model a case study in this component model. The model we consider represents the architecture of an Adaptive Cruise Controller (ACC) that has been a recurring example throughout the development of Save-CCT (e.g. [24]). ACC is an extension of a standard Cruise Controller and its main functionalities will be described in

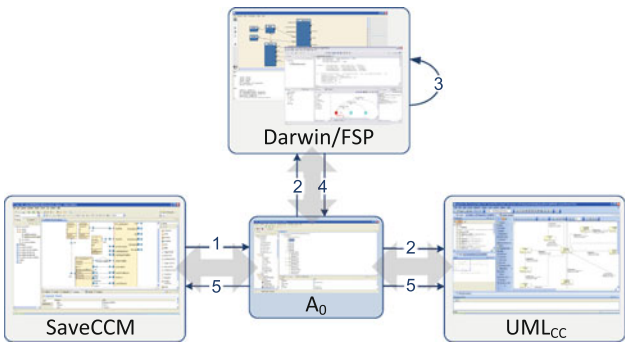


Fig. 2 Motivating example

Sect. 5. Once a satisfactory model has been designed, the whole process of the motivating example is organized into five main steps:

1. The first step consists in translating the initial model from SaveCCM to A_0.
2. The second step consists in translating the A_0 model to both Darwin/FSP and UML$_{cc}$. The designer will now focus on the Darwin/FSP model.
3. While the Darwin specification is obtained by transforming SaveCCM, FSP processes need to be specified by hand (since behavioural information is missing in SaveCCM). LTSA discovers an error and suitable modifications will be required.
4. A change propagation back to the A_0 model is required, so the changes made in Darwin/FSP are propagated to A_0. This may lead to the generation of a set of alternative A_0 models. The user selects the best model in A_0 among the generated alternatives. The model is valid also for A_0.
5. The selection made by the user is propagated to both SaveCCM and UML$_{cc}$. A report in MagicDraw can be generated to produce the required documentation of the system.

3.2 Bidirectional model transformation for change propagation

The proposed change propagation process allows us to keep synchronized models conforming to different ADLs by passing through A_0. In order to do this, bidirectional model transformations are specified using a relational and declarative approach based on the Answer Set Programming (ASP) language [3]. The approach exploits the benefits of logic programming that enables the specification of relations between source and target types by means of predicates and intrinsically supports bidirectionality in terms of unification-based matching, searching and backtracking facilities. A high-level overview of the ASP-based model transformation approach is provided in Fig. 3: starting from an encoding of involved metamodels and source model, the representation of the target one is generated. The ASP program performing the given model transformation consists of the following: (i) *Relations*, which describe correspondences among source and target element types; (ii) *Constraints*, which specify restrictions on the given relations that must be satisfied in order to execute the corresponding mappings; and (iii) *Transformation rules*, which are bidirectional rules implementing the specified relations as executable mappings.

This approach is capable to deal with *non bijective* [38] and *non total* transformations. Non bijectivity implies that each time a change occurs there is no guarantee to obtain a unique reverse result. In this case, the approach is capable to

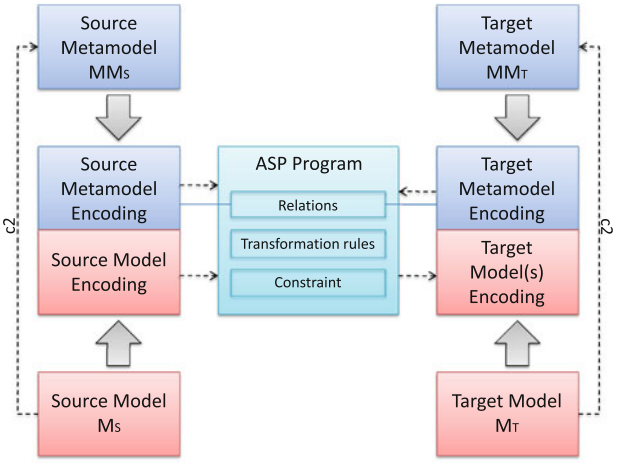


Fig. 3 ASP-based model transformation approach

generate a set of candidate models which are the closest to the ideal one from which to generate the previously modified model. So, the designer can choose the result of the reverse application of the given transformation among the approximated models. Non totality implies that the generated model has been manually modified and it is not any more reachable by any transformation. In the ASP-based approach, only the relevant concepts of the source models are mapped toward the corresponding target elements [18].

An overview of the integration of **DUALLY** features with a change propagation approach is depicted in Fig. 4. At each step, the approach involves a specific ADL_1 *metamodel* and the A_0 *metamodel*. For each ADL, a weaving model WM between the ADL and A_0 metamodels is defined. Such a weaving model is the mechanism to provide correspondences among the ADL and A_0 concepts at the metamodel level. In our approach, weaving models define the links between a notation and A_0 so as to later use them to generate

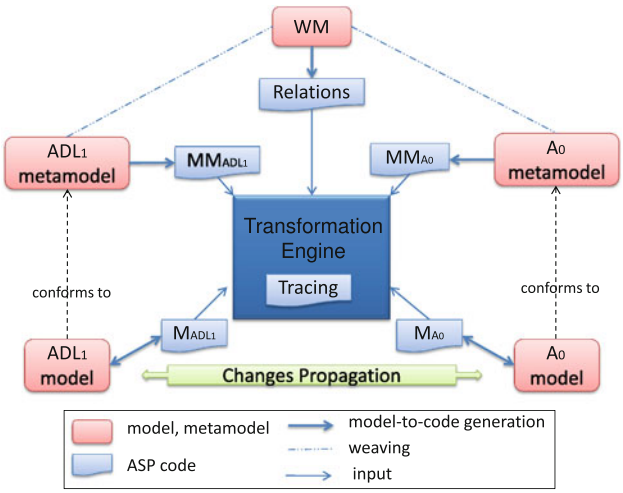


Fig. 4 Overview of the approach

specific *Relations* (encoded in ASP). Their role is in essence that of constituting the bridge between A_0 and the linked architectural notation. These links are defined once, during the process that integrates a certain ADL within **DUALLY**. Section 3.2.1 gives the details on how the *Relations* are managed within our approach.

Both metamodels (to be more precise, their ASP encoding MM_{ADL1} and MM_{A_0}) and the *Relations* defined among them are part of the input of the *Transformation Engine*. Starting from the ASP encoding of the source model (M_{ADL1} or M_{A_0}), the *Transformation Engine* automatically interprets and transforms it into the ASP encoding of the target model (M_{A_0} or M_{ADL1}, respectively). It relies on an ASP solver, which is a system for computing the solutions of an ASP program. This work makes use of the *Smodels* solver [40]. Furthermore, we developed a specific bridge between each model and its corresponding ASP encoding that guarantees no loss of information while passing from a model to its ASP encoding and vice versa (such a bridge is presented in Sect. 4.2).

In order to propagate changes among models, the *Transformation Engine* needs both the source and the target models as input. In this way, the transformation is able to generate a set of candidate models which are the closest to the ideal one from which to generate the previously modified model. In our approach, the designer's task is just to select the model that best fits his needs among the candidate models produced by the *Transformation Engine*.

The *Transformation Engine* is generic and independent from any specific metamodel. Indeed, it is composed of generic bidirectional rules interpreting the involved metamodels and the relations so that such relations hold on models conforming to the source and target metamodels. Due to the different expressiveness of the involved metamodels, bidirectionality is attained by means of tracing information, which are automatically generated by the engine and given as input of the transformation (*Tracing*).

As in standard model transformations engines, the ADL_1 and A_0 metamodels (and the *Relations* that link them) are defined once, whereas models to be transformed can change.

A more detailed description of the *Transformation Engine* and how it works for transforming and propagating changes is provided in Sect. 3.2.2.

3.2.1 Correspondences specification

The weaving model is a core element in both the transformation and the changes propagation features. A weaving model (WM in Fig. 4) defines a set of links between an ADL metamodel and the A_0 metamodel. A number of methods to specify and construct weaving models is currently being explored and exploited, but conceptually, weaving models conform to a given weaving metamodel, and can be defined either manually or by utilizing ad-hoc scripting languages. Our approach reuses the **DUALLY** weaving metamodel and therefore enables the (graphical) definition of **DUALLY** weaving models (see Fig. 11). The weaving metamodel provided by **DUALLY** defines the types of link that can be established in a weaving model. Its main elements are directed and bidirectional correspondences to relate two or more concepts, feature equivalences to relate attributes or references, bindings to user-defined constants, and some other auxiliary construct. The definition of the weaving model constitutes the main step to be carried out in order to integrate a certain architectural language into the notations topology. Weaving models (and so their corresponding ASP relations) contain the rationale that enables the transformation engine to synchronize two architectural models.

As can be noticed in Fig. 4, a weaving model WM is automatically transformed into a set of *Relations* interpreted by the transformation engine. This is performed by means of a model-to-code transformation that handles each element of the weaving metamodel and generates its corresponding ASP code. More precisely, a relation in *Relations* is a declaration of an assertion used by ASP to produce the set of answers (i.e., the set of alternative solutions to the changes propagation). Thus, the quality of the solution set is strongly affected by the way the weaving model is defined. Each relation between corresponding elements should be defined as precisely as possible. Implementation details on how weaving models are managed by our approach will bee provided in Sect. 4.3.

The presence of both weaving models and of the bridge between models and their ASP encoding represents a strong benefit for our approach. They constitute a layer that makes the whole ASP engine totally transparent for the designer. Designers just have to create a weaving model to define the correspondences between an ADL metamodel and A_0; the weaving model will be automatically transformed into the specific ASP program implementing the transformation. Furthermore, designers can create models using usual MDE techniques; it is up to the model-to-code bridge provided by our approach to automatically translate the models into their ASP encoding and vice versa.

3.2.2 Changes propagation engine

As anticipated in Sect. 3.2 and shown in Fig. 4, the *Transformation Engine* requires several inputs: the relations (*Relations*), the encoding in ASP of the metamodels (MM_{ADL1} and MM_{A0}) and the encoding in ASP of the models (m_{ADL1} and m_{A0}). As described in the previous section, *Relations* are generated from the correspondences expressed in the weaving model.

Both the encoding of metamodels and models are obtained by means of model-to-code transformations that generate,

for each element of the input model, its corresponding ASP code. It is important to remark that models encoding in ASP is defined without loss of information, thus allowing bidirectional transformations: model-to-ASP and ASP-to-model. This is particularly important during synchronization when a change propagated to an A_0 model has to be successively propagated to models conforming to other ADLs.

The overall output of the approach is the synchronization of the model instances by means of change propagation. As described, the proposed approach is able to back propagate from the modified targets a set of models that approximate the exact source. The possible solutions are proposed to the software architect that has to select what he/she retains the best one. The mechanism is based on mapping traces to automatically compute the reverse transformation. In particular, both bidirectionality and change propagation are strictly related to traceability information which stores the links between the elements of the target which have been created from a certain source element. This information is critical, especially if the back propagation is supported, because of the non-bijectivity of the relations between source and target. Since in general transformations are non-total, bidirectionality is provided even when the modifications performed by hand on the model makes it unreachable by any transformation. Furthermore, the selection made by the architect is stored in a tracing set in order to automatically re-propose the choice in subsequent round-trip journeys.

4 Realizing the approach

This section provides technical details of our change propagation approach. Section 4.1 illustrates the technologies we use and Sect. 4.2 presents the *model-to-code bridge* between models (along with metamodels) and their *ASP encoding*. Further on, Sect. 4.3 describes the generation of *ASP relations* starting from weaving models. Section 4.4 provides the details of the *ASP model transformation engine*, focussing also on *tracing* mechanisms and *change propagation*.

