A Mixed Approach to Rigorous Development of Control Designs

M. Satpathy[1], S. Ramesh[2], C. Snook[3], N.K. Singh[4] and M. Butler[3]

Abstract— **The control law of a typical industrial system has a modulating (continuous) component and a sequential/modal component. Control engineers are traditionally good at specifying the modulating part of the control laws unambiguously, correctly and completely. Software engineers have similar skills on the sequential component. In this paper, we discuss a mixed approach in which software and control engineers collaborate to develop control designs. The proposed approach is based upon a novel modeling notation called RRM diagrams. A formal refinement method based on RRM diagrams is developed which enables the development of sequential components as well as control designs. We illustrate our method by considering the case study of a simplified Adaptive Cruise Controller (ACC).**

I. INTRODUCTION

A control design specification can be broadly partitioned into two diverse components: the modulating (or feedback) control laws, usually expressed as continuous functions, and the sequential (or discrete logic) component which models the mode behaviour [8], [3]. Control engineers derive the feedback control laws from known plant behaviour and environmental conditions, while software engineering expertise is needed to model accurately the mode behaviour, which could be complex when the input size is large and the plant is complex. The design techniques used by control engineers are typically different to those used by software engineers. Therefore, there is a case for using the expertise of both.

Consider the example of a cruise controller (CC). A control engineer first models the plant, then derives the feedback control law – say, a PID controller – from the plant behaviour with additional inputs like the road condition and the number of passengers in a vehicle. Furthermore, when the vehicle is riding a steep height, the plant is different from the plant when the vehicle is climbing down a hill. For such cases where the plant characteristics could change drastically, depending on the operating regime, multiple feedback control laws, each of which would be switched on at different operating situations, are often employed. Development of a feedback control component that meets right parameters like response time, settling time etc. requires sound control engineering principles.

A CC also requires inputs from the driver and the environment. Depending on all such inputs, it is determined whether a feedback control law would be operational or not. Furthermore, since there are multiple feedback control laws, which law would be invoked in which situations, must be determined accurately. All these tasks become much more complicated when we consider complex control systems like the Lane Centering Controller of a vehicle.

In this paper, we propose a mixed approach, in which both control and software engineers cooperate to develop control designs. This approach uses a notation called RRM (Requirement, Refinement and Modeling) diagrams [5]. A software engineer uses RRM diagrams to develop a frame of the final design containing only the discrete component; the frame has placeholders for continuous components, which are separately developed by control engineers. Our main contributions:

- Development of a novel approach in which software and control engineers collaborate to develop control designs. In a previous research [5], we have developed a RRM diagram based method for developing discrete control designs; in this research, we extend our earlier methodology to develop modulating feedback controllers.
- Integration of the discrete and modulating components developed respectively by software and control engineers to obtain the final controller in SL/SF. The discrete component in SL/SF is auto-generated from a RRM model; the modulating component is manually obtained.

The organization of the paper is as follows. Section II discusses the related work. Next, we present RRM diagrams in greater detail. In Section IV, we model an ACC using RRM diagrams. Section V presents our method of Simulink generation. Next we discuss our implementation, and analyse our approach. Finally, Section VIII concludes the paper.

II. RELATED WORK

Satpathy *et al.* in [5] have introduced RRM diagrams (RRMDs) for developing pure discrete controllers. Abstract RRMDs are gradually refined till they capture all requirements of a discrete controller. Such diagrams have been encoded as UML-B models [7]. The authors have developed a prototype tool to auto-generate SL/SF design models from RRMDs. The proposed work extends RRM diagrams and the refinement methodology to include continuous elements.

The SL/SF modeling notation allows unsafe constructs, which if used could lead to issues like non-termination, stack overflow *etc.* Scaife *et al.* in [6] have presented a safe SL/SF subset which can be translated to the synchronous language *Lustre*. Properties in a Lustre model can be verified by a model checker. We prove properties at an early stage so that by the time SL/SF models are generated, the models have safe constructs, and the properties are already verified.

