Model-Driven Diagnostics Generation for Industrial Automation

M. Behrens, G. Provan, M. Boubekeur, A. Mady,
CCSL, Computer Science Department,
University College Cork,
Cork, Ireland,
Email: {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 diagnostics from a more complex simulation model. We show how a model-transformation framework can transform a hybrid-systems simulation model into a propositional-logic diagnostics model with appropriate transformation rules. We illustrate our approach with an example from the domain of control for building lighting systems.**

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 an automated model generation approach. This model-generation approach uses abstract descriptions of properties of models, termed meta-models, in order to transform an initial model, called a source model, into a target model. In this article we illustrate the transformation of a simulation model into a qualitative diagnosis model. We assume that we can specify a *generic meta-model*, which is a detailed model that identifies a global 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 a discrete diagnostics model from a hybrid-systems simulation/control 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-model, 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 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 [1], [2]. 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. System Architecture

A. Architecture

Figure 1 depicts the main features of our system architecture. This figure shows that we have two applications, a source and a target application, with a corresponding meta-model for each application. We use the meta-models, together with transformation rules, to convert the source to the target model.

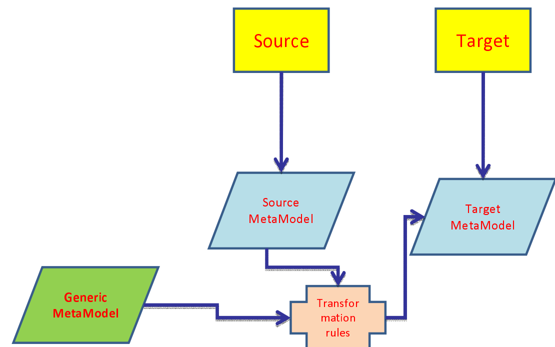

Fig. 1. Schematic of System Architecture

B. Compositional Modeling

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 [3]. 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 [1], [2], 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 an artificial light component can take on modes on, off and burnt-out.

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 work we specify our systems as hybrid systems [4], in which case our primitive component transformation functions will have hybrid systems semantics.

A *composite component* consists of a collection of primitive components which are merged according to a set ξ of composition rules [5]. 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 [2] for details.

A set of (primitive/composite) components defines a *component library*. In our case a component library consists of sensors, actuators, human-agent models, and building components such as lights, windows, rooms, etc.

C. Source and Target Applications

In this article we consider transforming a generic model, which we define as a hybrid systems model[4], 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 present an overview of our source and target applications.

1) Charon: The tool that we have adopted for this modeling is Charon [6]. Charon is a hybrid modeling language that supports the construction of hierarchical, component-based hybrid systems models. Each primitive system component is called an agent; Concurrency of agents means that agents can communicate with other agents asynchronously. Charon describes behavior in an agent using the formalism of a mode, which is a hierarchical hybrid state machine that can have submodes and transitions connecting them.

Charon runs simulations by alternating a series of discrete and a continuous steps for each agent. A discrete step consists of a series of transitions from the active atomic mode to another atomic mode. This flow of control is determined through the assignment of mode variables through mode invariants, transition guards, or transition actions. We assume that the continuous variables evolve according to algebraic and differential constraints governed by the active modes, and this continuous evolution may also trigger discrete mode changes.

For creating the hybrid system model we use Visual Charon, the graphical editor provided through the Charon project [6]. From the graphical model, Visual Charon generates a file in the Charon language which can be used for simulation and code generation to embedded java as shown in [7].

Figure 2 is a screenshot of the Visual Charon application showing the same system with the behaviour of the blinding selected and displayed in the editor. The tool also features dialog menues for entering the guard (condition) and actions for tranformations, for defining the variables, parameters and constraints.

