Model-Driven Diagnostics Generation for Industrial Automation

M. Behrens, G. Provan, M. Boubekeur, A. Mady
CCSL, Computer Science Department,
University College Cork, Cork, Ireland.
{mb20, g.provan, m.boubekeur, mae1}@cs.ucc.ie

Abstract- **We propose a methodology for overcoming the current approach of writing diagnostics code for industrial automation applications after the system is designed, which results in significant extra effort/cost, and potential discrepancies between design and diagnostics output. We show how we can automatically generate a diagnostics from a more complex simulation model. We show how a model-transformation tool, ATL, can transform a hybrid-systems simulation model into a propositional-logic diagnostics model with appropriate transformation rules. We discuss issues of correctness of model transformation.**

I. INTRODUCTION

One key drawback to model-driven design of large systems is the cost of constructing appropriate system models. This issue is compounded when multiple models must be built, as is typically the case when one model is constructed for design and simulation, and a separate model constructed for diagnostics.

We propose an approach for reducing the need to construct multiple models by using a meta-modeling and model-transformation approach. We assume that we can specify a *generic meta-model*, which is a detailed model that identifies a key set of properties of a system. For example, this model may be a detailed simulation model that includes control and sensor configurations. Given generic meta-models for a source and a target-system, we show how we can auto-generate an application-specific model. In particular, we show how we can generate a discrete diagnostics model from a hybrid-systems model.

Our approach has several key advantages. First, it can automatically generate arbitrary instances of target (e.g., diagnostics) models from generic model instances, given a generic meta-mode, a meta-model for the target system, and a set of model transformation rules. Second, it also ensures that inevitable changes to a system can then be reflected in the generic meta-model and then transferred to all derived models, thus ensuring consistency across all application models.

We assume that system models are constructed using a component-based framework [9,11]. Taking advantage of this framework, we propose a component-based model transformation process.

Our contributions are as follows:

1. We propose a model-generation process that creates simpler and/or abstracted instances of a target meta-model from an instance of a generic meta-model.
2. We adopt a component-based transformation process, such that we transform sub-models on a component-wise basis, and then compose the system-level model by sub-model composition.
3. We demonstrate our model transformation process using a hybrid systems generic model, with our target model being a propositional-logic diagnosis model.

II. COMPOSITIONAL MODELLING PROCESS

A. *Compositional Model Framework*

Component-based modelling is a key part of modern engineering design, as it embodies the principles of modularity, regularity and hierarchy, which enable cost-effective and reliable design [10]. Further, the regularity of component-based methodology translates into reduced design, fabrication and operation costs.

In creating models from a component library within a Model-Based framework [9,11], we call a well-defined model fragment a *component*. We assume that each component can operate in a set of behaviour-modes, where a mode M denotes the state in which the component is operating. For example, a pump component can take on modes nominal, high-output, blocked, and cavitating.

We define two classes of components: primitive and composite. A *primitive component* is the simplest model fragment to be defined. For such a component we specify the inputs I, outputs O, and functional transformation φ, such that we have $O = \varphi(I)$. In this article we will assume that we specify our systems as hybrid systems [12], in which case our primitive component transformation functions will have hybrid systems semantics. Fig. 1 shows two examples of components; for example, the controller component has one input (C_{in}) and one output (C_{out}), with $\psi_C(C_{in}) = C_{out}$.

A *composite component* consists of a collection of primitive components which are merged according to a set ξ of composition rules [6]. In this article we assume standard composition rules; specifying the semantics of composition is beyond the scope of this article, and we refer the reader to [11] for details.

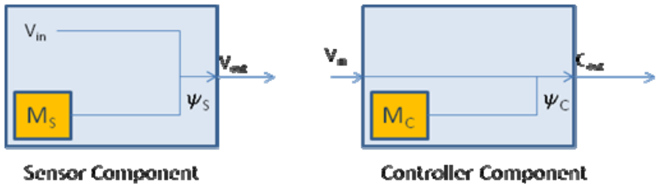

Fig. 1. Examples of diagnosis components for Sensor and Controller. For each component with a failure mode, it is indicated using Mi.