[1] M. Satpathy, Indian Institute of Technology, Bhubaneswar, Orissa; nuapatana@gmail.com

[2] S. Ramesh, Global GM R&D, Warren, USA; ramesh.s@gm.com

[2] C. Snook and M. Butler, School of Electronics and Computer Science, University of Southampton, UK; {cfs,mjb}@ecs.soton.ac.uk

[3] N.K. Singh, Univ of York; neeraj.singh@cs.york.ac.uk

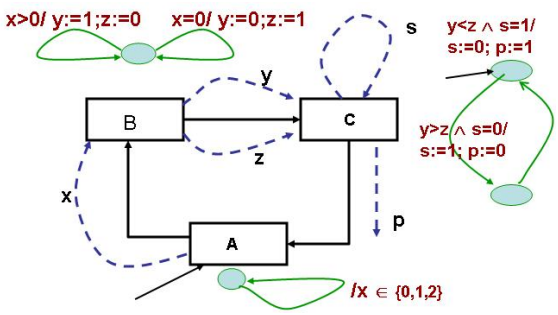

Fig. 1. An RRM Diagram; C is a state holding block.

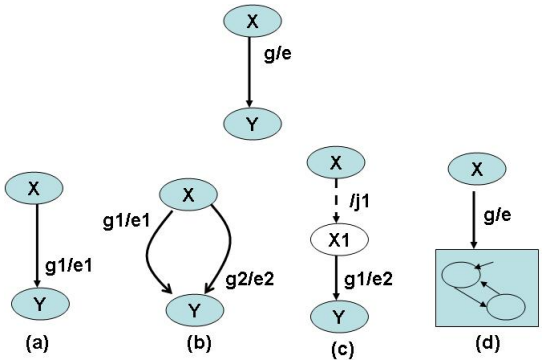

Fig. 2. Refinement of RRMD blocks

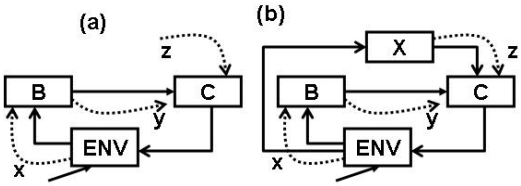

Fig. 3. Closing of open inputs and outputs

Rajhans et al. in [4] discuss compositional verification of heterogeneous systems. They use constraints over parameters at architectural level to define the semantic relationship between the individual components and the system properties. Under certain conditions, the system level safety properties could be verified from the properties of the component models. Our work has similarity with this approach in that development of a logic component imposes some constraints which the modulating part must respect.

III. RRM DIAGRAMS

A. Why RRM Diagrams

The SL/SF modeling notation is widely used in industry for developing control applications. The benefits are: (a) graphical models are better understood, and (b) model-in-loop and processor-in-loop simulation platforms are readily available around SL/SF. The negative points are: (a) models are usually not amenable to formal reasoning, and (b) simulation alone may not remove bugs. Control systems can also be developed using formal methods like Event-B [1]. The plus points are: (a) design steps can be verified, and (b) incremental development can deal with scalability. The negative points here are: (a) the languages are text based, and (b) control/data flow information are not explicit in a model. RRM Diagrams are formal models which incorporate the benefits of both the above formalisms.

B. Structure of RRMDs

Figure 1 shows an RRM diagram, which consists of blocks and connectors. Connectors are of two kinds: control flow and data flow edges. In the figure, they have been respectively shown as solid and dashed lines. A control flow edge determines a block execution order, and a labeled data flow means the source block computes the value of the labeled variable and the target block uses it. Within a block, some computation is performed which is represented as a state machine. Figure 1 shows three computation blocks, A being the initial one. The control flow edges signify that B is computed after A, C is after B, and so on.