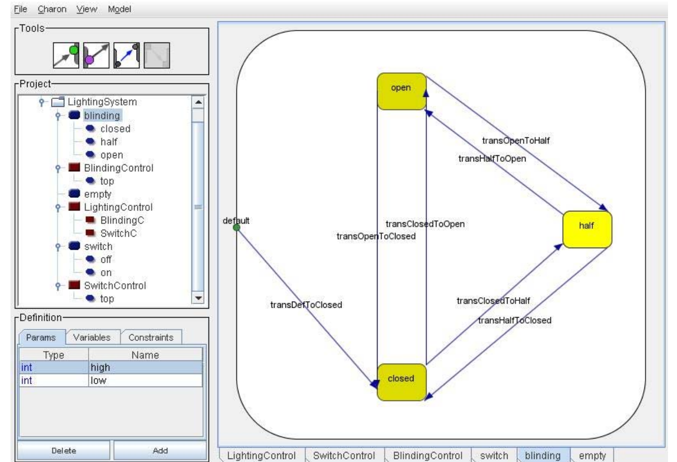

Fig. 2. Creating the source model with Visual Charon

2) Lydia: We intend to transform the hybrid systems simulation models into simpler diagnostics models, which describe only discrete-valued variables, and abstract the hybrid systems models. The tool that we have adopted for diagnostics modeling is Lydia (Language for sYstem DIAgnosis) [8]. Lydia is a language for modeling of complex systems so that their behavior can be analyzed for fault diagnosis. A Lydia model is compiled to fault diagnostic software for observing the according system. If a failure is observed the diagnostic software is able to isolate the system that caused the fault.

A Lydia model consists of systems which may contain instances of previously defined systems to form a modeling hierarchy. Lydia allows only boolean types, enumerations and structure types as variables. Attributes can be assigned to variables. The variable specifying the `health` attribute, the *system health variable*, denotes the health state of the system. The `probability` attribute appends the presumed probability of a variable for a certain value, e.g. the *system health variable*, if boolean type, to be `true`. Figure 3 illustrates the structure of a Lydia diagnosis system. The system modelled here is a *weak model* [9], since it defines only the healthy behaviour of the systems. In contrast, *strong models* also describe the

faulty behaviour, usually with the `if {...} else {...}` construct. The `observable` attribute is attached to variables which can be observed/measured in the real system and which act as input parameters to the diagnostic software.

```
system subSys ( bool i, j ) {

  bool h; // system health variable

  if (h) {
      i && !j; // health condition
  }

  attribute health (h) = h ? true : false;
  attribute probability (h) = h ? 0.99 : 0.01;
}
system mainSystem ( bool i, j1, j2 ) {

  // declare components
  system subSys sub1, sub2;

  // connect components
  sub1 (i, j1);
  sub2 (i, j2);

  attribute observable (i, j1, j2) = true;
}
```

Fig. 3. Lydia system model for a simple light (switch) controller

To start diagnostic inference, the modelled system must be fed with an observation of the real system. Each observation contains a conjunction of assignments for the observable variables of the Lydia system model.

3) Extended Charon Metamodel: we deduced the basic version of a meta-model for Charon models from the Visual Charon DTD, which defines the structure of the XML document to describe the Charon model, by dropping the graphical information.

Charon models do not typically encode any diagnostic information, so we had to add extra modes to encode faulty behavior. We extended the basic Charon Metamodel with several components which allow the diagnosis expert to add information needed for the transformation to a diagnosis system. We omit presenting the UML diagrams of the meta-models, but details can be found in [10].

III. Model Transformation Process

We use the theory of model transformation [11] 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 [12]. According to [12], 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 openArchtectureWare (oAW) [13] framework, to implement our model transformation process. For the generic model, we use the Charon language [6], and for the diagnostics models we use the Lydia language [8]. We adopt the definitions of [14] for meta-model mapping and instance.

Definition 3.1: 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 3.2: An *instance* of mapping $\otimes$ is a pair $\langle \check{S}_1, \check{S}_2 \rangle$ such that $\check{S}_1$ is a model that is an instance of S_1, $\check{S}_2$ is a model that is an instance of S_2 and the pair $\langle \check{S}_1, \check{S}_2 \rangle$ 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 3.1. 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.

A. Model Transformation Overview

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 $\mathcal{C}$. Further, we assume that for each component $C_i \in \mathcal{C}$, we have an associated meta-model. Given that we are mapping from component library $\mathcal{C}_1$ to $\mathcal{C}_2$, we have two component libraries, with corresponding meta-model libraries, $S_1 = \{s_{1,1}, \ldots, s_{1,n}\}$ and $S_2 = \{s_{2,1}, \ldots, s_{2,m}\}$.

Given a component-wise transformation, we must 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 $\oplus$ for the transformed system. In this case, we have $\Phi^T = \gamma(\Phi) = \gamma(C_i) \oplus (C_j)$.

B. Transformation Rules

To perform the transformation step, some transformation rules had to be defined and implemented. These transformations basically consider all the constructors and statments provided by Charon, and define how to generate the corresponding Lydia programs. In the following we show a few of these rules and advise the reader to consult [10] for more details.

1) Transformation Rule for Type Definitions: The diagnosis expert, after analysing the usage of all variables in the source model, defines a mapping to boolean and multivalued types.