4.1 Technologies overview

The proposed approach is implemented as a plug-in of the Eclipse[5] framework. Figure 5 presents an overview of the technologies we use in this work.

Our approach uses two components of the **DUALLY** framework: (i) the weaving metamodel (see Sect. 4.3 for the details on how it relates to the ASP transformation engine) and (ii) the UML importing mechanism, so that also UML models may be used in our approach; please refer to [30] for more details on this specific feature of **DUALLY**.

[5] Eclipse project Web site: http://www.eclipse.org.

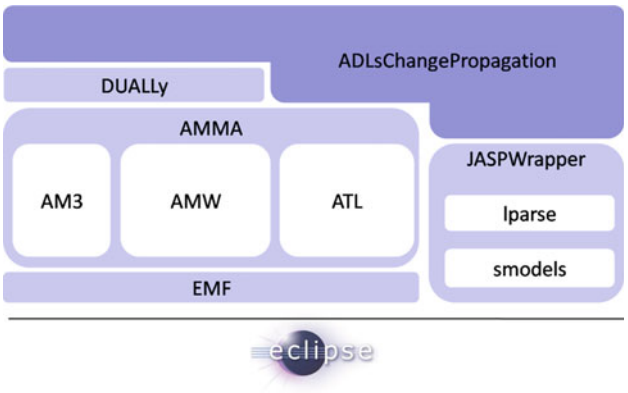


Fig. 5 Technologies of the approach

The approach of this paper is implemented in the context of the ATLAS Model Management Architecture (AMMA) [2]. We make use of the following AMMA components:

- Atlas Transformation Language (ATL) [23] is a model transformation language, with its own abstract syntax and environment. ted editor, and so on. In the context of this work, we use ATL as part of the technological bridge between models and ASP code (see Sect. 4.2).
- Atlas Model Weaver (AMW) [10] is a platform for managing weaving models. Typed links among models (and metamodels) are saved in a weaving model, which conforms to an extensible weaving metamodel. The AMW component is extensible; this enabled us to extend it with the model-to-code facility producing ASP relations from weaving models.
- TCS (Textual Concrete Syntax) [22] is a DSL for the specification of textual concrete syntaxes. TCS automatically generates tools for model-to-text and text-to-model transformations. In our case indeed, we use it to bridge models with their ASP encoding.

AMMA features also other additional components (like KM3 [21]) and ATP technical projectors [2] that are out of the scope of this work.

In the AMMA platform models and metamodels are based on the Eclipse Modelling Framework (EMF[6]). EMF provides runtime model support, a persistence layer based on XMI and a generic user interface for viewing and editing models.

The core of the point-to-point change propagation mechanism is the ASP solver. In this work we use the Lparse and Smodels[7] as front-end and ASP solver, respectively. Since both Lparse and Smodels are standalone C applications, we use the JASPWrapper[8] component to integrate them into

[6] EMF Web site: http://www.eclipse.org/modeling/emf.

[7] Smodels site: http://www.tcs.hut.fi/Software/smodels.

[8] http://www.pirrotta.it/giovanni/jaspwrapper/index.php.

our Eclipse Environment. Our model transformation engine interacts exclusively with the JASPwrapper interface.

4.2 Technological bridge between EMF and ASP

If on the one hand our work is built on a model transformation approach based on ASP, on the other hand we do not want designers to manually create ASP encodings of their models. For this reason we developed a technological bridge between EMF entities (like models and metamodels within the Eclipse framework) and their ASP encoding. We succeeded in automating such a bridge, so designers just model using EMF and the corresponding ASP encodings are *automatically* produced by our technological bridge. Figure 6 presents its main elements.

For the sake of clarity, this section will provide small examples taken from the motivating example introduced in Sect. 2.

Referring to the upper part of the figure, at the metamodelling level the technological bridge is implemented as a two-step process: in the first step, the metamodel is translated into an intermediate metamodel (called *ASPmm*) via a model-to-model transformation. The second step effectively produces ASP code by using a model-to-text solution.

ASPmm represents an abstract representation of the ASP encoding for metamodels. It will be described later in this section. The model-to-text transformation is realized through TCS. In order to do this, TCS requires that the involved metamodel is similar in structure to the code being generated; this makes unavoidable the intermediate step of passing through *ASPmm*.

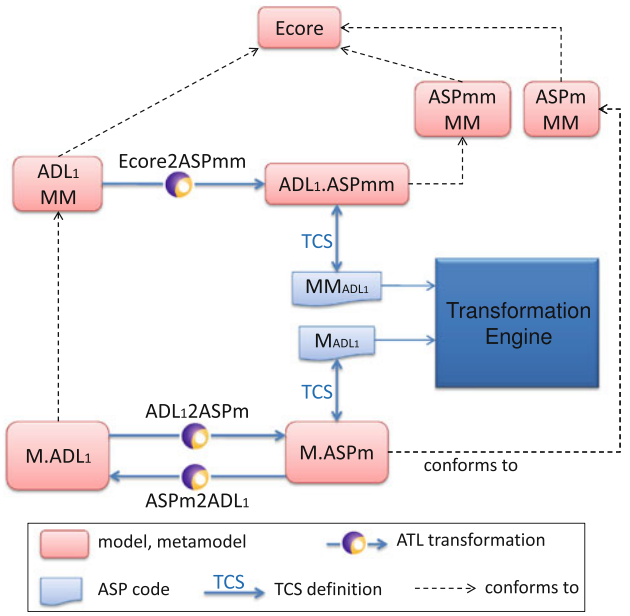


Fig. 6 Technological bridge between EMF and ASP

The same two-steps bridging mechanism and rationale hold at the modelling level, lower part of Fig. 6.

Figure 7a, b represent the *ASPmm* and the *ASPm* metamodels, respectively, and contain only the necessary elements for creating the needed ASP encodings. The *ASPmm* metamodel is composed of the terms: `metanode`, `metaprop`, and `metaedge`, while the *ASPm* metamodel is composed of the terms: `node`, `edge` and `prop`. The detailed description of the encoding is available in Appendix A.

The *Ecore2ASPmm* ATL transformation is in charge of producing an *ASPmm* model starting from an EMF metamodel. The domain of that transformation is Ecore and its codomain is the *ASPmm* metamodel itself; this implies that *Ecore2ASPmm* is generic and does not depend on the metamodel being transformed. In the following we give an idea on how the ASP encoding relates to its initial EMF metamodel: Fig. 8 shows the Component metaclass of the SaveCCM

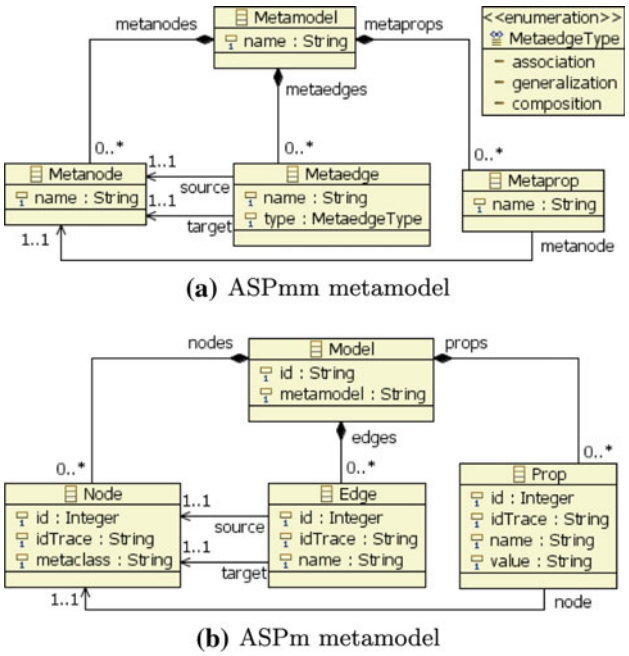


Fig. 7 Intermediate metamodels of the technological bridge

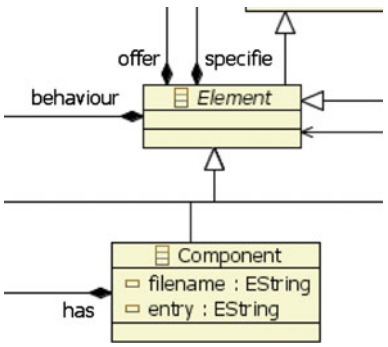


Fig. 8 Part of the SaveCCM metamodel in EMF

```
1 metamodel(saveccm).
2 ...
3 metanode(saveccm, Component).
4 ...
5 metaprop(saveccm, name, Component).
6 metaprop(saveccm, filename, Component).
7 metaprop(saveccm, entry, Component).
8 ...
9 metaedge(saveccm, composition, specifie, Component,
      Attribute).
10 metaedge(saveccm, composition, has, Component,
      BindPort).
11 ...
```

Listing 1 Part of the SaveCCM metamodel encoded in ASP

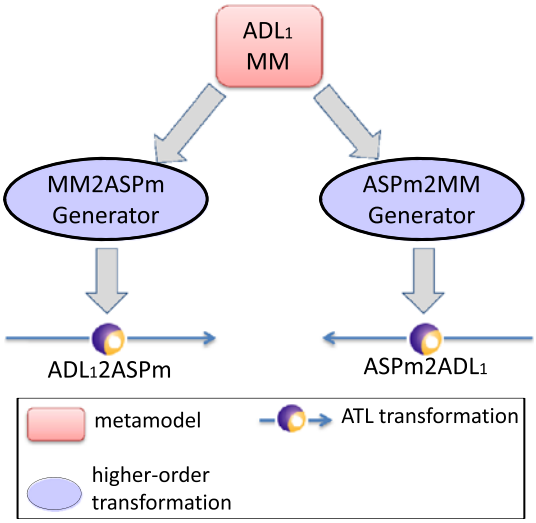


Fig. 9 Bridge generation at the modeling level

metamodel and Listing 1 shows its related ASP encoding. The figure and listing below are self-explaining; it should be noted how the concepts inherited by Component (like *+++specifie*) are part of Component itself in ASP. This ensures that the definition of each element in ASP is self-contained, simplifying our internal model transformation engine.

Since the our TCS specification contains only one-to-one mappings and its realization can be considered just a technical step, we will not further detail this aspect.

In order to be considered fully automatic, our technological bridge must work also at the modelling level. Moreover, it must work in two directions: from EMF to *ASPm* models and vice versa. Let us suppose that we need to transform an ADL_1 model into an A_0 model; on the one hand we must pass from the source EMF model to its ASP encoding, while on the other hand we must be able to obtain the target EMF model from its ASP encoding produced by the ASP transformation engine. As shown in Fig. 6 the model-level bridge is similar to that at the metamodelling level. As previously stated, the main feature of this technological bridge must be its generality, and then the ATL transformations must work for every EMF metamodel. It implies the use of higher-order transformations (HOTs), see Fig. 9.

In this specific case, we need two higher-order transformations defined once forever (also defined in ATL):

1. *MM2ASPmGenerator* takes as input an EMF metamodel `MM` and produces the *MM2ASPm* transformation. This transformation generates a model conforming to *ASPm* from a model conforming to `MM`;
2. *ASPm2MMGenerator* starting from an EMF metamodel `MM` generates the *ASPm2MM* transformation; it performs the inverse task of *MM2ASPm*.

After the generation phase, depending on the direction in which the designer is transforming, the next step is to execute the just created model transformations in order to obtain an *ASPm* (or EMF) model.

The final step of the bridge is implemented in TCS. The implementation is analogous to the one described for metamodels, so we will not describe it in this paper.