The state machines for the respective blocks are as shown in Figure 1. When a block starts execution, one of the enabled transitions of the block's state machine is executed. Execution of transitions within the block continues until there is no enabled transition, control then moves to the next block. In the figure, the lone transition in block A assigns a non-deterministic value to variable x; next, the state machine finishes execution and control moves to block B. Additional flags disable the block A and enables the transition(s) in block B; in the figure, they have not been shown to avoid clutter. In B, either of the transitions is executed which uses x and produces outputs y and z. Next B finishes execution and control moves to C, which uses y and z and produces outputs p and s. Variable s is produced as well as consumed by block C, the dashed self-loop around C signifies that it has a state. p is an open output, not yet used by any block; possibly, a latter refinement would use it.

C. Refinement of an RRMD

RRMDs can be refined in three different ways: (a) refinement of block state machines, (b) closing of open inputs and outputs, and (c) sequencing of parallel control.

Refinement of State Machines: A refinement of a block state machine can be any of (a) strengthening of a transition guard or reduction of non-determinism in the action, (b) case splitting of a transition, (c) adding a transition modeling an internal action, and (c) creation of sub-states and introducing transitions between those. In Figure 2, the transition g/e on top is an abstract transition. Figure 2(a) shows strengthening of this transition; i.e., $g1 \rightarrow g$, and $e1$ reduces the non-determinism in e; $e1$ may also have assignments to new variables. Figure 2 (b) shows the case splitting of the abstract transition into parallel transitions; here, $g1 \vee g2 = g$ and $g1 \wedge g2 = \emptyset$. Figure 2 (c) shows addition of an internal action; i.e., $j1$ can have assignments to new variables, which could be used in $e1$. Figure 2 (d) shows creation of sub-states within a state, and addition of transitions between those.

Handling of open inputs and outputs: State machine refinement of a block may result in creation of new variables,

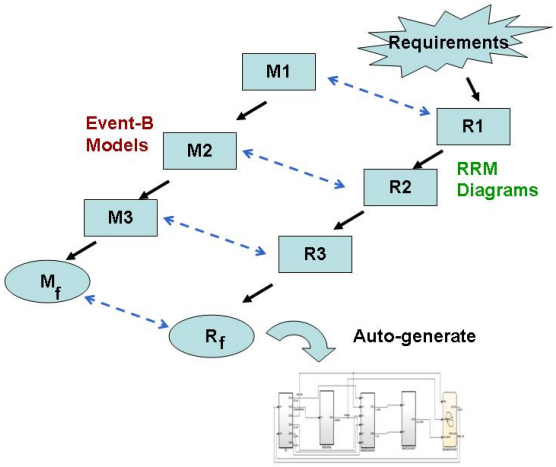

Fig. 4. Development Process using RRMDs

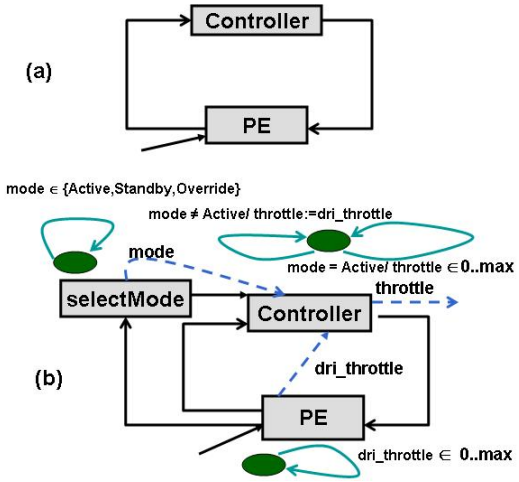

Fig. 5. (a) The initial RRMD, (b) The first refinement

which will be represented as open inputs and outputs; z is an open input in Figure 3(a). An open input can be any of (i) an environmental input, (ii) the input is already produced by another block, or (iii) it is neither. Our aim is to build control applications, so, we assume an RRMD has a special block representing the environment, say ENV. If the open input is from the environment, this is closed by making it an output of ENV. If the input is an output of another block, this is closed by connecting it to the latter. If the input is not from the environment and not yet produced by any block, this has to be an internal input; it would be computed in the subsequent modeling steps. So we create a new block which generates this open input. In Figure 3(b), the open input z has been closed by creating a new block X in a parallel control path, block X – without consuming any input – gives a non-deterministic value to z; the non-determinism in X would be refined at a later stage. The parallel control path is created between ENV and the block which needed this input.

