A Transformation Framework for the Compositional Interchange Format for Hybrid Systems*

D. Hendriks * R.R.H. Schiffelers*** M. Hüfner***
C. Sonntag***

* Systems Engineering Group, Eindhoven University of Technology,
The Netherlands, d.hendriks@tue.nl
** ASML, The Netherlands, ramon.schiffelers@asml.com
*** Process Dynamics and Operations Group, Technische Universität Dortmund, Germany
{martin.huefner|christian.sonntag}@bc.tu-dortmund.de

Abstract: The purpose of the Compositional Interchange Format for hybrid systems (CIF) is to establish inter-operability of a wide range of tools by means of model transformations - using the CIF as intermediate, the implementation of many bi-lateral translators between specific formalisms can be avoided. This paper presents the architecture of the CIF transformation framework. Languages in the CIF transformation framework are defined by means of conceptual models, and transformations are specified by means of transformation languages. To avoid large monolithic transformations, transformations are divided into many small single- or cross-formalism transformations, each with their own concern. In this way, transformations are obtained that are easier to understand and can be re-used individually. A domain-specific language TOOLDEF has been developed to specify complex, automated tool chains that are based on these small individual transformations. The TOOLDEF architecture is illustrated using a translation between the CIF and UPPAAL, and an example for the application of the transformation framework is given based on a transformation between the CIF and gPROMS.

Keywords: Compositional Interchange Format, Model transformations, Model exchange, Hybrid systems, Formal semantics, Equation-based languages

1. INTRODUCTION

Model-based methods for the design and analysis of control systems for technical systems are becoming increasingly important in industry since they offer many advantages over traditional design methodologies, such as increased safety, a more economical operation of the controlled system, and a significant reduction of the design duration which translates to large financial savings. In recent years, sophisticated methods and tools have been developed for all stages of control systems design, ranging from requirements analysis and simulation to the synthesis of optimal and robust controllers and verification. A major obstacle in the application of model-based techniques is the incompatibility of model formalisms, as most of the existing software tools are based on different model formalisms, and existing export and import functions are mainly limited to the exchange of results. The translation of models must therefore be carried out manually, which is time-consuming, error-prone, and expensive. To overcome this obstacle, a framework for the integration of model-based design and analysis tools into a coherent and transparent design flow is currently developed in the European MULTIFORM project (MULTIFORM (2008)), where the interconnection of design and analysis tools via the exchange of models between a large number of formalisms, such as Matlab/Simulink, MODELICA (Modelling Assocation (2011)), gPROMS (Process Systems Enterprise Ltd. (PSE) (2010)), UPPAAL (Larsen et al. (1997)), PHAVER (Frehse (2005)); logic controllers as SEQUENTIAL FUNCTION CHARTS (Sonntag and Fischer (2010)), and many others, assumes an important role. The model exchange is realized by defining bi-directional translations between proprietary modeling formalisms and a powerful interchange format, the Compositional Interchange Format (CIF) for hybrid systems.

The main purpose of the CIF (Beek et al. (2007a, 2008); Baeten et al. (2010)) is to establish inter-operability of a wide range of tools by means of model transformations. Using the CIF as intermediate, the implementation of bi-lateral translators between specific formalisms can be avoided. The CIF provides a generic modeling formalism and tools for a wide range of untimed, timed, and hybrid systems. An overview of previous related work on interchange formalisms can be found in Beek et al. (2007a). The concepts in the CIF and the relations between them are defined in a conceptual model or metamodel (Baeten et al. (2010)). This model is defined in terms of Ecore class diagrams (Steinberg et al. (2009)) and OCL (Object Man-
agement Group (2006-2009)) constraints. The behavioral semantics of the CIF (Baeten et al. (2010)) is formally defined by means of deduction rules in a SOS style (Plotkin (2004)). It defines the mathematical meaning of a hybrid model in terms of a hybrid transition system (Cuijpers et al. (2002)). More information about CIF, CIF applications, and CIF tools can be found in Systems Engineering Group TU/e (2008); Sonntag et al. (2009).

To establish tool inter-operability, a transformation framework for the CIF has been developed. In this paper, the architecture of the CIF transformation framework and its design decisions are described. The domain-specific language TOOLDef has been developed to specify tool chains that are based on these small individual transformations. The TOOLDef architecture is demonstrated using a transformation from the CIF to the UPPAAL model checker\(^1\), and the transformations are illustrated using a transformation from the equation-based modeling language GPROMS to the CIF.