A set of (primitive/composite) components defines a *component library*. In this article we assume a component library consisting of sensors, actuators, human-agent models, and building components such as lights, windows, rooms, etc.

A model consists of the pair $(\mathcal{C},\mathcal{G})$, where $\mathcal{C}$ is a component set and $\mathcal{G}$ is a graph denoting the topology of component connections. We assume that our models are causal, so $\mathcal{G}$ has directed edges.

Each component has a set of properties that depends on the model class, e.g., discrete-event model, hybrid model, etc. We represent an abstract component using the tuple $\{\mathcal{V},\mathcal{P},\Psi,\pi\}$, where

- $\mathcal{V}$ is the set of system variables;
- $\mathcal{P}$ is the set of input/output ports for each component;
- Ψ is the equation set for the component; and
- π is a probability distribution over the equations and/or variables.

We can further specify the system variables as follows. For each variable $V \in \mathcal{V}$ we have a domain for V, denoted D_V. A subset of variables, $\mathcal{M} \subseteq \mathcal{V}$, is denoted as the set of mode variables, with domain D_M.

B. *Model Classes*

In this article we consider transforming a generic model, which we define as a hybrid systems model, into a propositional logic diagnosis model. The hybrid systems model Φ can be used to simulate the behavior, both discrete and continuous, of the system, as well as for verification and other purposes. The diagnosis model Φ_D is used only when some anomalous behavior has been detected in Φ, and its sole purpose is to isolate the mode that the system Φ is operating in. Hence it is a simpler model, and the diagnostics inference is simpler than analogous inference in Φ. In most real-world systems, the diagnostics inference is implemented using some formalism different than the simulation mechanism, such as rules or some model-based framework.

In the following, we outline the models we use for specifying the hybrid systems and diagnosis frameworks.

Hybrid Systems Model

Definition 1 An autonomous hybrid automaton *HS* is a tuple *(M,X, Init, Inv, t,f)*, where *M* is a finite set of mode variables interpreted over finite domains, **M** denotes the finite set of all valuations of the variables *M* over the respective finite domains, *X* is a finite set of variables interpreted over the reals $\mathbb{R}$, $\mathbf{X} = \mathbb{R}^x$ is the set of all valuations of the variables *X*, $Init \subseteq \mathbf{M} \times \mathbf{X}$ is a set of initial states, $Inv : \mathbf{M} \rightarrow 2^x$ assigns to each discrete state $M \in \mathbf{M}$ an invariant set, $t \subset \mathbf{M} \times \mathbf{X} \times \mathbf{M} \times \mathbf{X}$ is a set of (guarded) discrete transitions, $f : \mathbf{M} \rightarrow (\mathbf{X} \rightarrow T\, \mathbf{X})$ is a mapping from the discrete states to vector fields that specify the continuous flow in that discrete state.

Propositional Diagnosis Model

Definition 2 A discrete diagnosis model is specified by a tuple $\{\mathcal{V}, \mathcal{M}_D, O, \Psi, \pi\}$, where

- $\mathcal{V}$ is a set of discrete-valued variables;
- $\mathcal{M}_D \subseteq \mathcal{V}$, is the set of failure mode variables;
- $O \subseteq \mathcal{V}$ is the set of observable variables;
- Ψ consists of propositional equations; and
- π is a discrete probability distribution over the equations and/or variables.

Note that there are several differences between the hybrid-systems and propositional diagnosis models. Whereas the hybrid-systems model has both continuous and discrete variables, the diagnosis model has only discrete. In addition, the diagnosis model requires the specification of failure-mode and observable variables, which the hybrid-systems model does not.

C. *Example: Components for Diagnosis and Hybrid-System Formalisms*

In this section, we give examples of our modeling framework for two simple components, a sensor and a setpoint-based controller.