Sequencing of Parallel Control: Two parallel control paths would mean that the two paths can be executed in arbitrary order. So sequencing them can be a refinement step. In Figure 3(b), blocks B and X can be made sequential depending on the data dependency between those.

D. Development Process using RRMDs

Figure 4 shows the development process using RRMDs. The R-chain (R1, R2, R3,...) represents an incremental development chain using RRMDs. R1 models some of the requirements. R2, a refinement of R1, models some more requirements. This goes on until we model all the requirements in R_f. Each R_i has a corresponding representation in the Event-B modeling language [1], we call it M_i. In the figure, M2 is a refinement of M1, M3 is a refinement of M2 and so on. The consistency of each M_i and the refinement relationship between two adjacent Event-B models, are proved using tool support [2]. Thus the correctness of the R-chain is established by proving the M-chain (M1, M2, M3, ...). Finally, a model-generating tool auto-generates a SL/SF frame from R_f. As mentioned earlier, the place holders in the SL/SF frame are filled up with the feedback control laws.

IV. ACC CASE STUDY

The important requirements of an ACC are: (a) Only when CC or ACC is engaged, the throttle value is determined by the controller; in all other cases, the driver determines the throttle, (b) CC/ACC can only operate above a given minimal speed, (c) The driver can activate CC by pressing a button if speed conditions are met, and then the vehicle maintains the cruise speed. (d) When in CC mode, and a forward vehicle appears and the gap is less than a threshold, then ACC is engaged; the task of the ACC is to maintain the set gap with the forward vehicle, (e) When any of CC/ACC is engaged, the driver can take control by pressing either the brake or the throttle. The driver can re-engage the CC/ACC by pressing the resume button provided speed conditions are satisfied, (f) When ACC is engaged, if the controller cannot deal with a situation (say, a vehicle suddenly appearing too close to the host vehicle), then the ACC can raise an alarm until the driver takes control. Until that time, the ACC is expected to control the speed of the host-vehicle.

A. Modeling using RRMDs

Figure 5(a) shows the initial RRMD, where `PE` represents the plant and the environment. There are no data flow edges. This models the requirement that block `PE` gives control to the `Controller` block, and the latter gives control back to `PE`, and this continues.

Figure 5(b) shows the first refinement. The trivial state machine of `Controller` is refined, the new state machine has two transitions. When mode is active – CC or ACC is engaged – throttle gets a non-deterministic value between 0 and max. When mode is not active, the driver determines the throttle (variable dri_throttle). `mode` and `dri_throttle` are two open inputs. The latter is an environmental input; so this is connected to `PE`; its state machine gives a non-deterministic value to `dri_throttle`. `mode` is neither an environmental variable nor it is produced by any block. So a new block – `selectMode` – is created in between `PE`

