Computation Platform for Automatic Analysis of Embedded Software Systems Using Model Based Approach

A. Dubey, X. Wu, H. Su, T. J. Koo

Embedded Computing Systems Laboratory
Institute for Software Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University, Nashville, TN 37212
{abhishek.dubey, xianbin.wu, hang.su, john.koo}@vanderbilt.edu

Abstract. In this paper, we describe a computation platform called ReachLab, which enables automatic analysis of embedded software systems that interact with continuous environment. Algorithms are used to specify how the state space of the system model should be explored in order to perform analysis. In ReachLab, both system models and analysis algorithm models are specified in the same framework using Hybrid System Analysis and Design Language (HADL), which is a meta-model based language. The platform allows the models of algorithms to be constructed hierarchically and promotes their reuse in constructing more complex algorithms. Moreover, the platform is designed in such a way that the concerns of design and implementation of analysis algorithms are separated. On one hand, the models of analysis algorithms are abstract and therefore the design of algorithms can be made independent of implementation details. On the other hand, translators are provided to automatically generate implementations from the models for computing analysis results based on computation kernels. Multiple computation kernels, which are based on specific computation tools such as d/dt and the Level Set toolbox, are supported and can be chosen to enable hybrid state space exploration. An example is provided to illustrate the design and implementation process in ReachLab.

1 Introduction

Embedded software systems are becoming an integral and ubiquitous part of modern society. They are often used in safety critical tasks such as in airplanes and nuclear reactors. Typically, they consist of one or more discrete software components performing computation on a real-time operating system (RTOS) to control the continuous environment. Fig. 1 shows a typical embedded software system, in which the continuous state of plant is controlled by software control tasks. The control task and the plant exchange information of continuous state x and input u via sensors and actuators. In a very simple case, the sensor can be a periodic sampler, while the actuator can be a zero order hold.

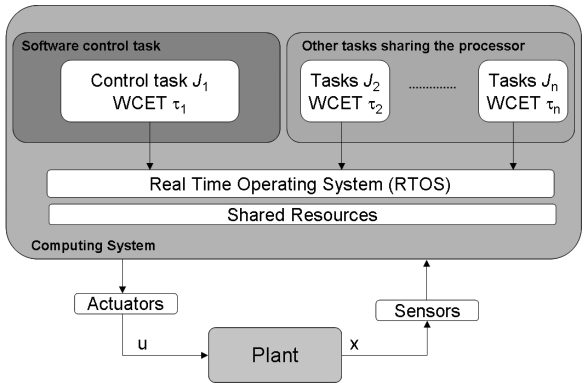


Fig. 1. A typical embedded software system.

To ensure high confidence in these systems, rigorous analysis is required before deployment. However, it is often infeasible to perform analysis on the actual system due to its scale and complexity. Model based approach has been advocated for design and analysis of these complex systems in order to produce confidence in the design and reduce development costs. In this approach, representative models of the system are judiciously used to predict its behavior and analyze various properties. Hybrid automaton [1, 2, 13] has been used to model and analyze embedded systems in which discrete and continuous components are tightly coupled.

In order to automate the analysis of hybrid automata, algorithmic approach has been developed. Algorithmic approach can be classified into two categories: reductionist methods and symbolic methods [3]. The former reduces the infinite hybrid (discrete and continuous) state space to an equivalent finite bisimulation and then explores the resulting finite quotient space, while the latter performs direct exploration of this infinite state space. Even though the reductionist method based algorithms are guaranteed to terminate, the classes of systems to which they can be applied are very limited. Therefore, symbolic method based algorithms are generally used. Various computation tools with vastly different implementations have been developed for symbolic method based analysis. For example, d/dt [5] computes reachable sets by approximating reachable states based on numerical integration and polyhedral approximation; whereas the Level Set toolbox [4], which applies the level set methods [14], computes the evolution of a continuous set by solving the associated partial differential equation on grid structure. Due to these implementation differences in computation method, data structure as well as analysis purpose, designing new analysis algorithms by using or modifying existing tools becomes infeasible or inefficient. Furthermore, designing a common interchange format [8] for these tools is difficult.

In order to resolve the analysis problem, the computation platform called ReachLab is designed to enable (i) separating the concern of algorithm design for analysis of hybrid automaton model from any specific computation implementation; (ii) separating the design of algorithm from specific hybrid automaton model so that the same algorithm can be reused for other system models. ReachLab is developed based on the Model Integrated Computing (MIC) [6, 7] approach.