2. THE CIF TRANSFORMATION FRAMEWORK

In the CIF transformation framework, languages are defined using conceptual models, also known as metamodels. A conceptual model (see Fig. 1) represents concepts (entities) and their relationships. The conceptual model of the CIF is described using (Ecore) class diagrams (Steinberg et al. (2009)), where classifiers represent concepts, and associations represent relationships. The advantage of a conceptual model is that during the design of a language, it is not necessary to deal with concrete syntax. After the definition of the language, a concrete textual and/or a graphical syntax for the language can be defined. There are two kinds of classifiers, data types and classes (depicted as rectangles). Data types are used for simple types and are identified by a name. Examples of data types include Boolean, number, string data types (optionally restricted using regular expressions), and enumerations. A class is identified by its name as well and can have a number of attributes and references. Classes support inheritance, giving them access to the structural features of their supertype (where the name of abstract classes is shown in italic font). Inheritance relations are depicted as edges between two classes with a (non-solid) triangle on the side of the superclass. Attributes are identified by name and have a data type (shown inside the rectangle of the class). Associations between classes are modeled by references (depicted as an edge between two classes). Like attributes, references are identified by name and have a type that is given by the class at the other end of the association. A reference specifies lower and upper bounds on its multiplicity. The multiplicity indicators that can be used are defined as follows: \( n \) means exactly \( n \) (where \( n \geq 1 \)), \( n..m \) means \( n \) up to and including \( m \) (where \( n \geq 0 \), \( m \geq 1 \), and \( m > n \)), and \( n.* \) means \( n \) or more (where \( n \geq 0 \)). A reference can also represent a stronger type of association, called containment (depicted as a solid diamond at one end). The approach is illustrated by means of the most basic language construct of the CIF: an automaton.\(^2\).

Fig. 1 shows the conceptual model of an automaton. A CIF automaton contains actions, locations, and edges between locations. Automata, locations, and actions have a name, an automaton contains exactly one initial location, and edges can be labeled with an action.

For a language, many different metamodels can be defined. For example, for a BNF (Backus-Naur Form) specification, a metamodel representing the abstract syntax tree (AST) can be defined using containment relations only. An instance of such a metamodel is a tree of language elements, and additional scope (reference) resolving is required to map identifiers to objects. A metamodel can also be developed using (non-containment) relations, as shown by the action reference from the Edge class to the Action class in Fig. 1. In addition, the CIF automaton can be defined using the tree approach: Instead of an action reference from the Edge class to the Action class, the Edge class may contain an attribute ActionName. Based on this attribute, scope resolving would return the Action with name ActionName. It is possible to include static type information for variables and omit type information for the operators and value nodes of expression trees. Additional type computing is then required in order to determine whether expressions are type-correct. Alternatively, type information on all nodes of expression trees can be included. In the CIF transformation framework, it was chosen to design the conceptual models of languages such that instances of these metamodels are fully static type-annotated graphs. In this way, 1) transformations do not have to deal with issues related to type computing and scope resolving; 2) transformations only have to deal with mappings between concepts; and 3) the metamodel abstracts away from parser-related elements that are often contained in language definitions.

The class diagram of a language contains static semantic constraints such as every automaton contains exactly one initial location. However, other static semantic constraints such as within an automaton, the names of locations should be different cannot be expressed using class diagrams. In order to define such static semantic constraints, the Object Constraint Language (OCL) (Object Management Group (2006-2009)) can be used. For example, the aforementioned constraint can be specified using OCL as:

\[
\text{context } \text{Automaton}
\text{inv } \text{AutLocUniqueNames}: \\
\text{self. locations } \rightarrow \text{ forAll(x,y | x.name = y.name implies x=y)}
\]

The first line (context Automaton) states that the invariants apply for instances of the class Automaton. Invariants are of

\(^1\) The translation details are described in Nadalos Agut et al. (2011)

\(^2\) The conceptual model of the CIF consists of 131 classifiers and is too large to show or explain here.
the form `inv <name>::<booleanExpression>`, where `<name>` and `<booleanExpression>` denote the name of the invariant and the expression that should be satisfied. The expression syntax itself resembles common mathematical operators such as universal quantification or set union. Using OCL, all static semantic constraints for the CIF can be specified. A class diagram and the accompanying OCL constraints form the conceptual model of a language. Although the development of these conceptual models for existing languages is an additional effort, it is very likely that the aforementioned advantages outweigh the additional effort.