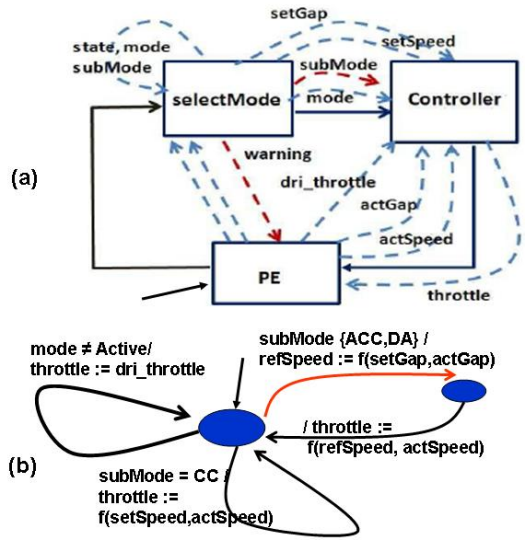

Fig. 6. (a) The final RRMD, (b) State machine of `Controller`

and `Controller` to produce the value of `mode`; this new block assigns a non-deterministic value to `mode`.

Figure 6(a) is the final refined RRMD that captures all the requirements. We omit the details of the intermediate refinement steps. Figure 7 is the state machine of `selectMode` in the final RRMD. Initially control is at the `OFF` state. When the driver presses a switch – input `sw` – and there is no fault, control moves to `Standby`. When the driver presses the `setCC` button and the speed conditions are met, control moves to the `CC` state. When there is a front vehicle within some threshold and the control is already in `CC`, then control moves to the `ACC` state. From the `CC` or the `ACC` states, the driver can take control by pressing either the brake or the throttle – inputs `brake` and `thr` – and then control moves to `Override`. We omit the discussion on all other transitions. Note that block `selectMode` is state-holding.

Figure 6(b) shows the state machine of `Controller`. The bottom transition shows that in CC mode, $f(setSpeed, actSpeed)$ computes the throttle value,

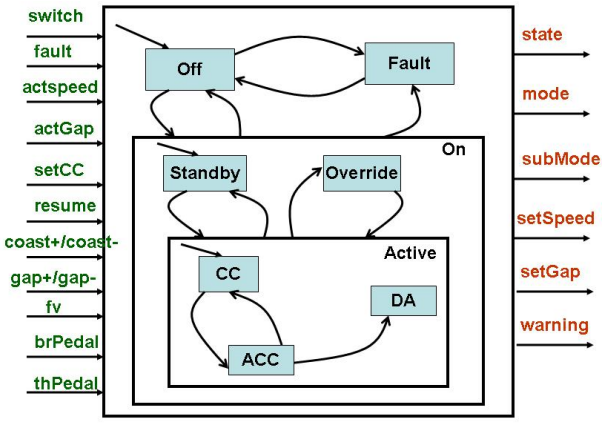

Fig. 7. State machine of `selectMode` in Figure 6

Model	Total number of POs	Automatic Proof	Interactive Proof
Abstract Model	17	17(100%)	0(0%)
First Refinement	224	218(97%)	6(3%)
Second Refinement	104	90(87%)	14(13%)
Third Refinement	184	138(75%)	46(25%)
Total	529	463(88%)	66(12%)

TABLE I
PROOF STATISTICS

where f is a function of the set speed and the actual speed of the host vehicle. f is not yet defined, its input/output types though are known. The sequence of two transitions in the right corresponds to the ACC mode of operation. The first transition computes $refSpeed = g(setGap, actGap)$, where g is another undefined function which computes the speed which needs to be maintained so that the desired gap would be met. The inputs of g are the set gap between the host and the front vehicle, and $actGap$ is the actual gap. In the next transition – in the ACC mode of operation, $f(refSpeed, actSpeed)$ computes the desired throttle value to be sent to the actuator. What will be done with the undefined functions f and g, will be discussed in Section V.

B. Correctness of the RRMD refinements

As shown in Figure 4, the respective RRMDs are translated to Event-B models. If RRMD `R2` is a refinement of RRMD `R1`, let `M2` and `M1` respectively be the corresponding Event-B models. The fact that `M1` and `M2` have a refinement relationship is proved in the Event-B domain using tool support, say Rodin Platform [2]. For the details, refer to [5]. Invariants are also added to the Event-B models, and using tool support it is checked that the models satisfy the invariants. Let us consider the following two invariants:

$$subMode = CC \wedge \ldots \Rightarrow throttle = f(setSpeed, actSpeed)$$
$$subMode \in \{ACC, DA\} \wedge \ldots \Rightarrow throttle = f(g(setGap, actGap), actSpeed)$$

In CC mode of operation, the throttle value is computed by function f. In the ACC and DA modes of operation, first g computes the reference speed, and next f computes the throttle value. Note that both these functions are yet to be defined, yet we are able to prove properties involving those.