Figure 10 and Listing 2 give a general idea on how the bridge works when considering a SaveCCM component and producing its corresponding ASP encoding. There is a component (line 3) called *Sensors* that offers 5 ports (lines 4–9); the names of both the component and ports are shown in lines

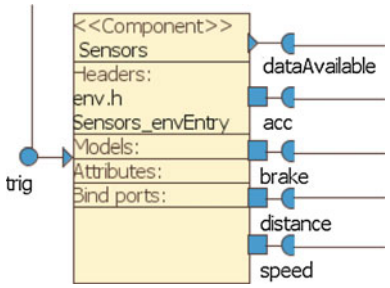


Fig. 10 Part of a SaveCCM model

```
1 model(ms1, saveccm).
2 ...
3 node(saveccm, 91, cc3, Component).
4 node(saveccm, 187, p3, DataOut).
5 node(saveccm, 191, p4, DataOut).
6 node(saveccm, 195, p5, DataOut).
7 node(saveccm, 199, p6, DataOut).
8 node(saveccm, 136, p7, TriggerIn).
9 node(saveccm, 216, p8, TriggerOut).
10 ...
11 prop(saveccm, 1071, cc31, 91, cc3, name, Sensors).
12 ...
13 prop(saveccm, 1881, p31, 187, p3, name, speed).
14 prop(saveccm, 1921, p41, 191, p4, name, distance).
15 prop(saveccm, 1961, p51, 195, p5, name, brake).
16 prop(saveccm, 2001, p61, 199, p6, name, acc).
17 prop(saveccm, 1371, p71, 136, p7, name, trig).
18 prop(saveccm, 968, p81, 216, p8, name, dataAvailable).
19 ...
20 edge(saveccm, 9394, cc392, offer, 91, cc3, 187, p3).
21 edge(saveccm, 9697, cc395, offer, 91, cc3, 191, p4).
22 edge(saveccm, 99100, cc398, offer, 91, cc3, 195, p5).
23 edge(saveccm, 102103, cc3101, offer, 91, cc3, 199,p6).
24 edge(saveccm, 105106, cc3104, offer, 91, cc3, 136,p7).
25 edge(saveccm, 107932, cc3106, offer, 91, cc3, 216,p8).
26 ...
```

Listing 2 Part of a SaveCCM model encoded in ASP

13–18. The reference linking the component to its offered ports is called *offer* and is represented in lines 20–25.

Summarizing, by providing this technological bridge, we can use the ASP-based change propagation approach in a fully automatic way. Furthermore, it is independent from the underlying ASP-specific mechanism. This allows designers to concentrate only on modelling using EMF and transparently apply the change propagation approach. Moreover, further developments of the change propagation mechanism may be carried out without modifying the modelling platform. In that case the technological bridge is the only component that must be upgraded.

4.3 Generation of ASP relations from weaving models

As previously stated, the creation of a weaving model is the main step to be carried out in order to link an architectural language with A_0, and thus with all the other notations in the topology. An interesting feature of our approach is that software designers do not need to manually encode models in ASP at any level (i.e. both modelling and metamodelling). The automatic generation of ASP relations (*Relations*) from weaving models completes the hiding of the ASP engine.

In this section we present how our framework generates ASP relations (*Relations*) starting from AMW weaving models.

Designers create weaving models by using the AMW graphical editor extended by **DUALLY**. Figure 11 provides a screenshot of the extended weaving model editor: both the considered ADL and the A_0 metamodels are rendered using a tree-based editor, and the weaving links are graphically created via a central panel providing the set of correspondences defined in the weaving metamodel.

Figure 12 represents how we realized the automatic generation of ASP relations. All metamodels conform to Ecore, an implementation of meta-metamodel in Eclipse. The **DUALLY** weaving metamodel (*DUALLyWMM* in the figure) specifies the types of correspondences between a metamodel and A_0; the main elements of the **DUALLY** weaving metamodel are:

- *Correspondence:* represents a generic mapping between elements of the woven metamodels. There are three types of *Correspondence*: *Left2Right*, *Right2Left* and *Equivalence*. This adds navigability semantics to weaving links. Indeed, the first two types represent unidirectional correspondences, while the latter one represents bidirectional correspondences.
- *FeatureEquivalence:* defines a mapping between two structural features (i.e. attributes or references).
- *WovenElementLink:* specifies a correspondence between a woven element and a feature of another woven element.
- *ConstantBinding:* specifies a correspondence between a structural feature and a constant value; so, every instance of the target structural feature is initialized with the specified constant value.
- *EnumerationMapping:* specifies a mapping between two enumerations (and between their literals as well); its

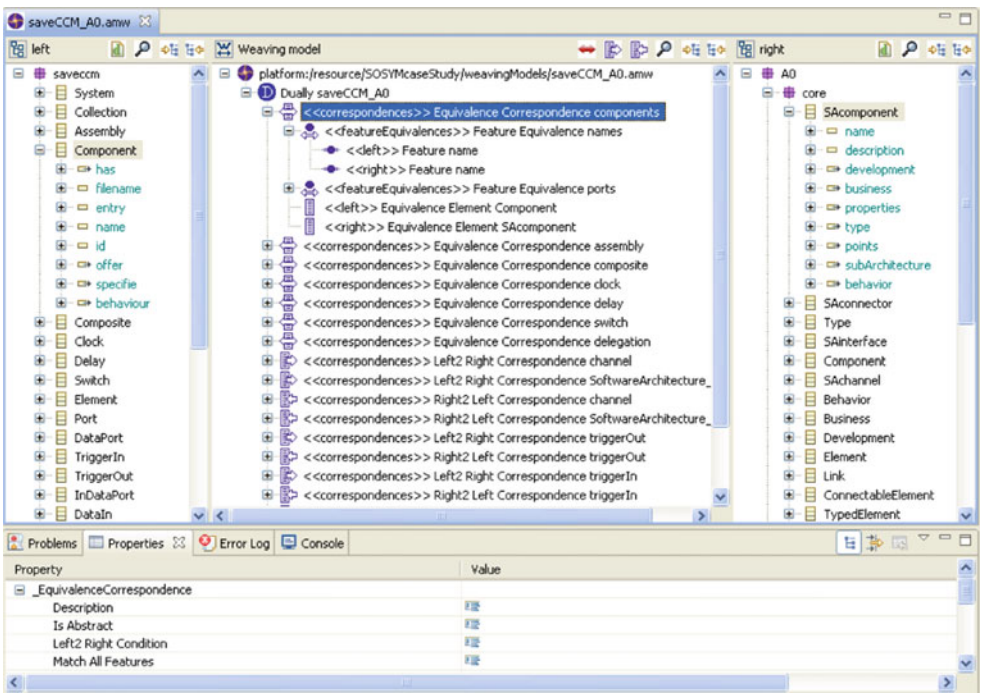


Fig. 11 Screenshot of the weaving models extended editor

semantics is that each occurrence of a specific literal will be transformed into an instance of the mapped literal.

Summarizing, from a high-level point of view, a **DUALLY** weaving model defines a set of correspondences between metaclasses, each of which may contain feature equivalences and some auxiliary elements; optionally, it may also contain enumeration mappings and constant bindings.

Once a weaving model between the ADL metamodel and A_0 has been developed, it is transformed into a set of ASP relations by executing a generic ATL transformation (*ASPrelationsGenerator* in Fig. 12). Appendix B provides additional details on this transformation. This transformation takes as input a weaving model and produces a fragment of the ASP code for each element of the weaving model. It is important to remark that, since the weaving model operates at the metamodelling level, *ASPrelationsGenerator* is executed only once for each ADL to be integrated into the languages topology; the generated relations are valid for each model conforming to the woven ADL metamodel. Further on, prior to generate the ASP relations, *ASPrelationsGenerator* performs additional semantic checks which help in generating ASP code that reflects the logic of the weaving model.

Listing 3 shows a fragment of ASP code produced by *ASPrelationsGenerator*. In this case the input of such a transformation is an *EquivalenceCorrespondence* defined between a SaveCCM Component and an A_0 SAcomponent. The generated ASP code is composed of the *Component_SA-Component* relation and the set of constraints complementing it. The details of the encoding are available in Appendix A.

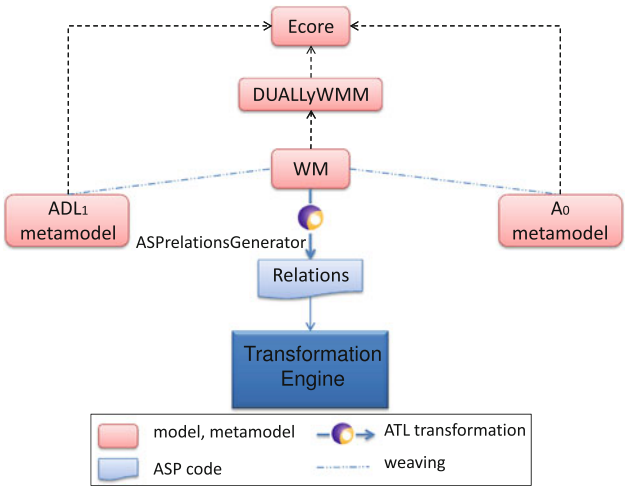


Fig. 12 Generation of ASP relations from weaving models

```
relation(equivCorresp1,bx,bx).
relation(equivCorresp1, a0, sAcomponent, 1).
relation(equivCorresp1, saveccm, component, -1).
:- node(saveccm, IDx, ID, component), not node'(a0,
    IDx + 1, ID, sAcomponent), mmt==a0.
:- node(a0, IDx, ID, sAcomponent), not node'(saveccm,
    IDx - 1, ID, component), mmt==saveccm.
```

Listing 3 Fragment of *Component_SAComponent* ASP relation

4.4 The ASP model transformation engine

As described so far, this approach takes as input a source model and the involved metamodels (see Fig. 4). After the encoding phase, the deduction of the facts representing the target model is performed according to the set of relations between the elements of the source and target metamodels, constraints which specify restrictions on the relations (both derived from the weaving model), and the bidirectional rules defined in the ASP transformation engine. The transformation process can be logically divided into two steps: (i) given the input (meta) models, the transformation rules induce all the possible solution candidates according to the specified relations; and (ii) the set of candidates is refined by means of constraints.

In this section we explain how the ASP engine is able to realize bidirectional transformation between source and target models supporting propagation of changes.

4.4.1 Transformation rules

Transformation rules produce target elements according to the given relations. In particular, target elements *node*, *edge* and *prop* are created if the following conditions are satisfied:

- at least a relation exists between a source element and the candidate target element;
- the source element involved in the relation is declared in the input source model;
- the candidate target element conforms to the target metamodel;
- finally, any constraint defined on the relations is violated.

For instance, Listing 4 shows an excerpt of the code implementing the transformation rules able to create *node′* elements according to the *Component_SAComponent* relation

```
mapping_node(MM,ID,MC) :- relation(R,MM,MC), relation(
    R,MM2,MC2), node(MM2,ID,MC2), MM!=MM2.

is_related(MM,Node) :- relation(R,MM,Node), relation(R
    ,MM2,Node2).

is_target_metamodel_conform(MM,Node) :- metanode(MM,
    Node).

{is_generable(MM,ID,Node)} :- mapping_node(MM,ID,Node)
    , is_related(MM,Node), is_target_metamodel_conform
    (MM,Node), MM=mmt.

node'(MM,ID,Node) :- is_generable(MM,ID,Node),
    mapping_node(MM,ID,Node), MM=mmt.
```

Listing 4 Fragment of the ASP *SaveCCM_2_A0* transformation

of Listing 3. The rules in line 1 check if the mapping can be executed. The rules in line 2 verify if a given metaclass is involved in a relation. The rules in line 3 check if a given metaclass conforms to the target metamodel. The rules in line 4 compute all the elements that may be generated. Finally, the rule in line 5 creates target elements.