In the diagnostics framework, a sensor is a device with one input, one output, and 2 modes, {*OK*, *bad*}. If the discrete value of the variable being measured is V_{in}, and sensor is OK, then the sensor reading must equal V_{in}, and the output is V_{out}. We adopt a weak-fault model framework for diagnostic models [13], i.e., we do not specify the faulty behavior, when the mode M_S=*bad*, but just the normal behavior. This approach simplifies the modeling process, and has relatively simple associated diagnosis inference [13]; the drawback is that fault isolation may be less precise compared to models in which failure-mode behaviours are specified. The generic sensor equations ψ_S for a weak fault model are thus:

$$(M_S = ok) \Leftrightarrow (V_{in} = V_{out}).$$

A simple setpoint-based controller will always try to maintain a setpoint μ for the signal, which in Fig. 1 is V_{in}. The generic weak-fault controller equations for such a controller, ψ_C, are given by:

$$(M_C = ok) \Leftrightarrow [(V_{in} \geq \mu) \Rightarrow (C_{out} = off)]$$
$$(M_C = ok) \Leftrightarrow [(V_{in} < \mu) \Rightarrow (C_{out} = on)].$$

In contrast to the diagnosis model for a sensor, in which the measured variable V_{in} takes on a finite set of discrete values, in a hybrid-systems framework V_{in} can take on continuous values governed by a differential equation. In addition, hybrid-systems simulation models typically do not restrict themselves to modes covering the health of a sensor, but will specify modes for the behavior of V_{in}. For example, the equations may take the form:

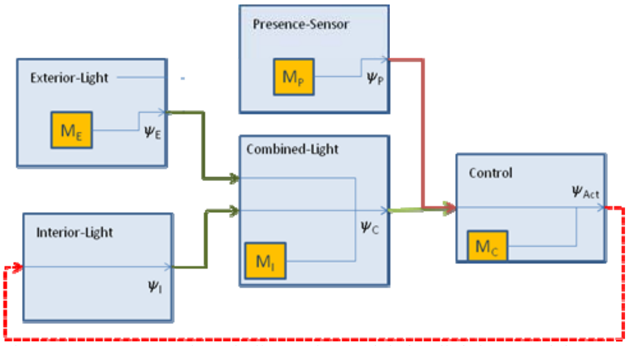

Fig. 2. Component-based diagnosis model for lighting model. For each component with a failure mode, it is indicated using Mi.

$$13 \geq t \geq 6 \qquad \partial V_{in}/\partial t = 50 + V_0;$$
$$18 \geq t \geq 13 \qquad \partial V_{in}/\partial t = 800 - V_{in};$$
$$24 \geq t \geq 18 \qquad \partial V_{in}/\partial t = 0.$$

D. Example: Lighting System

This section now extends the components to create a simple lighting system model. Fig. 2 shows a component-based model for a lighting system. Here, we aim to maintain a setpoint light-level inside a room if there is anyone present in the room, by switching on an internal light when necessary to complement the external light coming through a window. We assume that we can measure both the internal light and external light, to define the *Combined-light*. Hence, we have three components that we integrate for estimating the light level, *Interior-light*, *Exterior-light*, and *Combined-light*.

In this model, a controller must take the value of *Combined-Light*, in addition to the output of *Presence-Sensor*, P_{out}, to determine whether to switch on the *Interior-light*. Hence we have three components for estimating light level (*Combined-Light*), complemented by the *Presence-Sensor* and the *Control* component that controls turning *Interior-light* on and off.

Note that Exterior-light and Presence-Sensor are examples of the sensor component described previously. The Control component is a setpoint-based controller that also incorporates the presence of a person in a room.

In a diagnosis model, the controller equations, ψ_C, are given by:

$$(M_C = ok) \Leftrightarrow [(V_{in} \geq \mu) \wedge (P_{out} = t) \Rightarrow (S_{out} = off)]$$
$$(M_C = ok) \Leftrightarrow [(V_{in} < \mu) \wedge (P_{out} = t) \Rightarrow (S_{out} = on)].$$

It is relatively straightforward to define transformations for the sensor and setpoint-based controller components, However, it is much more complicated to define other components, such as the *Combined-light* component. In these other cases, one needs to define specific transformation rules, on an instance-specific basis. To create a diagnosis model for the *Combined-light* component, for example, we must define a qualitative algebra to specify the Combined-Light equation. Given the possible qualitative values of *Exterior-light* and *Combined-light* are {*low, optimal, high*}, and of *Interior-Light* are {*on, off*}, a truth-table for this qualitative relation is given in Table 1.