### 2.1 Transformations

In the CIF transformation framework, transformations between languages are called Model-to-Model (M2M) transformations (see Fig. 2). In this example, the M0 level contains three models, $M_a$, $M_t$, and $M_b$, each conforming to their own metamodel (at level M1). Model $M_a$ is the source model to be translated using transformation model $M_t$, which is specified in the transformation language $MM_t$ and based on $MM_a$ and $MM_b$, to model $M_b$. The metametamodel $MMM$ (at level M2) denotes the language in which the metamodels are defined. The subset of $MM_a$ models that is accepted as input is the domain of the transformation (function), and the subset of $MM_b$ models that can be generated is the range of the transformation.

Transformations are divided into many small, single- or cross-formalism transformations, each with their own concern. In this way, transformations are obtained that are easy to understand and can be re-used. Fig. 3 shows how the CIF transformation framework deals with cross-formalism transformations. The first task is to (manually) determine the subsets of the two formalisms $A$ (with exemplary model instances $Ta1$, $Ta2$ in Fig. 3) and $B$ (with instances $Tb1$ - $Tb3$) that can be translated via the CIF (yellow striped, central circles). Then, cross-formalism M2M transformations of those subsets are defined for both directions. Concepts that are directly transformable are, e.g., the variable concept, the arithmetic operators and literals, i.e. concepts from the same domain. These transformations can be formalized and automated using a M2M transformation language. Thus, it should in principle be possible to transform models within these subsets with a lossless round trip (possibly after reaching a fixed-point). Subsequently, the source and target subsets may be widened by means of M2M transformations within each formalism, if needed. For a model that is not part of the transformable subset, a (possibly model-specific) transformation can be defined by re-using/extending the basic transformations.

Since the CIF has formal semantics, transformations defined on the CIF can be proven to be correct, if the transformed languages are also equipped with formal semantics. Then, it is possible to transform models from one formalism via the CIF (possibly using M2M transformations within the CIF) to another formalism, where these M2M transformations bridge different language concepts of the two languages. Based on transformable language subsets, algorithms can be developed that, given a source model, a target language, and a set of transformations, try to find one or more sequences of transformations to perform the requested transformation. How to formally specify the subsets of transformations, and how to implement such algorithms efficiently, is left as future work.

### 2.2 Behavioral Semantics

Depending on the availability of a formal definition of the behavioral semantics of a language, two different categories of transformations can be distinguished: A) transformations from formalisms that have formal semantics to the CIF (and vice versa), and B) translations from formalisms that do not have formal semantics to the CIF (and vice versa). If both formalisms have formal semantics, an equivalence relation\(^3\) between the semantics of the two formalisms can possibly be defined. By means of mathematical proof, it can be shown that the behavior of the input model and of the output model of the transformation are equivalent. Examples of such correctness proofs can be found in e.g. Man and Schieferle (2006); Beck et al. (2007b); Bortnik et al. (2005); Nadales Agut et al. (2011). If the source formalism does not have formal semantics and the target formalism has formal semantics, formal semantics is given to the source formalism through a (formally defined) transformation. An example of this approach can be found in Sonntag and Hüfner (2011a,b) where bi-directional transformations between MODELICA and the CIF (see Section 3.3) are defined. The correctness of the transformations has been validated by means of comparing the simulation results of several input and output models.

### 2.3 Generic tools in the CIF transformation framework

The CIF transformation framework contains several generic tools to support the development of languages, models,\(^3\) In fact, multiple equivalence relations can be defined depending on the properties to be preserved.
and model transformations. These deal with support of the modeler in the form of predefined copy and walker transformations, automated documentation and error generation, model-specific transformations, and a language for the development of complex (automated) tool chains.

To facilitate modeling transformations within one formalism, the CIF transformation framework contains tooling to generate an identity (copy) transformation for a given metamodel. Single-formalism transformations can extend this transformation with specific transformation rules. In addition, the CIF transformation framework contains tooling to generate a QVTo walker transformation that "walks" over the containment relations of a model and calls customizable processing helpers for each element (object) in the model tree (and for all its super-classes). The walking logic is completely hidden from the modeler - only the classes that actually require processing code must be implemented.