Table I shows the proof statistics for our ACC case study when Rodin Platform was used as the tool support. Each row quantifies the proof effort in each step, which contains (a) the proof obligations (POs) generated, the POs automatically discharged by the Rodin prover, and the POs proved interactively. As can be seen from the table, the total no. of POs generated is 529, out of which 88% were proved automatically, and remaining 12% were proved interactively.

V. SL/SF MODELS FROM RRM DIAGRAMS

We will now outline how SL/SF models can be derived from RRMDs. All the control flow edges in the final RRMD can be removed since, after all requirements are captured, the data flow edges already determine the control flow [5]. Now if we ignore the block internals then the final RRMD looks like a SL/SF model; i.e., if we view it as a SL/SF model, the

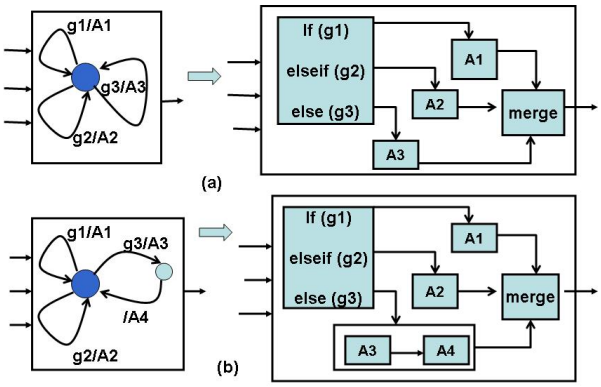

Fig. 8. Rules for RRMD block to Simulink subsystems

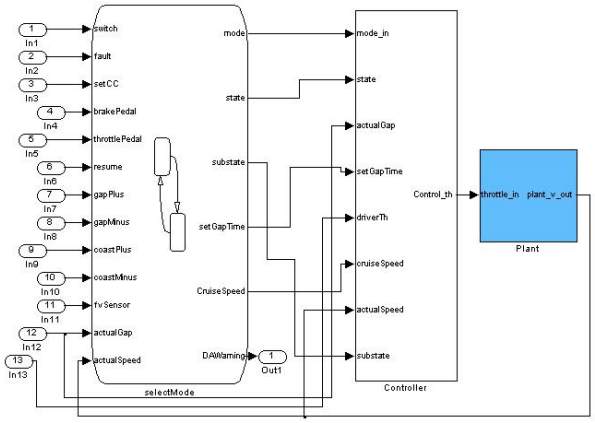

Fig. 9. The SL/SF model for ACC

inter-block connections would remain the same. If we now translate the block internals to Simulink subsystems, then we would get an equivalent SL/SF frame; at this stage, this model has some undefined functions (like f and g discussed in the previous section) which will be derived by control engineers, this we will discuss in the next section.

The role of block PE (Plant and environment) needs some discussion. PE outputs all driver inputs which in the RRMD receive non-deterministic values. All such outputs from PE become external inputs to the generated SL/SF frame. In addition, PE contains the plant function; for us, it is the computation of speed given the throttle value, this could be modeled by a plant model (refer Figure 9).