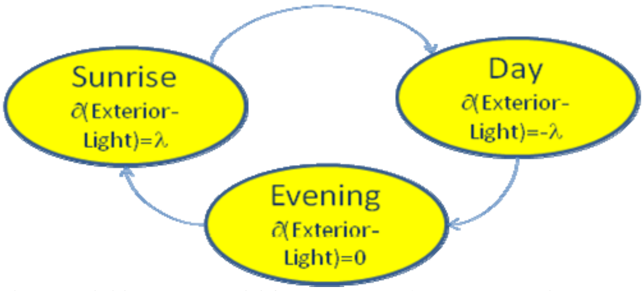

Fig. 3. Hybrid-Systems model for *Exterior-Light* component. The automaton has 3 modes, *Sunrise*, *Day* and *Evening*, with differential-equation dynamics governing the state-variable for the mode, based on a parameter for light factor,λ. We omit the transitions and guards to ensure comprehensibility of the figure.

Exterior-light	*Interior-Light*	*Combined-light*
Low	Off	*Low*
Low	On	*Optimal*
Optimal	Off	*Optimal*
Optimal	on	*High*
High	Off	*High*
high	on	*high*

Table 1. Qualitative truth table for determining value of *Combined-Light* from *Exterior-Light* and *Interior-Light*.

The hybrid-systems model for the *Exterior-Light* function is quite different from that of the diagnosis model. Here, we have an automaton with 3 modes, *Sunrise*, *Day* and *Evening*, where the behavior in each mode is governed by a differential equation based on a light factor λ. Note that this model has a different set of modes to the diagnosis model for *Exterior-Light*, which is given by:

$$(M_E = ok) \Leftrightarrow (\textit{Exterior-Light} = V_{out}).$$

In other words, the diagnosis model makes explicit only the distinction between the functional states of the sensor, and does not represent any nominal modes of the light level. A similar difference holds between all the hybrid-systems and diagnosis component models.

III. MODEL TRANSFORMATION

A. Model Transformation Underpinnings

We assume that we perform component-wise transformation, and then assemble to resulting transformed model Φ^T from the transformed components. In other words, given input components C_i and C_j, we can create the original model as $\Phi = C_i \otimes C_j$. We can then compose the transformed components, $\gamma(C_i)$ and $\gamma(C_j)$, to create the transformed model, using the model composition operator ⊕ for the transformed system. In this case, we have $\Phi^T = \gamma(\Phi) = \gamma(C_i) \oplus \gamma(C_j)$.

There are two entities that we must transform give a model $\Phi=(\mathcal{C},\mathcal{G})$, the components $\mathcal{C}$ and the component connections, defined by $\mathcal{G}$.

It is beyond the scope of this article to prove such properties, and we refer the reader to [3] for details of such formal properties. In this article, we assume that we have an abstraction transformation, in that the diagnosis model is an abstraction of the generic model. We adopt the following