As already mentioned, transformation rules produce several target models (answer sets), which are all the possible combinations of elements that the program is able to create. Constraints have the role to refine such set of solutions by removing all answer sets that satisfy their conditions and can be based both on transformation specifications and on target metamodels. It is worth noting that the refinement process can build more than one target model conforming to the target metamodel. Usually, it is desirable to have a univocal correspondence between source and target elements, i.e. a bijective relation between them. However, multiple valid solutions can be obtained for the same problem because of the different expressive power of the involved metamodels, this aspect will be discussed later in this section.

4.4.2 Tracing

The invertibility of transformations may depend on a number of factors, such as the different expressive power of the involved metamodels (as discussed in Sect. 3). For example, this issue can affect approaches for performing model transformation-based reverse and round-trip engineering. A possible solution can rely on trace information that connects source and target elements; in this way, once a model transformation has been performed, all the target elements are linked to the corresponding source ones.

During the transformation process, the relationships between models that are created by the transformation executions can be stored to preserve mapping information in a permanent way.

Elements managed by the ASP transformations do not need to store further information since the transformation already contains all the needed information. The mechanism makes use of identifiers that bind elements of source models to elements belonging to the set of tracing information. Nevertheless, this information is critical especially if the back propagation is supported, because of the non bijectivity of the relations between source and target. Furthermore, transformations are not total in general and consequently some source elements might not be mapped to the target model. In these cases, tracing information needs to be stored in order to avoid loss of information when the transformation is applied again.

Thereby, trace information is used to provide bidirectionality even in the case in which the generated model has been manually modified in such a way it is not reachable any more by any transformation. Without loss of information, the inverse transformation is able to create critical elements from the trace information. Furthermore, it is useful to remark that whenever traces are required, they are also created in backward transformations due to the bidirectionality of the mappings.

Bidirectional transformations and trace links play a key role in supporting change propagation. The next section describes how ASP can be used for supporting change propagation and to approximate source specifications when manual changes occur on target models.

4.4.3 Change propagation

Change propagation management aims at reflecting modifications done in a target model to the corresponding source one. The availability of this feature is not obvious because of the partial and non-injective nature of model transformations (as illustrated in Sect. 3).

Let us consider the following scenario. Once a target model *TM* is automatically obtained from a source model *SM* by means of a transformation *T*, the designer may need to manually modify the generated model to accommodate unforeseen requirements or limited expressiveness of the source metamodel. At this point, being that *T* is a bidirectional transformation, it can be applied to *TM* and one or more models can be obtained.

Due to the different expressive power of the involved metamodels, changes may be propagated in a number of different ways and the application of the reverse transformation could approximate more solutions. ASP is able to select suitable solutions on the basis of conformance or structural issues, leaving to the designers the evaluation of semantics aspects.

Furthermore, considering the transformation as a function, after manual changes the obtained model can be in the image of *T* or outside. If we consider the reverse application of *T*, the models that can be generated by means of it are in the domain of *T*. However, the proposed approach is able to guarantee that, when brought forward again, all sources result in the changed target model. This is supported by the use of trace information explained in Sect. 4.4.2.

In the next section will present how the proposed approach is used in practice and how it is able to manage the propagation of changes among three distinct ADLs.

5 The adaptive cruise controller case study

In this section we show the application of the proposed approach to a real case study[9]. The objective of this case study

[9] The source code of the approach as well as the models that compose the case study can be found on http://dually.di.univaq.it/changePropagation.

is to present how the change propagation mechanism is able to deal with interoperating architecture models conforming to three different architectural notations. A high-level description of the case study has been presented in Sect. 2. The following preliminary section presents the involved architectural notations and the weaving model between each of them and A_0; the other sections apply the approach by following the steps of the discussed motivating example scenarios.

5.1 The involved ADLs and their weaving models

The architecture description languages involved in this case study are SaveCCM, Darwin/FSP and the UML$_{cc}$ profile. Consequently, three are the weaving models in charge of establishing the correspondences between their metamodels: (i) *SaveCCM_*A_0 contains the correspondences between the *SaveCCM* and A_0 metamodels, (ii) *DarwinFSP_*A_0 links Darwin/FSP and the A_0 metamodels and (iii) *UML$_{cc}$_*A_0 captures the correspondences between the UML$_{cc}$ profile and the A_0 metamodel.

In order to leave the discussion as readable as possible, the weaving models presented in this section contain only the minimum number of correspondences to generate the needed model-to-model transformations; auxiliary weaving elements have been abstracted out.

5.1.1 SaveCCM

The SaveComp Component Model (SaveCCM) [24] defines a graphical syntax and a runtime framework to design software embedded systems with a focus on system resources and timing requirements. The SaveCCM execution semantics is based on the control flow (pipes-and-filters) paradigm, and on the distinction between data transfer and control flow. Modelling explicitly the control flow makes the design analysable with respect to temporal behaviour. The SaveCCM metamodel defines three main architectural elements: components, switches and assemblies. Components are the main architectural element in SaveCCM; they may have input/output ports and quality attributes. The switch construct provides the means to change the components interconnection structure, either statically or dynamically. An assembly can be considered as a means for naming a collection of components and hiding its internal structure.

The *SaveCCM_*A_0 weaving model we use in this work has been developed by adapting the *SaveCCM_*A_0 weaving model in [6]. A SaveCCM `System` is mapped into an A_0 `SoftwareArchitecture` containing all the other A_0 elements. SaveCCM `Component` and `Connection` are mapped to `SAcomponent` and `SAchannel`, respectively. `Assembly` and `Composite` components are both mapped to `SAcomponent`. SaveCCM `Clock` and `Delay` elements correspond to generic `SAcomponents`. `Switch` is mapped into an `SAConnector` because of their common coordination and interaction semantics. `DataPort`, `TriggerPort`, and `TriggerData-Port` correspond to `SA- interface`; 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. SaveCCM attributes correspond to A_0 properties.

5.1.2 Darwin and FSP

Darwin [27] and FSP [26] are two complementary efforts for describing the structure (Darwin) and behaviour (FSP) of a software architecture. Darwin is a structural ADL for describing a system in terms of hierarchically structured components, interfaces (which can be required or provided) and interconnections among components. FSP models behavioural aspects of a software system in terms of concurrent processes; the LTSA tool in turn generates a labelled transition system from the FSP description. An FSP specification is attached to a Darwin one by specifying the behaviour of Darwin components via FSP processes. The Darwin/FSP metamodel and the *DarwinFSP_*A_0 weaving model we use in this case study are taken from [30]. The Darwin/FSP metamodel contains the union of all the constructs defined in the specification of both the Darwin and FSP languages.

In the *DarwinFSP_*A_0 weaving model, a Darwin `ComponentInstance` corresponds to an A_0 `SAcomponent` and vice versa. Also `SAconnector` in A_0 is mapped into a Darwin `ComponentInstance`. This is due to the fact that Darwin does not have the concept of software connector and the semantically closest concept to it in Darwin is `ComponentInstance`. A `Portal` corresponds to an A_0 `SAinterface` and vice versa. The kind of `SAinterface` is set according to the type of the *Darwin/FSP* `Portal`. `ComponentDeclaration` and `Binding` in *Darwin/FSP* correspond to `SAchannel` and `SAStructuredtype` in A_0, respectively. Each *Darwin/FSP* `ProcessDeclaration` is mapped to a `StateDiagram` and vice versa. `Action-Prefix` and `Choice` are both mapped to an A_0 `State` and vice versa. *Darwin/FSP* `Action` is mapped to `Transition` and vice versa. `ERROR`, `END` and `STOP` can be considered as special kinds of final states, so we mapped them to A_0 `FinalState`.

5.1.3 UML$_{cc}$

UML$_{cc}$ is a UML profile for specifying component-based software architectures. The profile has been introduced in [30], and it is designed having in mind the guidelines provided in [20]. According to the UML$_{cc}$ profile, a software architecture is described by a component diagram containing the main components (represented by the `CcComponent Instance` stereotype), connectors (`CcConnector`

Instance) and their configuration. CcComponent Instances interact with their external environment through a set of CcPorts, while CcConnectorInstances communicate via CcRoles; they both extend the UML port metaclass. Both Components and connectors communicate through CcAttachments (i.e. stereotyped UML Dependencies). CcDelegations are stereotyped UML delegation connectors that link hierarchically structured components. Behaviour is expressed in terms of state machines (CcBehavior).

The *UML$_{cc}$_A_0* weaving model contains the correspondences between the UML$_{cc}$ profile and the A_0 metamodel. It defines an equivalence correspondence between CcComponentInstance and SAcomponent, whereas CcConnectorInstance corresponds to SAconnector. Both CcPort and CcRole correspond to SAinterface in A_0. CcAttachment in UML$_{cc}$ corresponds to SAchannel in A_0 and vice versa, while CcDelegation corresponds to SAbinding in A_0 and vice versa. Both CcComponentType and CcConnectorType correspond to SAstructuredType in A_0. CcBehavior is mapped into StateDiagram in A_0 and vice versa. State from UML corresponds to State in A_0 and Transition from UML is mapped into Transition in A_0.

5.2 The ACC model: from SaveCCM to A_0

In this case study we model the software architecture of an adaptive cruise controller (ACC) system. It is a standard Cruise Controller with extra functionalities allowing it to autonomously adapting to the changing environment. Specifically, two of them are the main extra functionalities: (i) keeping a safe distance towards a preceding vehicle; (ii) adjusting the maximum permissible speed in function of the speed-limit regulations. In our case study all relevant information to perform the adaptation is received by means of suitable sensors.

Figure 13a shows the software architecture of ACC modelled in SaveCMM by using the Save-IDE [36] modelling environment. The ACC system is composed of two main components: *Sensors* and *ACC_Application*. The former periodically provides data collected from the various sensors, whereas the latter is the core of the system. It accomplishes three main tasks: (i) analyse information provided by sensors and plan actions to be performed by the actuators; (ii) log the status of the system; and (iii) provide data to the driver display (called HMI). For the sake of simplicity, we do not show the internal structure of *ACC_Application* and the external context of the whole system. Three clocks regulate the timing of the ACC system. More specifically, *clk50hzA* triggers *Sensors* to sense the available current data, *clk50hzB* triggers *ACC_Application* that becomes active and ready to receive data. The availability of new data is notified by the *Sensors* component through a *dataAvailable* trigger. Finally, *clk10hz* operates at a lower rate; it triggers *ACC_Application* to log the status of the system and to provide data to the driver display.

Once the ACC system is modelled in SaveCCM, our approach automatically transforms such a model into an A_0 model. Such a transformation is driven by the weaving model described in Sect. 5.1.1. The transformation produces only one A_0 model since in the defined *SaveCCM_A_0* weaving model each element of SaveCCM has only one corresponding element in A_0. Specifically, the clocks, the *Sensor* component and the *ACC_Applica*-tion assembly become A_0 SAcomponents. Their ports become SAinterfaces independently from their nature (i.e. trigger or data). Finally, SaveCCM connections are transformed into SAchannels.