MIC approach is based on models and automatic generation of useful artifacts. In this approach, models are used not only to design and represent the system, but also to synthesize and implement the system using a modeling language tailored to the needs of a particular domain. These modeling languages, termed as Domain Specific Modeling Languages (DSML), have necessary constructs to allow the capture of useful information of a system as model particular to that domain. One can perform *system analysis* on this model. When this modeling capability is augmented with the capability of model transformation, even *automated synthesis of other design models*, and *generation of executable system* can be performed [7].

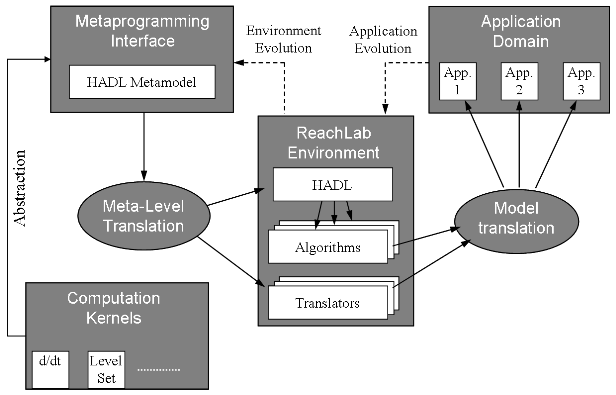


Fig. 2. Design of the ReachLab platform using the MIC multigraph architecture.

Based on MIC approach, the domain specific modeling language for analyzing hybrid systems called Hybrid System Analysis and Design Language (HADL) is introduced. Specified by meta-models, it provides a rich library comprising of abstractions of entities and operations commonly found in the symbolic method based computation tools, so that it enables effective design of symbolic method based analysis algorithm for systems modeled as hybrid automata. Then, we will focus more on ReachLab which utilizes this language to design system models and corresponding analysis algorithms, and provides various model translators to implement the models using the facilities provided by Generic Modeling Environment (GME) [9], which provides an end-end solution for building and deploying MIC applications. By keeping the implementation of computation method of computation tools, and enriching them with additional features such as support for comprehensive data structures implemented by existing functions provided in these tools, various computation kernels have been supported by ReachLab, such as d/dt kernel and Level Set kernel. Model translators are used to automatically generate model implementations for these computation kernels. Fig.2 shows how MIC approach is applied to encapsulate HADL and automate the design and implementation process based on the MIC multigraph architecture [10].

This architecture has three model development stages, namely meta-model, domain specific models and the executable artifacts. The first level is the meta-programming interface, which is used to define the meta-model of HADL. This meta-model is based on abstract entities found in the symbolic method based computation kernels and is later implemented as the domain specific modeling language, HADL, using the meta-translation facility provided by GME. Model-Integrated Program Synthesis (MIPS) environment [11] is the second level and provides tools to build and modify system models and the analysis algorithms using HADL in a graphical manner. This level also supports construction of model translators. The last level is the different applications (implementations) that can be generated by translators from these models. Environment evolution refers to modification of HADL meta-model to update features. The models of algorithms can also be refined to evolve the analysis application.

The remainder of this paper is organized as follows: Section 2 gives an introduction to HADL. Section 3 presents the architecture of ReachLab and the details about ReachLab construction, including the model translation process. An example is provided to illustrate the design and implementation process in ReachLab in Section 4. Finally, we conclude our work with the future goals for this platform.

2 Introduction to HADL

HADL is a language that enables the design and analysis of hybrid automata. For this design and analysis purpose, HADL is used to specify models of hybrid automata and corresponding analysis algorithm models. In [13], the mathematical definition of a hybrid automaton is given as a collection $H = (Q, \mathbb{X}, f, I, E, G)$ where $Q = \{q_1, \ldots, q_N\}$ is a set of discrete modes; $\mathbb{X} \subseteq \mathbb{R}^n$ is the continuous state space; $f : Q \to (X \to \mathbb{R}^n)$ assigns each discrete mode a Lipschitz continuous vector field on $\mathbb{X}$; $I : Q \to 2^{\mathbb{X}}$ assigns each $q \in Q$ an invariant; $E \subseteq Q \times Q$ is a collection of discrete transitions; $G : E \to 2^{\mathbb{X}}$ assigns each $e = (q, q') \in E$ a guard. The analysis algorithm model specified in HADL is hierarchical in nature, and complex algorithms can be composed from existing algorithms by using them as subroutines. Data variables used in analysis algorithms are strong-typed, and currently, only global scoping is supported. However, in the future, it will allow local scoping as well.