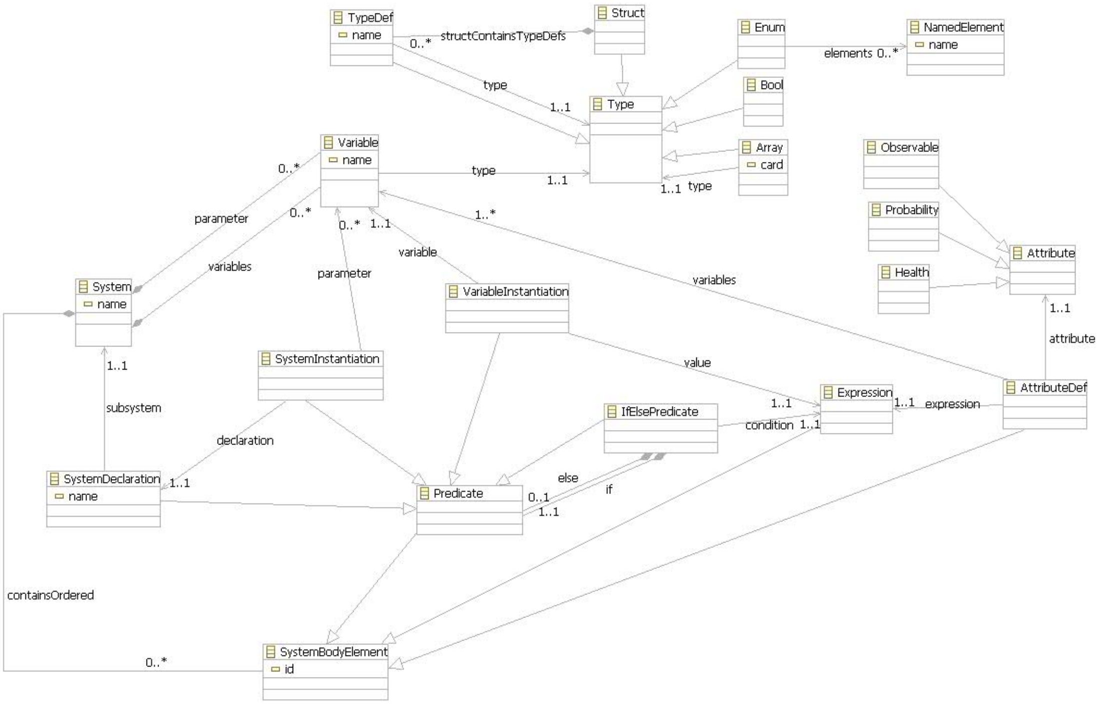

Fig. 5. Meta-model for propositional diagnosis model (Lydia).

definition of a discrete (finite-state) abstraction of a hybrid system from [3].

Definition 3 Let $\Phi = (M, X, Init, Inv, t, f)$ be a hybrid automata and $\Phi^T = (M', Init', t')$ be a discrete transition system. We say Φ^T is an *abstraction* for Φ if there exists a mapping $\gamma: \mathbf{M} \times \mathbf{X} \rightarrow \mathbf{M}'$ such that

(a) If $(\mathbf{M}, \mathbf{x}) \in Init$, then $\gamma(\mathbf{M}, \mathbf{x}) \in Init'$, and

(b) If $((\mathbf{M}, \mathbf{x}), (\mathbf{M}', \mathbf{x}')) \in t_\gamma$ is a transition in the discrete transition system Φ_α corresponding to Φ with respect to γ, then there exists a transition $(\gamma(\mathbf{M}, \mathbf{x}), \gamma(\mathbf{M}', \mathbf{x}')) \in t'$ in DS.

Hence, as noted in [3], the abstraction Φ^T of a hybrid automaton Φ is a discrete transition system that *simulates* the discrete system Φ_γ associated with Φ, where γ defines the corresponding *simulation relation* [7]. Consequently, if a temporal logic formula (e.g., in *ACTL**) is true in the model Φ^T, then it is also true in Φ_γ [6].

B. Transformation Process

We use the theory of model transformation [5] to formalize our transformation process in terms of a rewrite procedure. Model transformations that translate a source model into an output model can be expressed in the form of rewriting rules. Such rules can be classified according to a number of categories [1]. According to [1], the transformation we adopt is an *exogenous transformation*, in that the source and target model are expressed in a different languages, i.e., hybrid-systems and propositional logic languages.

In order to use this approach to model transformation, we need to represent our models in terms of their corresponding meta-models. A *meta-model* is an abstraction of the model that highlights the properties of the model itself. We adopt the OMG standard for the meta-model, called Meta-Object Facility, or MOF. We use the MOF-based tool, ATL [2], to implement our model transformation process. For the generic model, we use the Charon language [15], and for the diagnostics models we use the Lydia language [16].