5.3 Transforming the A_0 model into Darwin/FSP and UML$_{cc}$

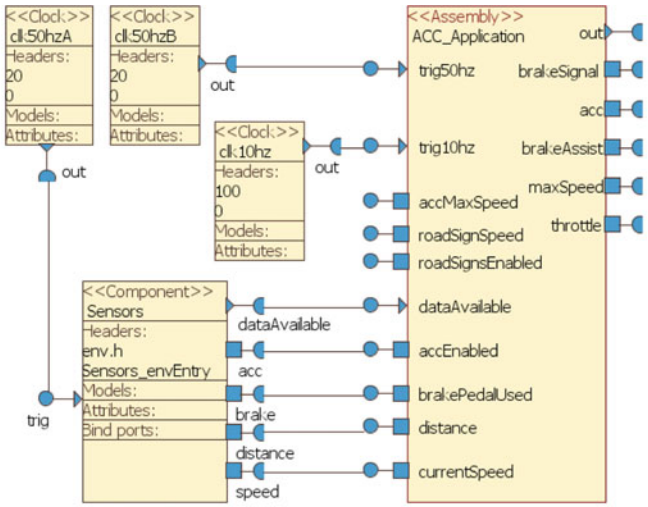
The ACC model in A_0 is then translated to both Darwin/FSP and the UML$_{cc}$ profile. Figure 13b, c shows the generated model in Darwin/FSP and UML$_{cc}$, respectively. In these transformations only one alternative is generated. In order to understand how these models are obtained, the interested reader can refer to the weaving models described in Sects. 5.1.2 and 5.1.3. As the models are pretty straightforward, we do not further discuss them in this paper.

It is important to remark how our approach is integrated with the tools supporting the involved architectural languages. SaveCCM and several UML tools are based on EMF and so they are already integrated into our Eclipse platform. A different case is that of Darwin/FSP: Software Architect's Assistant and LTSA (Darwin and FSP tools, respectively) are based on two proprietary textual formats. In this case study we use a previously developed facility [30] to transform those textual specifications into a model conforming to the Darwin/FSP metamodel and vice versa.

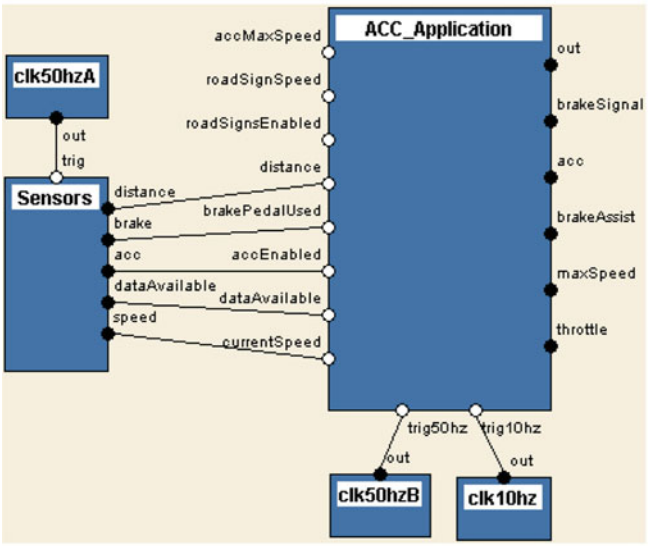
5.4 Modelling the behavior in LTSA

According to the scenario described in Sect. 2, we assume that now the designer is interested to define the behavioural description of ACC by means of an FSP specification.

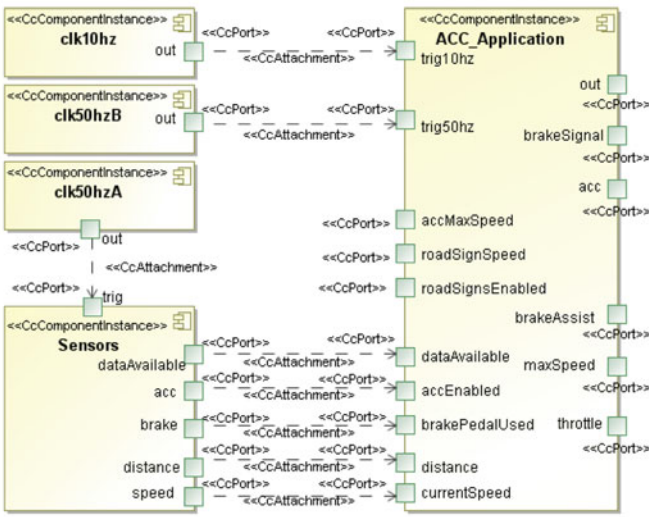
Listing 5 shows the behaviour of ACC_Application; it is organized into two different parallel processes: COMPUTE_ STATUS and LOG_HMI. The COMPUTE_STATUS process encodes the retrieval of the data information once a clock is received (tick3) and the data are made available by the *Sensors* component (dataAvailable). The LOG_HMI process is activated by *clk10hz* (tick1), provides the data to the HMI and logs the status of the system. For the sake of simplicity, the behaviour of the other



(a) The ACC system in SaveCCM



(b) The ACC system in Darwin



(c) The ACC system in UML$_{cc}$

Fig. 13 The ACC system modelled in SaveCCM and the generated models in Darwin and UML$_{cc}$

```
1 ||ACC_Application = (COMPUTE_STATUS || LOG_HMI).
2 COMPUTE_STATUS = (rule -> ACTIVE),
3 ACTIVE = (tick3 -> checkStatus -> dataAvailable ->
4       readDistanceData -> readSpeedData ->
5       readAcceleratorData -> readBrakeData -> COMPUTE
6    | off -> COMPUTE_STATUS),
7 COMPUTE = (compute -> trigActions -> ACTIVE).
8 LOG_HMI = (on -> ACTIVE),
9 ACTIVE = (tick1 -> provideHMIData -> log -> ACTIVE
10   | off -> LOG_HMI).
```

Listing 5 The ACC_Application behavior in FSP

components (i.e. *Sensors* and the clocks) is not described in this paper.

The modelled behaviour is verified by means of LTSA and the tool discovers an error when checking the property shown in Listing 6. This property expresses that every `publishDistanceData` (an internal action of the *Sensors* component) must be followed by a `readDistance-Data` in *ACC_Application*. This is not verified by the current behavioural model due to the existence of two different clocks, namely *clk50hzA* and *clk50hzB*, that can create misalignment between *ACC_Application* and *Sensors*. It is important to note that the problem was not visible in the SaveCCM model since actually in SaveCCM the clocks operate at the same rate. In FSP we focus exclusively on functional aspects of the system and then clocks are represented by components sending messages without any temporal regulation. This

```
1 property DistancePublishReadSafety =(
      publishDistanceData -> readDistanceData ->
      DistancePublishReadSafety).
```

Listing 6 Excerpt of the checked property

analysis highlighted a problem of robustness of the current version of the ACC software architecture. In fact, having two different clocks, even if synchronized at the same rate, may lead a designer to modify the rate of a clock without accordingly adapting the other one. Therefore, we decided to refine the ACC model in Darwin/FSP by introducing a new component, *Splitter* in Fig. 14, that receives a message from a single clock and splits it into two messages: one is sent to *ACC_Application* and the other one to *Sensors*. This prevents the designer to break the temporal alignment of the two components.

5.5 Propagating the changes to A_0

The changes made on Darwin/FSP must be propagated back to the other ADLs and thus first to A_0. The model transformation engine produces in A_0 two alternative versions of ACC: *Splitter* can become either an `SAcomponent` or an `SAconnector` in A_0. This is due to the lack of connectors in Darwin/FSP as already discussed while presenting the *DarwinFSP_*A_0 weaving model (see Sect. 5.1.2). Obviously, the choice on how to represent *Splitter* in A_0 cannot be automatized since the model in Darwin/FSP does not have enough semantics to disambiguate this aspect. Between the two generated alternatives, the designer chooses the one in which *Splitter* is represented as an `SAconnector` that better matches with the semantics of *Splitter*. The chosen A_0 model contains all the changes made in Darwin/FSP: the addition of the *Splitter* connector and the behaviour specification of the system.

5.6 Propagating the changes to the whole topology

The next step is to propagate the changes from the A_0 model towards SaveCCM and UML$_{cc}$ models.

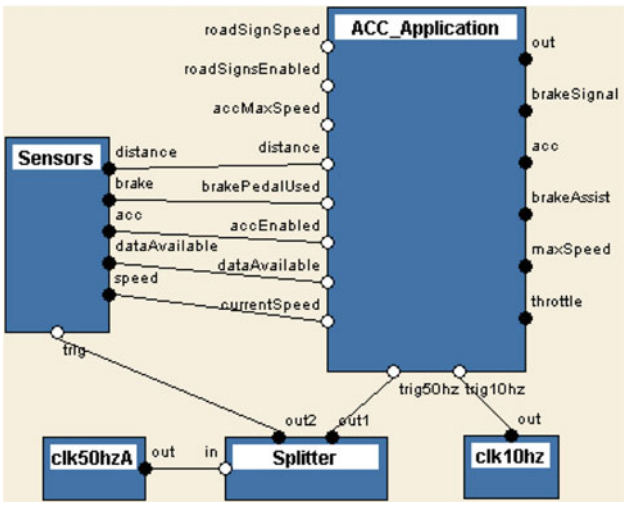


Fig. 14 The Darwin revised version of ACC

When propagating the ACC changes from A_0 to SaveCCM, the transformation engine produces 27 different alternative versions of the ACC model. According to the formula in Sect. 3, we have one non bijective transformation (the one that transforms an `SAinterface` into three different types of ports: data ports, trigger ports or data-trigger ports), and the number of model instance elements that match with the considered transformation is three (only the three ports of the Splitter component been added in Darwin/FSP–the other are disambiguated by the tracing engine). 27 alternatives are obtained by raising 3, i.e. the three instances that match the considered transformation, to the power of 3, i.e. the number of alternatives that the considered transformation has. Among these alternatives, the designer chooses the one in which the three ports of the *Splitter* component are represented as trigger ports. The model is shown in Fig. 15.

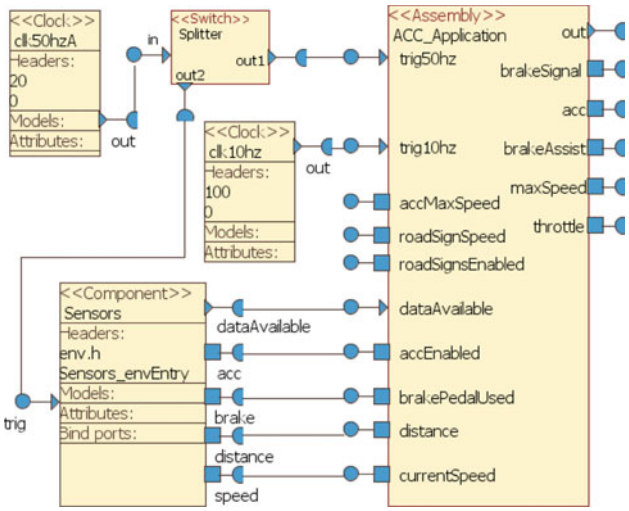


Fig. 15 The revised ACC system in SaveCCM

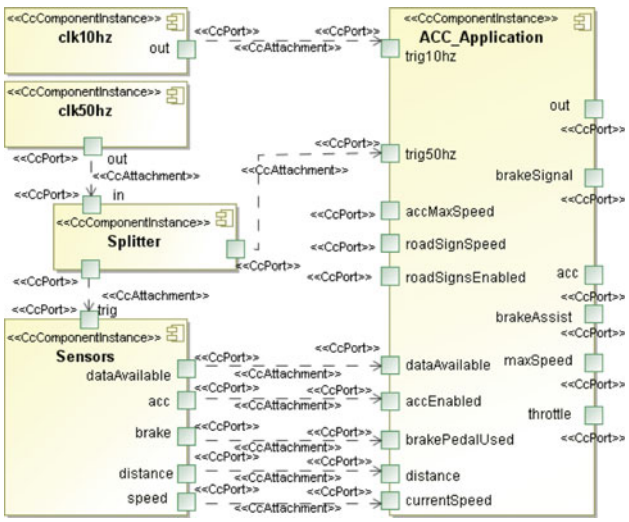


Fig. 16 The revised ACC system in UML$_{cc}$

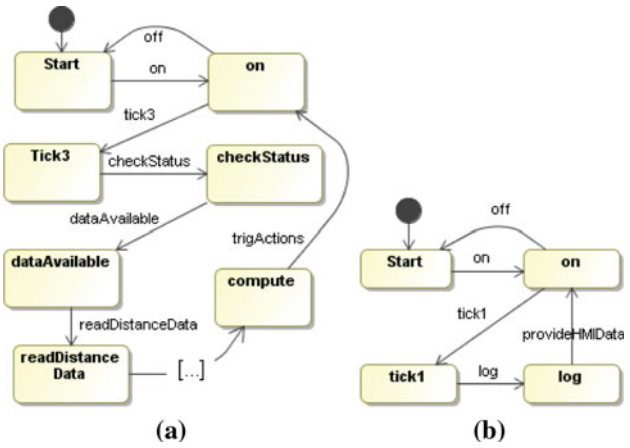


Fig. 17 ACC_Application Behavior

Finally, the UML$_{cc}$ model of ACC is generated starting from the A_0 model. Figure 16 shows the unique alternative proposed by the transformation engine. We obtained only one alternative since in this case the involved transformations are not ambiguous. Figure 17 shows the behaviour of the *ACC_Application* obtained by transforming the behaviour in A_0 and originally added in Darwin/FSP. The obtained UML$_{cc}$ model can be used to generate the required documentation of the system.