If an RRMD block is state-holding, we make it a Stateflow subsystem. The structure of Stateflow chart remains exactly the same as the block state machine. The guard/action of a transition in the state machine becomes the guard/action of the corresponding transition in the Stateflow chart; however, the Stateflow syntax needs to replace the RRMD syntax. If a block is stateless, we translate its state machine to a Simulink subsystem. Based on the pattern of the state machine, we define a set of mapping functions [5]; we outline two such functions here: (note: we translate in a certain way though other semantic preserving translations possible)

- If the pattern is as in Figure 8(a), then based on the guards of the transitions, an if-else-if Simulink subsystem is created. In the figure, guards g1, g2 and g3 are mutually exclusive (this is so by construction), and they become the conditions in the if-elseif-else Simulink block; its outputs trigger the subsystems for the transition actions, and the outputs of such subsystems are fed to a merge Simulink block(s); refer to the figure. The guards and actions are translated to the Simulink syntax.
- If the pattern is as in Figure 8(b), the translation is similar excepting the sequencing of the subsystems corresponding to actions A3 and A4.

VI. Implementation

In this section, we will discuss the tool support for the RRMDs and the Simulink model generator. Our RRMDs are encoded as synchronizing state machines in UML-B [5], a graphical formal modeling notation which acts as a front-end for Event-B models [7]. A RRMD-to-Simulink translator translates a RRMD to Simulink by using the mapping rules discussed in the previous section. Figure 9 is the SL/SF model which has been obtained by applying the mapping rules to the RRMD of Figure 6. The environmental inputs due to the driver have become the external inputs to this model; other environmental inputs come from the plant which has been shown as a Simulink subsystem in the SL/SF model. The internals of the Stateflow subsystem – called selectMode – has not been shown here because of space constraints; it resembles the state diagram of Figure 7.

We will discuss the generation of the Simulink subsystem for the `Controller` block in Figure 9, which is generated from the state machine of Figure 6(b). We apply the mapping rule of Figure 8(b). Figure 10(a) shows the corresponding Simulink model. The `if-action-subsystem` encodes the 3 possibilities corresponding to the three disjoint guards in Figure 6(b). The three enabled subsystems in the Simulink model correspond to the transition actions in this figure. The first enabled subsystem is labeled with function `f`, which is a place holder for the cruise control function. The PID controller in Figure 10(b) is the actual cruise control function, designed by a control engineer, which is to replace the place holder labelled with `f` (refer Figure 6(b)).

The third enabled subsystem corresponds to the case when the driver determines the throttle value. The second enabled subsystem – with label `ACC control` – has also place holders, which are also to be replaced by actual control designs, we omit the details here.

VII. Analysis of our Method

On the Mixed Approach: The feedback control law and the discrete logic components of a control system require different skills and expertise. We use an incremental and formal approach to deal with the complexity and correctness of a discrete component. From our interaction with practitioners, we gathered that the discrete component is usually prone to errors, whereas the correctness of a feedback law is easier to establish.

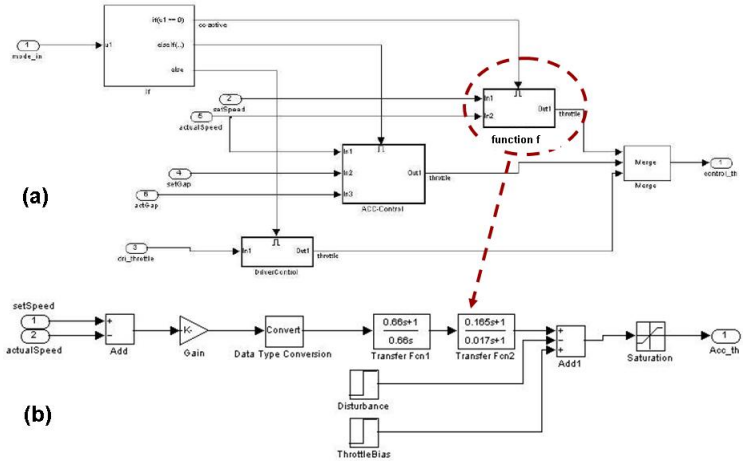

Fig. 10. (a) Simulink subsystem obtained from the RRMD state machine in Figure 6(b); (b) cruise control function developed by a control engineer

When a control design moves from laboratory to the actual plant, the real plant could be different from the plant model. After interviewing control engineers, we have observed that this change is local to the modulating component; the discrete part more or less remains frozen. A control engineer may modify the modulating part without touching the discrete part. So, we can infer that our mixed approach would be effective in industrial practice.