Figures 4 and 5 depict the meta-models for the hybrid-systems and diagnosis models, respectively. Fig. 6 depicts the meta-model transformation process.

We adopt the definitions of [4] for meta-model mapping and instance.

Definition 4: A meta-model mapping is a triple $\Omega = (S_1, S_2, \Sigma)$ where S_1 is the source meta-model, S_2 is the target meta-model and Σ, called the mapping expression, is a set of constraints over S_1 and S_2 that define how to map from S_1 to S_2.

Definition 5: An *instance* of mapping Ω is a pair $\langle \check{\mathcal{S}}_1, \check{\mathcal{S}}_2 \rangle$ such that $\check{\mathcal{S}}_1$ is a model that is an instance of S_1, $\check{\mathcal{S}}_2$ is a model that is an instance of S_2 and the pair $\langle \check{\mathcal{S}}_1, \check{\mathcal{S}}_2 \rangle$ satisfies all the constraints Σ.

For example, given our general meta-model, a sensor component must satisfy the property of being an instance meta-model.

S_2 is a translation of S_1 if the pair $\langle S_1, S_2\rangle$ satisfies Definition 3 [3]. Hence, we must specify an appropriate mapping, or set of rules Σ, to ensure that this holds. In the following, we will describe component-based rules.

C. *Component-Based Meta-model Transformations*

In this paper, we assume a component-based framework for meta-model mapping, in which we map component by component. In other words, we assume that we can represent a model in terms of a connected set of components, where we call our set of components a component library C. Further, we assume that for each component $C_i \in C$, we have an associated meta-model. Given that we are mapping from component library C_1 to C_2, we have two component libraries, with corresponding meta-model libraries, $S_1 = \{s_{1,1}, \ldots, s_{1,n}\}$ and $S_2 = \{s_{2,1}, \ldots, s_{2,m}\}$. In an analogous fashion to the system-level requirements for the model transformation, at the component level the following must hold: $s_{2,i} \in S_2$ is a translation of $s_{1,i} \in S_1$ if the pair $\langle s_{1,i}, s_{2,i}\rangle$ satisfies Definition 3. This means that there is a relation $\rho \subseteq S_1 \times S_2$ that maps meta-model components of S_1 to equivalent components in S_2.

In addition to these abstract requirements, there are two variable-types that must be defined to map to a diagnosis model: the failure-mode and observable variables, M_D and O respectively. In most model-based diagnosis applications, it is assumed that (1) every component has an associated failure-mode, and that (2) the observables O are restricted to sensor- and actuator-outputs. Using our component-based approach, property (1) can easily be ensured for weak-fault models

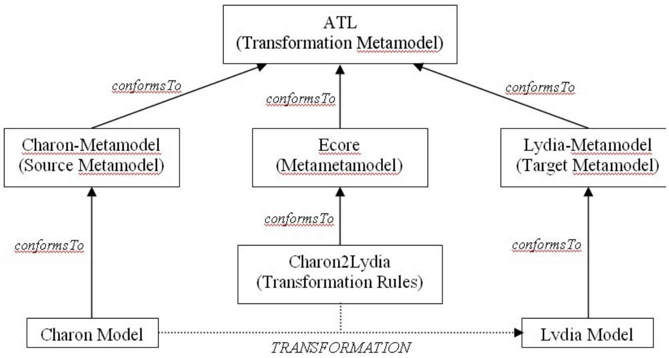

Fig. 6. Meta-model transformation process. At the bottom left, we have the generic (hybrid-systems) model in the Charon language, and at the bottom right we have the diagnostics model in the Lydia language. Each model must conform to its particular meta-model, and then to the ATL transformation meta-model.

D. *Verification Properties*

One important property of the model transformation is that it enables you to verify that the appropriate properties of the generated model hold.