6 Considerations and discussions

In this section we provide some considerations about the approach we proposed in this paper.

Our approach poses its bases on the "star" topology of **DUALLY**, with A_0 as the star centre. Our approach makes A_0 models always up-to-date and thanks to the **DUALLY**'s star topology, we can transform and propagate changes towards each other notation in **DUALLY**. In general, changes made on a model conforming to a notation need to be propagated towards several models conforming to other languages. Thanks to the **DUALLY** underlying language topology, the change propagation (one-to-many) is reduced to several consecutive change propagations (each one one-to-one) always from or to A_0. Furthermore, the change propagation result is independent from the order in which the set of one-to-one transformations is executed. This ensures the convergence of the proposed approach. This promotes also the scalability of the approach since the number of transformations to be executed is $n+1$ where n is the number of models to be synchronized. The approach makes use of the logic paradigm ASP, but this language is made transparent to the user that is asked only to define the weaving model, as required in the previous version of **DUALLY**.

In case a transformation is non bijective and it is not instrumented with suitable guards that resolve the ambiguities, the change propagation approach may produce several possible alternative models. As discussed in Sect. 3, the number of alternatives depends on how each transformation is defined and on the number of elements involved in the considered transformation. The approach in its current status delegates to the user the responsibility of selecting what she retains to be "a suitable" model. The changes propagation from ADL_1 to ADL_2 through A_0 requires to select one (suitable model) in A_0 and then one final model in ADL_2. It has to be noted that while the number of alternatives may be theoretically high, it can be considerably reduced and kept acceptable by defining weaving models properly (e.g. considering particular cases and then using guards, avoiding as much as possible that multiple correspondences hold on a single metaclass, etc.) and by following the most convenient transformation journey. In fact, it is important to note that our approach stores the selection made by the architect when transforming from one ADL to another. These choices are stored in order to automatically re-propose the preferred choice in subsequent round-trip journeys. Therefore, starting from the most expressive notation enables the use of the tracing engine which disambiguates the choices when coming back to the source language, not from the less expressive language to the more expressive one.

The experience we made in Sect. 5 confirms this intuition. In particular in the case study, see Sect. 5, we considered three ADLs involved in a round-trip journey composed of six different transformations. 4 transformations produced only one model, 1 transformation produced 2 different versions and finally 1 transformation produced 27 different versions.

Obviously, the number of different versions produced in each transformation depends on the considered ADLs and on how "far" they are from the concepts represented in A_0. More precisely, if an ADL contains very detailed concepts with respect to A_0 (such as the ports specialization of SaveCCM in data ports, trigger ports and data-trigger ports) an extension of A_0 could be realized. The current release of A_0 envisions a large number of extension points (such as business, behavioural aspects) that can be exploited to extend A_0 while maintaining its core unchanged. The extension mechanisms of **DUALLY** are an answer to a limitation of the star topology: since each transformation passes through A_0, when two ADLs, say ADL_i and ADL_j, share some domain-specific concepts that are not contemplated in A_0, these concepts are not optimally propagated. It is clear that it is up to the software architect to decide if and how the extension mechanisms of **DUALLY** should be used.

The experience we had with the case study was encouraging from the point of view of the efficiency of the approach. We used an Intel Pentium D-3.2 GHz, with 4 GB DDR-II of memory RAM, running Windows XP Professional. In each step of the case study the different alternative models have been computed in less than one second. More precisely, the

performance of a model transformation in our approach can be calculated by considering the performance of three main sub-steps:

1. transform the source EMF model into its corresponding ASP encoding. This step consists in executing the transformations at the model level of the technological bridge discussed in Sect. 4.2;
2. execute the ASP-based transformation in order to get the ASP representation of the target models representing the alternatives;
3. transform each produced target model from its ASP encoding into its EMF-based representation. These transformations are the inverse of the transformations executed in the first step.

The last step may cause a degradation of the performance of our approach. For each target model an ATL- and a TCS-based transformation must be executed (see Sect. 4.2 for the details). In case there are a lot of target alternative models, the overall performance of the approach may degrade. This problem is inherent to the ASP-based approach and is due to its requirement to operate on ASP encodings of both the source and target models. It is up to our technological bridge to automatically bridge EMF models and their ASP encoding and vice versa. On one hand the technological bridge is automatic and totally transparent for the software architect, on the other hand it degrades the overall performance of our approach. So, while designing our approach we had to make a choice between usability (through automation) and performance. The additional layer of the technological bridge promotes usability since it allows software architects to deal only with the well-known EMF technologies for representing input and output models and mask the complexity of the ASP-based transformation approach by providing the automatic technological bridge between EMF and ASP encodings. This has a cost in terms of performance of the overall approach; anyway, as stated earlier, the time required to execute each transformation in our industrial case study is more than acceptable since always took less than one second.

The approach we presented in this paper is generic since it depends neither on the notations to synchronize nor on their conforming models. However, as already discussed, the approach strongly relies and depends on A_0. More precisely, A_0 is unavoidable for its role in the star topology in **DUALLY**. In order to generalize the approach to a domain different from software architectures, the star topology should be replicated. Therefore, a new A_0 with the role of pivot element should be defined for the new domain if it is feasible. Once replicated A_0 in the new domain, the overall approach can be reused. The definition of the pivot element is strategic and requires particular attention. The selection of the elements within A_0 was mainly guided by the principle of maintaining the base notation as general as possible to ensure that **DUALLY** is able to potentially represent and support many different architectural notations. The selection phase has been performed by studying architectural languages with purposes similar to **DUALLY**, relevant papers, and UML. Our strategy has been to exploit and inherit features we judged satisfactory, overcoming identified limitations. Moreover, as described earlier, we defined A_0 as extensible in order to give the possibility to software architects to customize it for each specific domain. This is an important aspect to be taken in consideration when defining a new A_0. The kinds of extensions we provided in A_0 are realized by means of the inheritance mechanism; in particular, each element of the A_0 meta-model can be extended.

7 Related work

This section presents state-of-the-art approaches related to our proposal, and a comparison between what presented in this paper and existing work. In particular, Sect. 7.1 outlines approaches for ADLs interoperability, Sect. 7.2 describes state-of-the-art approaches for model-based change propagation, and Sect. 7.3 compares our proposal with existing work on change propagation and interoperability among different ADLs.

7.1 State-of-the-art in ADLs interoperability

Historically, the first attempt to deal with architectural data interchange can be identified in Acme [15]. Acme provides libraries for importing/exporting architecture models designed in different ADLs into/from AcmeStudio, i.e. the Acme tool support. As described in [14], in order to integrate two ADLs, e.g. ADL_1 and ADL_2, the Acme language needs to be augmented (through annotations) with specific information coming from ADL_1, and peculiar to it. Successively, the augmented model needs to be extended even further, with ADL_2 specific information (and therefore, yet again the kernel might itself need augmentation). As claimed in [14], the hard work occurs in the middle step, i.e. when bridging the semantic gap between ADL_1 and ADL_2, and this step is not properly supported.

Based on the observation that a single ADL is not enough to describe the software architecture of real and complex systems, the Institute for Software Research of the University of California, Irvine, proposed xADL [8], a highly extensible ADL based on XML [7]. While xADL is not an interchange language, its extensibility is here advocated as a means for limiting the proliferation of ADLs, thus reducing the need of interoperability. In the same line AADL [12], initially proposed for avionics domains and later on moved to represent and support embedded real-time systems, provides

extension mechanisms. Such extension mechanisms include the definition of custom properties to specify additional ADL-specific analysis and/or generic information to be attached on the architectural design.

7.2 State-of-the-art in changes propagation

A number of existing approaches for model synchronization and change propagation is available today, imposing different restrictions on the underlying transformations. In general, it is required that there is a one-to-one relationship between source and target changes. In point of fact, model transformations are neither total, nor injective, thus posing a number of difficulties and requiring appropriate support for change propagation [18].

An attempt to support change propagation is presented in [38], where an imperative paradigm capable of backtracking, and a language used to handle declarative aspects are used to build a hybrid method trying to deal with tool integration mechanisms. A declarative evolution of the mentioned work is described in [39] where the author illustrates a technique to implement a change propagating transformation language called PMT. After the first mapping from source to target, manipulations can occur on both models. When a new transformation is required, the target model is updated in a non destructive manner by exploiting the trace information generated from the previous execution. In this respect, PMT can be considered as an alternative implementation of change propagation that takes into account (and persists) target modifications when new transformations are performed.

The combination of bidirectionality and change propagation can enable the support of model synchronization. In general, the existing approaches impose relevant restrictions on the characteristics of the involved transformations. For instance, approaches like [13,16,33,41] require the mappings to be *total*, while [16,41,43] impose the existence of some kind of *bijection* between the involved source and target. In particular, [43] proposes to automate model synchronization from model transformations. It is based on QVT Relations and supports concurrent modifications on both source and target models.

Furthermore, in [18] the authors present the formal definition of a round-trip engineering process, taking into account the non-totality and non-injectivity of model transformations.

Another work to mention is xMOF [5] that while focussing on model transformations by means of OCL constraints, aims at providing change propagation.

7.3 Comparison

At the best of our knowledge, very few are the approaches combining interoperability and change propagation at the architectural level.

The approaches presented in Sect. 7.1 deal with architecture languages interoperability, but do not provide any means for dealing with the propagation of changes. While those approaches deal with interoperability through architecture language extensibility, they do not use any model transformation approach, thus not requiring change propagation management. XTEAM [11], an extensible tool chain for evaluating architectural models, while still not treating explicitly changes propagations, consists of a suite of ADLs extensions and model transformation engines to enable the rapid implementation of customized analyses at the architectural level.

By looking at approaches for change propagation (summarized in Sect. 7.2), we can notice that none of them have been targeting software architecture models. An interesting approach bringing together the software architecture and change propagation communities has been proposed in [42]. Its aim is to understand and propagate architectural changes within a product line architecture. The problem this approach aims to tackle is related to capturing architectural changes and propagating them to another product of the product line (within the same ADL, i.e. xADL). In case of conflicts, the merging algorithm, which is the responsible of the changes propagation, abandons the procedure.