While in the source application only boolean, integer and real types are supported, our extended source-metamodel provides features to declare a mapping to boolean types, enumerations and structures.

For the transformation of type definition we provide the following guidelines:

- Charon Boolean types can simply be mapped to Lydia boolean types.
- Integer and real types can be mapped to boolean types if they appear in only one simple condition (e.g., `a > 50`).
- Integer and real types can be mapped to enumerations if they appear in condition, which are mutually exclusive over the whole range (e.g., `a < 50`, `a >= 50 && a <= 100` and `a > 100` with `low`).
- Integer and real types must be mapped to structures (of booleans) if they appear in conditions, which are not mutually exclusive (e.g., `a > 50` and `a < 100`).

After making the type definitions for all variables in the source model, the diagnosis expert must also translate the conditional code with respect of the variables usage context. For example, in the lighting system example in section IV, the condition of the exterior light being below the constant value `low` (`Sext < low`) has to be translated to a condition of the exterior light conforming to the value `low` of the enumeration `lightlevel` (`Sext = lightlevel.low`).

2) Transformation Rule for System/Sub-system Hierarchy: The core of the Charon source model consists of Agents which contain either a set of concurrent Subagents or one Submode that models the Agent's behaviour. The transformation engine parses the source model following the hierarchical structure. The `Agent2System` rule 4 has been defined to transform Agent hierarchy into the corresponding system/sub-systems of the diagnosis model.

```
create LYDIA::System Agent2System (CHARON::Agent a) :
   setName(a.name) ->
   setSubsystem(a.SubAgent.SubAg2SubSy());
create LYDIA::SubSystem SubAg2SubSy (CHARON::SubAgent s) :
   setSystem(s.defid.createLydiaSystem());
```

Fig. 4. Transformation rule Agent2System and SubAgent2SubSystem

This rule must be called with Charon Agents in reverse hierarchical order to assure that in the target model the association from a Subsystem to its corresponding System can be created.

3) Transformation Rule for Parameters and Variables: In the generic source models the agents and modes communicate via shared variables. These shared variables have to be translated to input parameters for subsystems in the target model. At the same time parameters being passed to agents and modes as constant values can often be omitted. In the diagnosis model these constant values are used for the type definition as described and therefore obtainable throughout the system. The top-level system in the diagnosis model must also contain all variables, that are marked as *observable* in the extended source model, as input parameters.

4) Transformation rule for System Health and Conditions: At the bottom level, the transformation engine translates transitions from the generic source model into conditions for *system health* in the diagnosis model.

Again the diagnosis expert must analyse the system to specify health for each top-level mode in the extended source model.

Boolean health allows the resulting diagnosis application to detect either healthy or unhealthy behaviour. Submodes represent either health or failure states indicated through naming conventions (containing the string 'fail'). Agents containing modes with boolean health are transformed to weak model diagnosis systems. A transformation with an endpoint in a mode that represents a health state is transformed into a condition. If with given observation parameters this condition is `true`, then the system is regarded as healthy. For our simple lighting model we specified boolean health for the top-level modes and all submodes represent healthy states.

For modes representing failure states the resulting condition is simply inverted (`setCode('not(' + ... + ')')`).

To create a strong diagnosis system the diagnosis expert must specify *system specific health* for the top-level mode. With system specific health multivalued health can be specified in the extended source model (e.g., the health state of a lamp with two light bulbs might be `nominal`, `bulb1_broken`, `bulb2_broken` or `both_broken`). For each health state a *probability* associating at least one transition must be specified. The transformation rules for system specific health create an enumeration containing the names of the health states and probabilities associating conditions and one element of the enumeration.

At the mode level, the transformation engine translates the Charon transitions into Lydia conditions following the rules described in figure 5.

```
create LYDIA::Condition (CHARON::Transition t) :
  setName(t.name) ->
  setCode('(' + t.Guard.diagCode + ')' +
          ' && ' +
          '(' + t.Actions.diagCode + ')' +
          t.getInvariantConditions(this));
```

Fig. 5. Transformation rule Transition2Condition

The attribute `diagCode` has been added to guards and actions of transitions in the source model by the diagnosis expert. It contains a translation of the component's code to the syntax of the target model. In Lydia, the language of the diagnosis model, the same syntax is used for assignments and conditions. Therefore an assignment used in a conditional context is true, if the result of the assignment is true. This fact allows us to simply conjunct the guard and action of a transition to form a health condition for the diagnosis model. Some submodes might contain constraints of type *invariant* which contain a condition that must be true as long as the state of the mode is situated in that submode. These conditions are concatenated to the conjunction of the health condition as well.