The correctness of a model transformation depends on several factors, covering issues such as computational properties (e.g., whether the transformation terminates), syntactic properties (e.g., whether the output model complies with the syntactical rules of the output language), and correctness properties (e.g., whether output model achieved the intended result of mapping the semantics of the input model into that of the output model). In this article, the output diagnosis model is *simulation-correct* if it reproduces the behaviour of the original hybrid-systems model where model behaviours correspond. The mapping of Definition 3 ensures simulation-correctness [3].

A second correctness property could be termed *diagnosis-correct*. In this article we have specified a simple form of the diagnosis model, called a weak fault model, in which we specify only normal behaviour using a binary-valued failure mode [13]. This approach guarantees a diagnosis-correct output model.

Theorem 1: A meta-model mapping $\Omega = (S_1, S_2, \Sigma)$, where S_1 is a hybrid-systems meta-model, S_2 is a weak-fault propositional diagnostics meta-model and Σ is a set of *simulation-correct* mapping constraints over S_1 and S_2, is diagnosis correct.

If we want to specify detailed failure mode behaviours, as in strong fault models [14], then additional mapping constraints are necessary to provide guarantees of diagnosis-correctness.

A final form of correctness is topology-correctness. In this article we have assumed that the hybrid-systems and diagnosis models will have the same topology, and all that is transformed is the component specifications. In general, this assumption may not hold. When it does hold, work has been done to guarantee that the transformation matches certain graph structures in the input graph and creates certain structures in the output graph [8].

IV. EXAMPLE

In this section we illustrate through a simple lighting system the transformation steps to generate Lydia programs from Charon models. As explained earlier the system consists of a simple lighting controller that tracks the presence and defines the internal light level according to the external light intensity.

The Charon model is composed by two agents: agent Presence that deals with the presence, it implements the mode that switches the light according to presence and internal light sensor, agent Light defines the mode that controls the combined light.

The transformation engine parses the Charon model following the hierarchical structure in accordance to the transformation rules. The code shown in Fig. 7 illustrates the rules that apply to transform agent hierarchy into the corresponding system/sub-systems Lydia model.

At the bottom level, the transformation engine translates the Charon transitions into Lydia if/else structure following the rules described in Fig 8. The basic idea is to track to condition for the observable variables. This can be done by looking at the action and its related conditions. Transformation rules had to be written to consider the case where same actions are executed under different conditions.

```
module Charon2Lydia;
create Lydia : LydiaMM from Charon : CharonMM;
…
rule Agent2System {
   from s : CharonMM!Agent (s.SubAgent.size()=0)
   to t : LydiaMM!System (name <- s.name)}

rule AgentWithSubAgent2System {
   from s : CharonMM!Agent (s.SubAgent.size()>0)
   to t1 : LydiaMM!System (
           name <- s.name, containsOrdered <- t2,
            containsOrdered <- t3 ),
      t2 : distinct LydiaMM!SystemDeclaration
foreach (e2 in s.SubAgent)(
         id <- e2.id, name <- e2.name,
          subsystem <- e2.defid),
      t3 : distinct LydiaMM!SystemInstantiation
foreach (e3 in s.SubAgent)(
       id <- e3.id,
       declaration <- t2 )
}
```

Fig. 7. Code Generation Rules for Agent Hierarchy

```
rule TransitionalMode2IfElsePredicate {
   from s : CharonMM!Mode
   using
     seq : s.Transition.getTransSeqSameActions()
   to t : distinct IfElsePredicate
         foreach (subseq in seq) (
               id <- subseq -> first().id,
                condition <- subseq ->
               first().Action,
               if <- subseq.getGuardCombining())
}

helper context Sequence{CharonMM!Action} def: …
getTransSeqSameActions() :
helper context String def: getGuardCombining() :
```

Fig. 8. Code Generation Rules for mode transitions

Fig. 9 shows the resulting Lydia code generated from the Light agent for a single zone model as described in [17].

```
  system Light (
   // inputs: Light externalLight,
             Switch internalLight,
  // outputs:  Light lightSensor)
  {bool h;                          // health
   Light combinedLight;
   if (combinedLight = Light.low)
   {(externalLight=Light.low) and (internalLight
     = Switch.off);}
   if (combinedLight = Light.high)
   {((externalLight=Light.high)and(internalLight
    = Switch.off)) or ((externalLight=
    Light.high) and (internalLight=Switch.on))or
   ((externalLight=Light.high)and (internalLight
    = Switch.off));}
  … …
```