Overall, the differences among our proposal and existing work are manifold.

While existing model synchronization and change propagation approaches impose many assumptions on the way transformations are done (e.g. total and bijective transformations), what proposed in this paper enables the support for partial and non injective mappings by using bidirectional transformation. In our opinion this represents a relevant advance, since closer to real requirements on how model transformations are today realized.

Traditional approaches to model synchronization typically provide a non convergent process of change propagations (as illustrated in Sect. 3): the introduction of the star model transformation topology inherited from **DUALLY** provides the great advantage to ensure convergence of the change propagation process.

What is proposed in this paper is automated and integrated into the Eclipse framework: a software architect may transform an architecture model into any other in the start topology and apply changes to one of the models that are automatically propagated to the other models in the network of model transformations.

8 Conclusions and future work

The flourishing of architectural notations and the consciousness that a universal notation acceptable by any architect cannot exist ask for technologies to let tools and notations

interoperate. Several solutions have been proposed to synchronize two different notations, i.e. with the aim to keep models conforming to them in a consistent state. However, when dealing with multiple notations, convergence and scalability problems may arise. In this paper we proposed to extend **D**U**AL**LY, a framework for architectural languages and tools interoperability based on model-driven techniques, with a convergent changes propagation approach between multiple architectural languages. The approach we proposed strongly relies on the underlying languages topology of **D**U**AL**LY that enables correctness and convergence.

The approach is independent from both the notations to synchronize and their conforming models. We experimented the approach, which is implemented in the Eclipse modelling framework, on well-known architectural languages.

As future work we plan to extend the framework with a wizard helping the architect to make decisions among proposed design alternatives. More precisely, the designer will be guided among the different alternatives in a graphical way. The alternatives (that can be many, according to the formula in Sect. 3) are initially partitioned, constrained, abstracted and graphically visualized to the user. Then, when decisions are made, they are stored and used to drive subsequent decisions. We envision a learning approach that deduces from the user choices information to be used to drive next selections. We are also in the process of running a series of experiments to carefully analyse the approach applicability and scalability in complex transformations.

Another interesting future work is to investigate and support collaborative modelling. A first step in this direction can be found in [4] where we proposed a framework for synchronous and asynchronous concurrent and collaborative modelling. Synchronous collaborative modelling offers a mean for sharing the modelling space while asynchronous collaborative modelling supports merging of models modified and edited separately by different designers.

Acknowledgments This work was partially supported by the Italian Government under the project PRIN 2007 D-ASAP (2007XKEHFA) and the project FIRB 2005 ArtDeco (RBNE05C3AH).

Appendix A ASP encodings

In this appendix we present the various ASP encoding of our change propagation approach.

A.1 ASP encoding of metamodels

Our ASP encoding of a metamodel is composed of the terms: *metanode*, *metaprop*, and *metaedge*. In particular, the following term represents the metaclass being encoded (MC) conforming to the metamodel *MM*.

$$metanode(MM; MC)$$

The term *metaprop* represents the attributes of the metaclass *MC*, with a certain name (*NAME*), a certain type (*TYPE*) and conforming to the metamodel *MM*.

$$metaprop(MM; NAME; TYPE; MC)$$

Finally, the term *metaedge* represents associations between metaclasses.

$$metaedge(MM; TYPE; NAME; SMC; TMC)$$

The *MM* element refers to the metamodel, *TYPE* denotes the type of the metarelationships being specified that can be distinguished into *association*, *generalization* and *composition*, *NAME* is the role name of the target association end, *SMC* refers to the source metaclass pointed by the association, whereas *TMC* refers to the target one.

A.2 ASP encoding models

Our ASP encoding for models is pretty similar to that of metamodels; it contains all the elements for encoding a model in ASP: `node`, `edge` and `prop`.

$$node(MM; ID; MC)$$

The term *node* represents an element conforming to the metamodel *MM*, where *ID* is the identity value for the element, and *MC* identifies the metaclass being encoded.

The attributes of a metaclass are expressed by means of the following term:

$$prop(MM; ID; NAME; VALUE)$$

The *MM* element refers to the metamodel, *ID* is the identity value of the class the encoded attribute belongs to, *NAME* is the name of the attribute being encoded, and *VALUE* is the attribute value whose type is specified in the correspondent metamodel class.
Finally, associations between elements is established by means of the following term:

$$edge(MM; ID; NAME; IDSC; IDTC)$$

The *MM* element refers to the metamodel, *ID* is the identity value for the element, *NAME* is the role name of the target association end, *IDSC* refers to the source class pointed by the association, whereas *IDTC* refers to the target one.

Our ASP encoding for models is lossless, i.e. passing from a model into its ASP encoding and then back to the model again results in the same initial model. This is very convenient for us because it allows to automate the bridge between EMF models and their ASP encoding.

A.3 ASP encoding of model-to-model relations

Model-to-model relations describe correspondences among source and target element types at the metamodel level. They are expressed by the following term:

$relation(ID, MMS, MMS)$

The *ID* element is the unique identifier of the relation, *MMS* refers to the source metamodel taken into account, and *MMT* refers to the target metamodel taken into account. This term defines the direction in which the relation holds by setting its source and target metamodels. Bidirectional relations are expressed as *relation(ID, bx, bx)*.
Furthermore, a relation is composed of a pair of this term (one for each metamodel):

$relation(ID, MM, MC, N)$

ID is the unique identifier of the relation, *MM* refers to the metamodel taken into account, and *MC* refers to the metaclass involved in the relation. Several constraints could be needed to specify restrictions on each relation in order to refine the set of generated candidates. If the body of a constraint is satisfied, then the set of atoms related to it is considered to be invalid and consequently the answer set taken into account is not produced.

A.4 ASP encoding of the generic transformation rules

In our ASP encoding, the code implementing the transformation rules is able to create $node'$ elements according to specific relations (defined as the terms described in Sect. A). In particular, the following term checks if a specific mapping can be "executed":

$mapping\_node(MM, ID, MC)$

It evaluates to true if a relation exists and involves elements referring to *MC* and *MC2* metaclasses and an element *node(MM2, ID, MC2)*. In other words, a mapping can be executed each time it is specified between a source and a target, and the appropriate source to compute the target exists.
Furthermore, the following term verifies if a given metaclass is involved in a relation:

$is\_related(MM, MC)$

It evaluates to true if exists a relation *R* between a *MC* metaclass referred to the *MM* metamodel and a *MC2* metaclass referred to the *MM2* metamodel.
The term described below is an auxiliary one and checks if a given metaclass conforms to the target metamodel.

$is\_target\_metamodel\_conform(MM, MC)$

It is true if the *MC* metaclass exists in the *MM* metamodel (i.e. the target metamodel).

There are also terms which compute all the elements that may be generated.

$is\_generable(MM, ID, MC)$

This term groups the results of all the previous checks. Therefore, it is evaluated to true if all the listed conditions have been successful. It is enclosed in curly brackets, meaning that whenever its body is satisfied, the corresponding element (that is, *is_generable(MM, ID, MC)*) may or may not belong to the solution.
Finally, the following term creates target elements if the generation and mapping conditions are satisfied:

$node'(MM, ID, MC)$

A.5 ASP encoding of the tracing mechanism

The terms described in this section represent the code that will manage the tracing information of the transformation. In particular, the term following term is true if the *MC* metaclass is involved in a bidirectional relation.

$is\_metanode\_related(MM, MC)$

In this specific context, the most important term is *trace*:

$trace(MM, IDx, ID, MC)$

It is true if the considered *node* is not involved in a relation or if it is involved on a relation but the condition for its mapping in the target model is not satisfied. The *trace* term refers to the specific element that needs to be stored and regenerated in the backward transformation.

Appendix B Model transformations involved in the approach

In this section we will provide the details of the various ATL transformations involved in our change propagation approach.

In the following we focus on the technological bridge described in Sect. 4.2. By referring to Fig. 6, the *Ecore2ASPmm* transformation is in charge of producing an *ASPmm* model starting from an EMF metamodel. Listing 7 presents an excerpt of the *Ecore2ASPmm* ATL transformation. From a high-level point of view, it produces an ASPmm metanode *x* for each metaclass *y* of the source EMF metamodel (lines 1–21), it creates meta-properties referring to *x* from all the attributes (including the inherited ones) of *y* (lines 22–32) and a set of meta-edges for each super type and all the references of *y* (lines 33–44).

Once the EMF metamodel has been transformed into an *ASPmm* model, a TCS specification performs the final step of producing ASP code. Since the TCS specification defines

```
...
rule EClass2Metanode {
    from
        y : ECORE!EClass
    to
        x : ASPmm!Metanode(
            name <- s.name,
            ...
        )
    do {
        for(e in y.eAllAttributes) {
            thisModule.EAttribute2Metaprop(e, x);
        }
        for(e in y.eAllReferences) {
            thisModule.EReference2Metaedge(e, x, e.eType);
            ...
        }
        for(e in y.eSuperTypes) {
            thisModule.Inheritance2Metaedge(x, e);
        }
    }
}
rule EAttribute2Metaprop(s : ECORE!EAttribute, node : ASPmm!Metanode) {
    to
        t : ASPmm!Metaprop(
            name <- s.name,
            metanode <- node,
            ...
            )
}
rule EReference2Metaedge(s : ECORE!EReference, node : ASPmm!Metanode, target : ASPmm!Metanode) {
    to
        t : ASPmm!Metaedge(
            name <- s.name,
            source <- node,
            target <- target,
            type <- if(s.containment) then #composition else #association endif,
            ...
        )
}
...
```

Listing 7 Excerpt of the Ecore2ASPmm transformation

```
module saveccm2ASPm;
create OUT : ASPm from IN : saveccm;
    ...
    rule Component {
        from s : saveccm!Component
        to  t : ASPm!Node (
            idTrace <- s.__xmiID__.getIdTrace(),
            id <- s.__xmiID__.getId(),
            model <- thisModule.model,
            metaclass <- 'Component'
        )
        do {
            s.eClass().eAllReferences->iterate(e; acc : Sequence(OclAny) = Sequence{} | acc.append(thisModule.
                createEdge(s, e));
            s.eClass().eAllAttributes->iterate(e; acc : Sequence(OclAny) = Sequence{} | acc.append(thisModule.
                createProp(s, e));
        }
    }
```

Listing 8 Excerpt of the generated *SaveCCM2ASPm* transformation

only one-to-one mappings and its realization can be considered just a technical step, we will not further detail this aspect.

At the modelling level, two higher-order transformations generate model-to-model transformations. Depending on the direction in which the software architect is transforming, next step is to execute the model-to-model transformations obtaining an *ASPm* (or EMF) model.

Listing 8 reports a small excerpt of the *SaveCCM2ASPm* transformation; such a transformation has been generated by executing *MM2ASPmGenerator* on the SaveCCM metamodel.