SL/SF model generation: SL/SF is a preferred modeling notation for developing industrial controllers, and engineers are very comfortable with it. If engineers are provided with high quality SL/SF models they would readily accept it; the same we can not say if engineers are directly provided with code. That is why instead of code generation, we have focused on SL/SF model generation.

Our work provides a way of introducing formal methods in current industrial practice, thus it contributes to the formal methods community. We also contribute to the SL/SF community; at present models are obtained manually, whereas our method generates them automatically.

Problems with Manual Generation: Generating SL/SF designs directly from requirements is a big intellectual step, and the process can be error-prone. For example, in the current case study, the generated Stateflow subsystem is complex, and the guards of some transitions have more than 12 conditions, linked together by various logical operators. We believe it would be very difficult to get those right if we model those manually. Since we use formal refinements, and invariants are proved using tool support, we can claim that the SL/SF designs that we generate are of high quality.

VIII. CONCLUSIONS

We have discussed a novel method in which the requirements of a typical controller is partitioned into requirements meant for the modulating component and those for the discrete component. The first component is developed by a control engineer and the second component is developed by a software engineer using a formal approach; both the components are integrated compositionally to obtain the final design. Our mixed approach produces high quality designs for modulating controllers. We have discussed a simplified ACC to illustrate our approach. Even in this simple controller design, there are a few non-trivial features – like hierarchical Stateflow with complex transition guards and actions – which are difficult to get right if a manual approach is used. In future we wish to perform industrial strength case studies and to make our Simulink auto-generation process robust.

ACKNOWLEDGMENT

Initial part of this research was carried out under the Discovery Project titled "Correct-by-construction Methodology for Distributed Automotive Control Software," awarded by Global General Motors R&D. This work was also supported by the project FP7 ICT 287563 ADVANCE (Advanced Design and Verification Environment for Cyber-physical System Engineering; *http://www.advance-ict.eu*). We would like to thank T.K. Ghoshal and S. Mohalik for useful discussions.

REFERENCES

[1] J.-R. Abrial, *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[2] J.-R. Abrial, M. J. Butler, S. Hallerstede, T. Hoang, F. Mehta, and L. Voisin, "Rodin: An open toolset for modelling and reasoning in Event-B," *Intl. J. on Software Tools and Technology Transfer*, vol. 12, no. 6, pp. 447–466, 2010.

[3] M. Kutz, Ed., *"Mechanical Engineers Handbook": Instrumentation, Systems, Controls, and MEMS*. John Wiley & Sons, 2006.

[4] A. Rajhans, A. Bhave, S. Loos, B. Krogh, A. Platzer, and D. Garlan, "Using parameters in architectural views to support heterogenous design and verification," in *IEEE Design and Control and European Control Conference*, 2011, pp. 2705–2710.

[5] M. Satpathy, C. Snook, S. Arora, S. Ramesh, and M. J. Butler, "Systematic development of control designs via formal refinement," in *1st Intnl. Conference on Model Driven Engineering and Software Development, Barcelona*, 2013.

[6] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi, "Defining and translating a "safe" subset of simulink/stateflow into lustre," in *ACM EMSOFT, Pisa*, 2004, pp. 259–268.

[7] C. Snook and M. Butler, "UML-B and Event-B: An integration of languages and tools," in *IASTED International Conference on Software Engineering*, 2008.

[8] H. J. Wade, *Basic and Advanced Regulatory Control:System Design and Application*. The Instrumentation, Systems and Automation Society, 2004.