Assertions are a feature of the Charon language that can be used to conduct primitive fault detection for agents already in the simulation model. Assertions contain a condition that must be `true` at all times, otherwise the simulation is aborted. Nevertheless assertions do not provide the accuracy to replace a proper diagnostics. If system specific health is used for diagnosis, assertions must be linked with a certain health state. With boolean health the assertion can be inverted and transformed to a failure condition.

IV. Lighting System Example

In this section we illustrate through a simple lighting system the transformation steps to generate Lydia programs from Charon models.

A. Lighting System Objectives

In this model, 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*.

Figure 6 depicts the main features of our model. This figure shows the door through which people can enter and exit the room; presence sensors detect the presence of people in the room. The figure shows that the room has a window with an automated blinding system, through which ambient light enters the room. There are also artificial lights, and light sensors for the ambient and artificial light levels. Finally, a controller governs the actuation of the blinds and artificial lights.

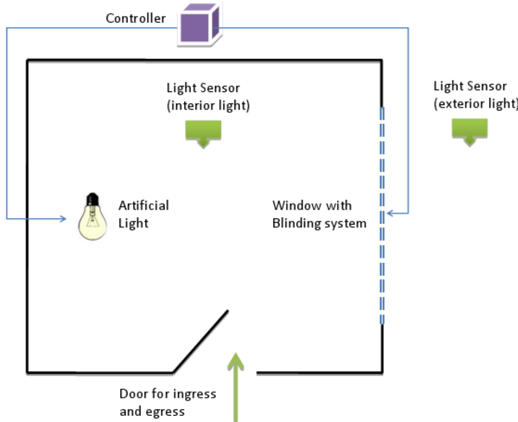

Fig. 6. Schematic of Lighting System

B. Source (Charon) Lighting System Model

The Charon model for our lighting system consists of a system agent with two concurrent control agents: a simple controller for the interior artificial light (`SwitchControl`) and another simple controller for the blinding (`BlindingControl`). Each control agent contains one top-level mode to describe its behaviour. Figure 7 shows the architectural and behavioural hierarchy of the simple lighting system.

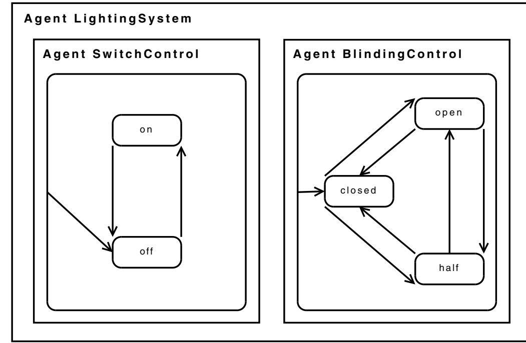

Fig. 7. Charon model of a simple lighting system

The top-level mode of the `SwitchControl` agent contains transitions between two submodes: `on` and `off`. Both modes are empty modes and for our simple example the behaviour could be modelled with only one submode (in which case transitions start and end in the same submode). Figure 8 shows the Charon code for this mode. Each transition contains a guard condition (`when`) and an action (`do`).

```
mode switch(int high, int low) {
  write discrete bool switch;
  read discrete bool Spers;
  read discrete int Sext;

  mode on = empty();
  mode off = empty();

  trans transDefToOff from default to off
    when (true) do {switch = false;}
  trans transOffToOn from off to on
    when (Spres && Sext < low) do {switch = true;}
  trans transOnToOff from on to off
    when (Spres == false || Sext >= low)
      do {switch = false;}
}
```

Fig. 8. Charon code for the top-level mode of the control agent for the artificial interior light

The `SwitchControl` agent encodes the following control rules: (1) if a person is present and the exterior light is low, the artificial interior light will be switched on (transition `transOffToOn`); and (2) the light will be switched off in any other case (transition `transOnToOff`).

The `BlindingControl` agent controls the degree of closing with three submodes (`open`, `half` and `closed`) in an analogous manner. To simplify matters we assume that opening and closing of the blinds is an atomatic operation. Otherwise we would need one additional submode for each transition, in order to model the states of opening and closing the blinds, and twice the number of transitions. If the presence sensor indicates `Spres = false` the blinds will be opened, otherwise the blinding will be opened if the exterior light is optimal, half closed if the exterior light is high and fully closed is the exterior light is low. The corresponding truth table is shown in Figure 9.