The listing above shows the header of the transformation (lines 1–2) and the rule that handles SaveCCM components and transforms them into *ASPm* nodes (lines 2–16); more specifically, in lines 7 and 8 it initializes the *idTrace* and *id* attributes of a node (described in Sect. 4.4.2), in line 9 it sets the reference to the model containing the node, and in line 10 it sets the name of the metaclass to which the node conforms to; finally, in lines 13 and 14 auxiliary rules are called to

```
1...
2helper context AMW!EquivalenceCorrespondence def : toString() : String =
3'relation(' + self.__xmiID__ + ',bx,bx).\n' +
4'relation(' + self.__xmiID__ + ',' + thisModule.leftMM + ',' + thisModule.getLeftInstance(self.left.element.ref).
     name + ',-1).\n' +
5'relation(' + self.__xmiID__ + ',' + thisModule.rightMM + ',' + thisModule.getRightInstance(self.right.element.
     ref).name + ',1).\n' +
6...
7':-node(' + thisModule.leftMM.normalize() + ',IDx,ID,' + thisModule.getLeftInstance(self.left.element.ref).name +
      '),not␣node\'(' + thisModule.rightMM + ',IDx+1,ID,' + thisModule.getRightInstance(self.right.element.ref).
     name + '),mmt==' +                  thisModule.rightMM + '.\n' +
8...
9self.featureEquivalences->iterate(e; acc : String = '' | acc + e.toString('1')) +   '\n';
```

Listing 9 Excerpt of the *ASPrelationsGenerator* transformation

create properties and edges related to the node being created. The final step of the bridge is implemented in TCS.

Focussing on the generation of ASP relations from weaving models (see Fig. 12), the *ASPrelationsGenerator* is an ATL transformation in charge of analysing each correspondence contained into a weaving model and producing its corresponding ASP code. Listing 9 shows a fragment of the *ASPrelationsGenerator* transformation; more specifically, it represents the helper (i.e. a sort of method in ATL) that generates the ASP code corresponding to an *EquivalenceCorrespondence* of the weaving model. This specific ASP fragment produces three relation facts (lines 3–5) and a constraint (line 7); it also calls another *toString* helper for each *FeatureEquivalence* contained into the current correspondence (line 9).

References

1. Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE-Std-1471-2000. The Institute of Electrical and Electronics Engineers (IEEE) Standards Board, September 2000
2. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the Large and Modeling in the Small. In: LNCS, vol. 3599, pp. 33–46 (2005)
3. Cicchetti, A., Di Ruscio, D., Eramo, R.: Towards Propagation of Changes by Model Approximations. In: Proceedings of the 10th IEEE on International Enterprise Distributed Object Computing Conference Workshops EDOCW, p. 24. IEEE Computer Society, Washington, DC, 16–20 October 2006
4. Cicchetti, A., Muccini, H., Pelliccione, P., Pierantonio, A.: Towards a Framework for Distributed and Collaborative Modeling. In: WETICE, pp. 149–154 (2009)
5. Compuware, S.: XMOF queries, views and transformations on models using MOF, OCL and patterns. OMG Document ad/2003-08-07 (2003)
6. Crnkovic, I., Malavolta, I., Muccini, H.: A model-driven engineering framework for component models interoperability. In: Poernomo, I., Hofmeister, C., Lewis, G.A. (eds.) Proceedings of the 12th International Symposium on Component-Based Software Engineering (CBSE 2009). LNCS, vol. 5582. Springer, Berlin (2009)
7. Dashofy, E.M., Van der Hoek, A., Taylor, R.N.: A Highly-Extensible, XML-Based Architecture Description Language. In: WICSA'01 (2001)
8. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In: ICSE '02, pp. 266–276. ACM Press, New York (2002)
9. Di Ruscio, D., Malavolta, I., Muccini, H., Pelliccione, P., Pierantonio, A.: Developing Next Generation ADLs through MDE Techniques. In: ICSE 2010 (2010)
10. Didonet Del Fabro, M., Bézivin, J., Jouault, F., Breton, E., Gueltas, G.: AMW: A Generic Model Weaver. In: IDM2005 (2005)
11. Edwards, G., Medvidovic, N.: A Methodology and Framework for Creating Domain-specific Development Infrastructures. In: ASE'08 (2008)
12. Feiler, H.P., Lewis, B., Vestal, S.: The SAE Architecture Analysis and Design Language (AADL) Standard. In: IEEE RTAS Workshop (2003)
13. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3), 1–65 (2007)
14. Garlan, D., Monroe, R., Wile, D.: Acme: An architecture description interchange language. In: Proceedings of CASCON 97, pp. 169–183. Toronto, Ontario (1997)
15. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In: Foundations of Component-Based Systems, pp. 47–68. Cambridge University Press, Cambridge (2000)
16. Giese, H., Wagner, R.: Incremental Model Synchronization With Triple Graph Grammars. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) Procs. of the 9th Int. Conf. on Model Driven Engineering Languages and Systems, MoDELS 2006, Genova, Italy. Lecture Notes in Computer Science, vol. 4199, pp. 543–557. Springer, Berlin (2006)
17. Hansson, H., Akerholm, M., Crnkovic, I., Torngren, M.: Save-ccm—A Component Model for Safety-critical Real-time Systems. In: EUROMICRO '04: Proceedings of the 30th EUROMICRO Conference, pp. 627–635. IEEE Computer Society, Washington, DC, USA (2004)
18. Hettel, T., Lawley, M., Raymond, K.: Model Synchronisation: Definitions for Round-Trip Engineering. In: Proceedings of the 1st International Conference on Theory and Practice of Model Tranformations, ICMT 2008. ETH Zrich, Switzerland, 1–2 July 2008
19. ISO. Fourth working draft of Systems and Software Engineering—Architectural Description (ISO/IECWD4 42010). Working doc.: ISO/IEC JTC 1/SC 7 N 000 (2009)
20. Ivers, J., Clements, P., Garlan, D., Nord, R., Schmerl, B., Rodrigo Oviedo Silva, J.: Documenting Component and Connector Views with UML 2.0 Technical Report CMU/SEI-2004-TR-008, CMU, SEI (2004)
21. Jouault F., Bézivin, J.: KM3: A DSL for Metamodel Specification. In: FMOODS06. LNCS, vol. 4037 (2006)
22. Jouault, F., Bézivin, J., Kurtev, I.: TCS: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: Proceedings of the 5th international conference on Generative programming and component engineering, pp. 249–254. ACM Press, New York (2006)
23. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: MoDELS 2005 (2006)

24. Kerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P., Tivoli, M.: The save approach to component-based development of vehicular systems. J. Syst. Softw. **80**(5), 655–667 (2007)
25. Kruchten, P., Lago, P., van Vliet, H.: Building up and reasoning about architectural knowledge. In: Proceedings of the 2nd International Conference on the Quality of Software Architectures (QoSA), pp. 43–58 (2006)
26. Magee, J.: Behavioral Analysis of Software Architectures using LTSA. In: ICSE '99, pp. 634–637. ACM, New York (1999)
27. Magee, J., Kramer, J.: Dynamic structure in software architectures. SIGSOFT Softw. Eng. Notes **21**(6), 3–14 (1996)
28. Magee, J., Kramer, J.: Concurrency: State Models & Java Programs. Wiley, New York (1999)
29. Magee, J., Kramer, J., Giannakopoulou, D.: Software architecture directed behaviour analysis. In: IWSSD'98 (1998)
30. Malavolta, I., Muccini, H., Pelliccione, P., Tamburri, D.: Providing architectural languages and tools interoperability through model transformation technologies. IEEE TSE **36**(1), 119–140 (2010)
31. Medvidovic, N., Dashofy, E.M., Taylor, R.N.: Moving architectural description from under the technology lamppost. Inf. Softw. Technol. **49**, 12–31 (2007)
32. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**, 70–93 (2000)
33. Mu, S.-C., Hu, Z., Takeichi, M.: An Injective Language for Reversible Computation. In: Kozen, D., Shankland, C. (eds.) Procs. of the 7th Int. Conf. on Mathematics of Program Construction, (MPC 2004), Stirling, Scotland, UK, July 12–14. Lecture Notes in Computer Science, vol. 3125, pp. 289–313. Springer, Berlin (2004)
34. Pelliccione, P., Inverardi, P., Muccini, H.: Charmy: a framework for designing and verifying architectural specifications. IEEE TSE **35**(3), 325–346 (2009)
35. Perry, D.E., Wolf, A.L.: Foundations for the Study of Software Architecture. In: SIGSOFT Soft. Eng. Notes, vol. 17 (1992)
36. Sentilles, S., Pettersson, A., Nyström, D., Nolte, T., Pettersson, P., Crnkovic, I.: Save-IDE—A Tool for Design, Analysis and Implementation of Component-based Embedded Systems. In: Proceedings of the Research Demo Track of the 31st International Conference on Software Engineering (ICSE'09), pp. 607–610, (May 2009)
37. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. Wiley, New York (2009)
38. Tratt, L.: Model transformations and tool integration. SOSYM **4**(2), 112–122 (2005)
39. Tratt, L.: A change propagating model transformation language. J. Object Technol. **7**(3), 107–126 (2008)
40. University of Helsinki. SMODELS solver Website (2008). http://www.tcs.hut.fi/Software/smodels/
41. Van Paesschen, E., De Meuter, W., D'Hondt, M.: SelfSync: A Dynamic Round-trip Engineering Environment. In: Briand, L.C., Williams, C. (eds.) Procs of the 8th Int. Conf. on Model Driven Engineering Languages and Systems, (MoDELS 2005), Montego Bay, Jamaica, October 2–7. Lecture Notes in Computer Science, vol. 3713, pp. 633–647. Springer, Berlin (2005)
42. Van Der Westhuizen, C., Van Der Hoek, A.: Understanding and Propagating Architectural Changes. In: Proceedings of the IFIP 17th World Computer Congress - Tc2 Stream/3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance. IFIP Conference Proceedings, vol. 224, pp. 95–109. Kluwer B.V., Deventer, The Netherlands (2002)
43. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Supporting Parallel Updates with Bidirectional Model Transformations. In: ICMT (2009)

Author Biographies

Romina Eramo is a Ph.D. student in the Computer Science Department at the University of L'Aquila, Italy, from 2006. Her main research interests include Model Driven Engineering and, in particular, bidirectional model transformations and techniques for model synchronization.

Ivano Malavolta is working toward the Ph.D. degree in computer science in the Computer Science Department at the University of L'Aquila, Italy. His research interests include software architecture languages (ADLs), architectural interchange and interoperability between software architecture notations, and MDE techniques.

Henry Muccini is an Assistant Professor at the University of L'Aquila, Italy. Henry's research interests are on software architecture modelling and analysis, component-based systems, model-based analysis and testing, and global software engineering education. He has published various conference and journal articles on these topics, co-edited two books, and co-organized various workshops on related topics. He serves as PC member and reviewer in many international conferences and journals. He is currently locally coordinating the GSEEM (Global Software Engineering European Master) and an Erasmus Mundus External Cooperation Window project. More information is available at http://www.HenryMuccini.com.

Patrizio Pelliccione is an assistant professor at the University of L'Aquila, Computer Science Department. He got his Ph.D. degree in the University of L'Aquila, computer science department. The research topics are mainly in Software Architectures modeling and analysis, Model checking, and Formal Methods. Patrizio published various journal and conference papers in these topics and serves as PC member and reviewer in several international conferences and journals. More information are available at http://www.di.univaq.it/pellicci.

Prof. Alfonso Pierantonio is Associate Professor in computer science at the University of L'Aquila (Italy); he is currently director of the Master in Web Technology degree program. His current research interests include Model-Driven Engineering and in particular the theory and practice of model versioning/evolution with a specific emphasis on coupled evolution. In particular, he investigated the problem of co-evolution between metamodels and other artifacts in order to define the basis for their (semi) automatic adaptation. He has been and is currently part of program and organization committees of conferences and has been among the initiators and in the steering committee of the International Conference on Model Transformation (ICMT). He co-edited several special issues on Model Transformations which appeared on Science of Computer Programming and are about to appear on the Intl. Journal of Software and Systems Modeling. Additional information can be found on his webpages: http://www.di.univaq.it/alfonso.