A problem with the implementation of constraints using OCL is the generation of user-friendly error messages, as OCL does not contain logging constructs to provide error information to the user. In this work, the operational Query View Transformation Operational mappings (QVTo) (Object Management Group (2006-2009)) implementation from the Eclipse Model-to-model project was chosen to implement the OCL constraints, since for complex actions, operational code is easier to write than declarative code. The benefit of using QVTo is that the expression part is exactly OCL. Using QVTo helper functions, it is, for instance, possible to construct a 'symbol table' while walking through the graph. In this way, it is possible to gather sufficient information that can be provided to the user using the QVTo log statement. While in OCL, the order in which the objects are checked is implicit, this needs to be coded manually in QVTo. Besides logging facilities, the CIF transformation framework contains tooling to generate a \LaTeX\ documentation skeleton for a given metamodel.

As stated above, a transformation between two formalisms consists of many small transformations that do not contain any model-specific information, i.e., they can be applied on arbitrary models. However, if this is not possible for a particular model, additional external information of the modeler can be integrated to define model-specific transformations for models for which the general transformation is not feasible.

Finally, the CIF transformation framework provides a flexible, user-definable way to specify transformation chains by means of the ToolDef language (see also Section 3.2). A ToolDef model contains a tool chain definition that is executed when the ToolDef model is interpreted. To this end, an interpreter has been developed such that ToolDef specifications can be interpreted (executed) in a manner. A tool chain defines all the steps (tool instantiations) required for a single task or experiment. In a tool chain, static variables can be declared, and the steps to be executed are specified by means of statements, including assignment, sequential composition, parallel composition, conditional execution and repetition. Furthermore, it is possible to load and save models, as well as to execute simple, predefined tools. New tools can be integrated by means of tool definitions that can be called using tool instantiations. Such tool definitions are parameterized tool chains and enable the hierarchical definition of tool chains, as well as reuse. Tool definitions specified in ToolDef libraries can be imported into other ToolDef models. Note that the ToolDef language is very general and can be used in contexts that are not related to the CIF, or even to model transformations.

3. EXAMPLE TRANSFORMATIONS

3.1 Transformations on the CIF

The goal of CIF is not only to serve as an exchange vehicle, but also to provide a sound basis for modeling hybrid systems. Thus, both a concrete textual and a graphical syntax have been defined. Within the CIF transformation framework, transformations have been defined and implemented that transform models specified in one of these concrete syntaxes to an instance of the conceptual model of the CIF. To facilitate modeling, extended syntactic concepts (syntactic sugar) and an automata definition/ instantiation mechanism have been defined. These modeling concepts get (behavioral) semantics by means of a formally defined mapping to the CIF core concepts. Following the previously mentioned paradigm of defining multiple, small, orthogonal transformations that can be re-used individually, transformations such as elimination of each modeling concept to core concepts, flattening of automaton definitions and instantiations, and process-algebraic linearization (elimination of operators such as parallel composition) have been defined on the CIF. Depending on the concepts present in the target language, the user can thus choose whether a particular modeling concept has to be eliminated or not.

3.2 UPPAAL and CIF

This section describes the application of the ToolDef language and implementation for the transformation of CIF models to the UPPAAL model checker (Larsen et al. (1997)) . Using QVTo, (executable) transformations between the conceptual model of the CIF and the conceptual model of UPPAAL, as well as transformations from and to the UPPAAL conceptual model, have been specified. In the cross-formalism transformations, most of the structural information (automaton definitions/instantiations, parallel composition) is preserved. However, in case a CIF model cannot be translated using these transformations, it is always possible to eliminate the CIF operators, such as parallel composition, from the CIF by means of the CIF linearization transformation, and then transform the result to UPPAAL. The ToolDef model of the transformation from the CIF graphical notation to UPPAAL XML notation is shown (partly) below:

\begin{verbatim}
toolchain TurnTable = [ var turntableCifg; Model := ( turntableCifg := load("turntable.cifg "); toolinst cifg2uppalaxml(turntableCifg, "output_dir", "turntable.xml") )]
\end{verbatim}

4 A detailed, formal definition of the transformation from CIF to UPPAAL can be found in Nadales Agut et al. (2011).
Fig. 4. Exemplary transformation of a bouncing ball model from gPROMS to the CIF

```cif
tooldef cif2uppaalxml(var cifgModel: Model, var uppaalXmlPath: String, var uppaalXmlFile: String) =
{  var cifModel : Model; var uppaalModel : Model = ( cifModel := toolinst cif2cif compactcs2voidacts(cifgModel); uppaalModel := toolinst cif2uppaal(cifModel); toolinst uppaal2uppaalxml(uppaalModel, uppaalXmlPath, uppaalXmlFile) )
}