S_{pres}	S_{ext}	$artLight$	$blinding$
present	low	on	closed
present	optimal	off	open
present	high	off	half
absent	low	off	open
absent	optimal	off	open
absent	high	off	open

Fig. 9. Qualitative truth table for determining value for artificial light and blinding

C. Extension of the Source Model towards Tranformation

In order to support the diagnosis process on the generated Lydia model some extensions had to be implemented for the Charon model of the lighting system. This mainly concerns the type definitions and certain assumptions as explained below.

1) Type Definitions: Following the guidelines itemised in section III-B1 we specity the following type definitions:

- Boolean types, `Spresence` and `switch`, will not be changed.
- The integer type `Sext` is compared only against the integer constants `low` and `high`. The conditions `Sext < low`, `Sext >= low || Sext <= high` and `Sext > high` are mutually exclusive so that the variable for the sensor value can be transformed to enumerations with three elements, one for each of the conditions.
- The integer type `blinds` is set inside the `BlindingControl` agent the values `0` for open blinds, `50` for half closed blinds or `100` for closed blinds and can therefore be transformed to an enumeration containing three elements.

2) Assertions: We assume that the interior light can also be measured by a sensor and will be provided as input parameter to the diagnosis system. In relation to the behaviour of the `SwitchControl` agent the following assertion can be added: `assert` {`(switch and (Sint > low)) or !switch`} If the artificial interior light is switched on, the interior light `Sint`, that is measured by a sensor, must be above `low`.

In an analogous way we can make the following assertions for the behaviour of the `BlindingControl` agent. (1) if the blinds are open, the interior light is not (significantly) lower than the exterior light; and (2) if the blinds are half closed and the exterior light is high (`Sext > high`), the interior light must be higher than `low`.

D. Tranformation of the Lighting System for Diagnosis

Figure 10 shows an overview of the transformation process from the Charon model of the lighting system example to its corresponding Lydia model.

1) System/Sub-system Hierarchy: Applying the transformation rules introduced in section III-B2 Charon agents are transformed to Lydia systems. First the `SwitchControl` agent is transformed into the Lydia `SwitchControl` system and the `BlindingControl` agent respectively. Second the

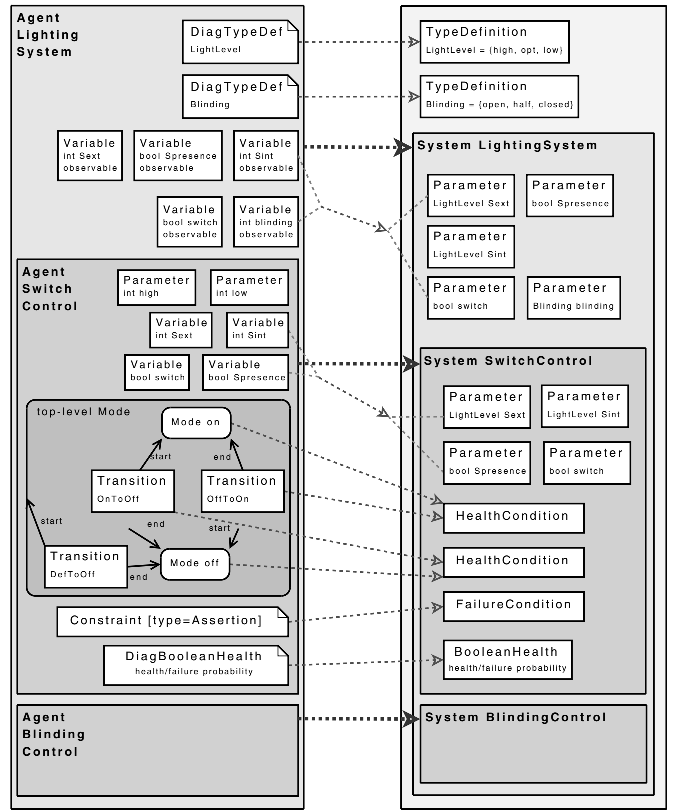

Fig. 10. Tranformation of the Charon Model for the Lighting System

superordinate `LightingControl` agent is transformed into the corresponding Lydia system with the two previously transformed systems as associated subsystems.

2) Parameters and Variables: After the type definitions have been specified by the diagnosis expert, they will be transformed without any changes. The Parameters `high` and `low` can be marked as *redundant*, because they appear just in the context of comparing the values for the light intensity. This context has already been covered by the enumeration type definition `LightLevel` with its elements `high`, `opt` and `low`. Therefore these parameters are not considered for next transformation steps.