Fig. 9. Lydia code for the Light System

The current translation rules consider only the control agents; therefore the Lydia code for the environment agents has to be provided by the diagnosis expert. He has to provide as well the health definition as described in Fig. 10.

```
if (h) {(lightSensor = combinedLight );}

attribute health (h) = (h = true) ? true : false;
attribute probability (h) = h ? 0.99 : 0.01;
```

Fig. 10. Code Generation Rules for mode transitions

V. SUMMARY AND CONCLUSIONS

We have described a framework for automatically transforming a generic model into an abstracted model. We have described how we can define a generic model using the hybrid-systems language, and then transform this to a discrete propositional diagnosis language. Further, we have shown how compositional model representation can be adopted for such model transformation. We have provided an example of a lighting system for building automation to demonstrate this procedure.

This approach can make significant contributions to building automation systems. Instead of needing to create multiple models, and maintain consistency among multiple models, this transformation approach provides the methodology for creating a single generic model, and then creating component-based meta-models and transformation rules to automate the generation of the additional models needed for building automation applications.

ACKNOWLEDGMENT

This work was funded by SFI grant 06-SRC-I1091.

REFERENCES

[1] G T. Mens, P. Van Gorp. A taxonomy of model transformation. In Proc. *Int'l Workshop on Graph and Model Transformation*. 2005.

[2] http://www.eclipse.org/m2m/atl/

[3] Ashish Tiwari, Abstractions for hybrid systems, *Formal Methods in System Design*, 32(1): 57-83, 2008.

[4] R. Fagin, P.G. Kolaitis, L. Popa, W.C. Tan, Composing schema mappings: second-order dependencies to the rescue, *ACM Transactions on Database Systems* 30 (4) (2005) 994–1055.

[5] Schmidt, D.C. "Model-Driven Engineering". *IEEE Computer* 39 (2): February 2006, http://www.cs.wustl.edu/~schmidt/PDF/GEI.pdf.

[6] Grumberg O, Long DE (1994) Model checking and modular verification. *ACM Trans Program Lang Syst* 16(3):843–871.

[7] Milner R (1971) An algebraic definition of simulation between programs. In: *Proc. 2nd IJCAI*, pp 481–489.

[8] Ruth Raventós and Antoni Olivé, An object-oriented operation-based approach to translation between MOF metaschemas. *Data & Knowledge Engineering* 67, 444–462: 2008.

[9] Gregor Gössler and Joseph Sifakis. "Composition for Component-Based Modeling", in *Formal Methods for Components and Objects*, Springer Lecture Notes in Computer Science, 443-466, 2003.

[10] Suh, N.P., "The Principles of Design," *Oxford University Press*, USA, 1990.

[11] Keppens, J. and Shen, Q. "On compositional modeling," in *The Knowledge Engineering Review*, 16(2), 157--200, 2001.

[12] Labinaz, G. and Bayoumi, M.M. and Rudie, K., "Modeling and Control of Hybrid Systems: A Survey," *Proc. of the 13th Triennal World Congress*, San Francisco, USA, 1996.

[13] de Kleer, J., Mackworth, A., & Reiter, R. "Characterizing diagnoses and systems". *Artificial Intelligence*, 56(2-3), 197–222, 1992.

[14] Struss, P., & Dressler, O. "Physical negation" - integrating fault models into the General Diagnostic Engine". In *Readings in Model-Based Diagnosis*, pp. 153–158. Morgan Kaufmann Publishers Inc., 1992.

[15] http://rtg.cis.upenn.edu/mobies/Charon/

[16] http://www.st.ewi.tudelft.nl/~gemund/Lydia/

[17] A. Mady, M. Boubekeur and G. Provan, "Integrated Simulation Platform for Optimized Building Operations," IACSIT SC 2009.