```

The model contains a tool chain called TurnTable that loads the turntable.cif file (containing a model in the CIF graphical notation) into a variable turntableCif, and transforms it using the tool cif2uppaalxml which saves the result in the file turntable.xml. The tool cif2uppaalxml is defined using a definition that transforms the CIF model to an instance of the CIF conceptual model (stored in variable cifModel) and performs a CIF-to-CIF transformation (cf2cif compactcs2voidacts) to end up in the subset of CIF that can be transformed to UPPAAL. Using the M2M transformation cif2uppaal, the corresponding UPPAAL model is obtained. Finally, the M2T transformation uppaal2uppaalxml results in an XML UPPAAL file that is readable by the UPPAAL model checker. The tool definition cif2uppaal validates whether its input conforms to the corresponding conceptual model, and whether the input is contained in the domain of the transformation to be performed. If this validation is successful, the corresponding (simple) tool (cifto) is instantiated.

3.3 gPROMS and CIF

gPROMS is an equation-based modeling language that offers hierarchical structures, equation sections to describe (hybrid) dynamics, and algorithm sections to realize forced actions. Using QVTo, M2M transformations between the conceptual models of CIF and gPROMS have been specified. The resulting models are subsequently transformed to the concrete textual file formats. The transformation is illustrated with a concrete translation example of a bouncing ball model, as shown in Fig. 4. Part (a) of Fig. 4 shows the gPROMS representation of the bouncing ball. A process, which is the top-level element in gPROMS, instantiates a gPROMS model, which holds the (hybrid) system dynamics. This is translated to an equivalent CIF automaton definition (part (c)) and instantiation (part (b)), thereby connecting the variables. The algorithmic scheduling section of the process implements the re-initialization of the system variables when the ball hits the floor. This is transformed into several parallel automata in the main CIF model. More generally, an equation section of gPROMS is transformed into a set of automata, where the (conditional) equations are stored in the discrete locations. The scheduling section of gPROMS is transformed into a network of automata, in which location/transition elements represent algorithmic statements. Using a top-down, recursive translation approach, the gPROMS

6 The transformation is based on M2M transformations on the conceptual models, but is shown here in concrete text syntax.
model hierarchy can be preserved by using automation definitions and instantiations in the CIF. The transformation from CIF to gPROMS requires more preprocessing on the respective source model than transformations between the CIF and other automation-based languages, because the compatible subsets of the CIF and gPROMS are smaller.

For the transformation from the CIF to gPROMS, a flat CIF source model without any 'syntactic sugar' of the CIF modeling extensions is used. To this end, transformations on the CIF described above ('elimination of modeling extensions' and "linearization"), are applied to the source model, reducing the source model's language to a CIF subset which contains only a single location and self-loops which can directly be transformed to gPROMS. The transformation procedures have been validated on several test models, including the bouncing ball example. In all cases, the differences in the simulation results of transformed models were within the simulation tolerance in comparison to the source models.

4. CONCLUSIONS AND FUTURE WORK

In this paper, a transformation framework for the Compositional Interchange Format (CIF) has been presented that allows to easily establish inter-operability of a wide range of model-based tools by means of model transformations. Languages in the CIF transformation framework are defined by means of conceptual models, and transformations are specified by means of transformation languages. In addition, a domain-specific language TOOLDEF has been developed for the convenient specification of tool chains. The specification of conceptual models and transformations are based on OMG standards, such as class diagrams, OCL, and QVT which can be used in an Eclipse environment in a platform-independent manner. The TOOLDEF language has been integrated into the software-based Design Framework that is currently being developed within the MULTIFORM project. Therefore, defined model transformations can be executed seamlessly from within the design process.

Future work entails, amongst others, 1) the development of formal semantics for languages for which currently only informal semantics are available; 2) the efficient specification of the domains of transformations and the development of algorithms that enable a (semi-)automatic transformation based on these domains, and 3) the extension to other model-based languages such as Matlab/Simulink and EcocimPro.

REFERENCES


7 See also Fischer et al. (2011); Sonntag and Hünfner (2011b).

12514