All variables in the Charon lighting example model, those providing the values measured by the sensors (`Sext`, `Sint` and `Spresence`) and those representing the state of the devices (`switch` and `blinds`), are *observable* and must be transformed into input parameters for the `LightingControl` system, the main system in the Lydia model. These variables are also shared variables in the Charon model and must be translated to parameter for the subsystems so that they can be passed down in the systems' hierarchy.

3) System Health and Conditions: We assume that for the simple lighting example only *boolean health*, i.e. the information whether the system is in a healthy state or not, is required for the diagnosis software. From the `SwitchControl` agent in the Charon lighting example model two *health conditions*, one for each transition except the default transition `defToOff`, are

created for the Lydia system by applying the transformation rules described in section III-B4. For example the transition `offToOn` with condition `Spresence && Sext<low` and action `switch=true` is tranformed to the health condition `(Spres and Sext=LightLevel.low)and switch`

The assertion constraint which has been specified by the diagnosis expert is transformed into a *failure condition* in which the conditional code is inverted.

E. Observation

From the Lydia model created by processing the model transformation we produced rules to generate Lydia code. The code fragment generated from the `SwitchControl` system is shown in figure 11.

```
system switch (
 bool switch, Spresence,
 lightlevel Sext, Sint )
 {
  bool h; // system health variable
  if (h) {((
       // health condition 1
       Spres && Sext=lightlevel.low && switch
    ) or (
       // health condition 2
       !Spres && Sext!=lightlevel.low && !switch
    )) and (
     // failure condition
     (switch and (Sint!=lightlevel.low)) or !switch
   )}
  attribute health (h) = h ? true : false;
  attribute probability (h) = h ? 0.99 : 0.01;
  attribute observable (switch, Spers, Sext) = true;
}
```

Fig. 11. Lydia system model for a simple light (switch) controller

To show the execution of a diagnosis, we feed the genereted Lydia system with the example observation shown in figure 12. The first observation (`obs_simpleLighting_1`) shows a correct behaviour, the second observation (`obs_simpleLighting_2`) is incorrect: The measured interior light `Sint` is low even though the artificial interior light (`switch`) is turned on. This contradicts the assertions introduced in Section IV-C2.

```
observation obs_simpleLighting_1 {
  Sext=LightLevel.low && Spresence &&
  Sint=LightLevel.opt &&
  switch && blinds=Blinding.closed; }

observation obs_simpleLighting_2 {
  Sext=LightLevel.low && Spresence &&
  Sint=LightLevel.low &&
  switch && blinds=Blinding.closed; }
```

Fig. 12. Lydia observation on the simple light controller.

Figure 13 shows the output from executing the diagnosis with those two observations. For the first observation the diagnosis report `d` is empty, for the second observation the system detects a fault in the `SwitchControl` subsystem. Most probably a lamp is burnt out and should be replaced.

```
lydia> diag simpleLighting obs_simpleLighting_1
@ start output
d = {  }
@ stop output

lydia> diag simpleLighting obs_simpleLighting_2
@ start output
d = { SwitchControl.h = false }
@ stop output
```

Fig. 13. Diagnosis Output

V. Summary and Conclusion

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] Gregor Goessler and Joseph Sifakis, "Composition for component-based modeling," *Formal Methods for Components and Objects, Springer Lecture Notes in Computer Science*.

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

[3] N.P. Suh, *The Principles of Design*.

[4] G. Labinaz, M.M. Bayoumi, and K. Rudie, "Modeling and control of hybrid systems: A survey," *Proc. of the 13th Triennal World Congress, San Francisco, USA*, 1996.

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

[6] "Charon: Modular specification of hybrid systems," Website, http://rtg.cis.upenn.edu/mobies/charon/.

[7] M. Boubekeur A. Mady and G. Provan, "Integrated simulation platform for optimized building operations," IACSIT SC, 2009.

[8] "The lydia project," Website, http://www.st.ewi.tudelft.nl/ gemund/Lydia/.

[9] K.J. De, AK Mackworth, and R. Reiter, "Characterizing Diagnosis and Systems," *Artificial Intelligence*, vol. 56, no. 2-3, pp. 197–222, 1992.

[10] M. Behrens, "Model transformation from simulation to diagnosis," Tech. Rep., 2009.

[11] D.C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39 (2), February 2006, http://www.cs.wustl.edu/ schmidt/PDF/GEI.pdf.

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

[13] "openarchitectureware," Website, http://www.openarchitectureware.org/.

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