HADL has been formalized as a five tuple of concrete syntax (C), abstract syntax (A), semantic domain (S), semantic mapping (M_S) and syntactic mapping (M_C) [16]:

$$L =< C, A, S, M_S, M_C > .$$

Concrete syntax (C) defines the graphical notation used to specify the models. Abstract syntax (A) specifies all the syntactical elements of the language, as well as the integrity constraints. Semantic domains (S) is defined by formalism which provides meaning to a correct sentence in the language. The mapping $M_S : A \rightarrow S$ relates every element of abstract syntax to a specific meaning in the semantic domain. Model translators are used for this semantic mapping. The mapping $M_C : A \rightarrow C$ assigns a notational construct to every elements of abstract syntax.

Advocated by the MIC approach, HADL is formalized by meta-models. It is designed to enable the use of multiple aspects [7, 9] to help decompose any analysis application designed in HADL into three separate components – data (data aspect), the system model (system aspect) and algorithm model (programming aspect). Hence, the abstract syntax of HADL can be written as a three tuple

$$A =< L_{data}, L_{system}, L_{program} > .$$

The semantic domain S of HADL is any chosen supported computation kernel. Model translators can be used to provide the semantic mapping $M_s : L_{data} \times L_{system} \times L_{program} \rightarrow S$. Hence, a translator is required for each semantic domain.

As part of the HADL's abstract syntax, integrity constraints can be checked by using Object Constraint Language (OCL) [18], which guarantees the correctness of designed models. The other part of the abstract syntax, the syntactical elements in these three aspects, provide basic notions and constructs to specify hybrid automaton models, analysis algorithms as well as the data variables used in these algorithms. To be specific, these elements are comprehensively listed in Table 1.

HADL has been provided with precise mathematical semantics, which are generic and not dependent on implementation details. For example, the discrete successor operation in HADL, denoted as $Post_d$, is defined as $Post_d(q_i) = \{q \in Q \mid \exists e \in E \ s.t. \ e = (q_i, q)\}$. This operation specifies the collection of reachable discrete states of the hybrid automaton in a single discrete transition. Similarly, the constraint continuous successor operation in a single step Δt, notated as $cPostc_{\Delta t}$, is defined as $cPostc_{\Delta t}(q_i, P, X_\psi) = \{x \in \mathbb{X} \mid \exists t \in [0, \Delta t], \exists y \in P \ s.t. \ x = \phi(t, y) \wedge \forall z \in [0, t], \phi(z, y) \in I(q) \cap X_\psi\}$ where P is the initial continuous set, $\frac{d}{dt}\phi(t, y) = f(q_i, \phi(t, y))$, and $X_\psi = \{x \in \mathbb{X} \mid \psi(x) \leq 0\}$ defines the constraint continuous set. This operation specifies the collection of reachable continuous state set of the hybrid automaton in a single time step Δt. By using such reachability operations and algorithmic approach, many properties of a hybrid automaton can be revealed, such as safety or liveness. However, it is known that computation of exact or even approximate continuous successor sets is a difficult problem due to representing continuous sets and computing the evolution of the sets. Existing computation kernels adopt different methods to approximate it. For example, kernels like Level Set kernel and d/dt kernel are tailored to their own analysis needs and computation capacities so that the implementations of these reachable set operations as well as Boolean set operations (such as union

Table 1. HADL Language Syntactical Elements

Aspect	**Model of**	**Syntactical Elements**
Data	Data	Primitive data types: integer, float, Boolean; Data structure: multi-dimensional list.
System	Hybrid automaton	Discrete mode, associated with invariant; Discrete transition, associated with guard and reset; Continuous set and initial continuous set; Analysis set, as a specialization of continuous set; Computation parameters.

Programming	Control flow	Routine, hierarchical in nature;
		Looping: "*while*" loop;
		Branching: "*if-then-else*";

	Operators	Primitive data operations: $+, -, *$; Logical operations: equal, less than, and, or, not; Multi-dimensional list operations: new, delete, append, element; Reachable set operations: discrete successor and predecessor, (constraint) continuous successor and predecessor in a single step (in bounded time), reset, projection, visualization; Boolean set operations: intersection, union, complement.

and intersection) are quite different. HADL is defined based on the mathematical definitions of these operations and HADL is designed to ensure there exists a correspondence between the semantics of these kernels and the semantics of HADL. Therefore, one can use the semantics of HADL to anchor the semantics of these kernels, which is referred to as *semantic anchoring* in [17]. Because of this feature, we can design analysis algorithms by using the mathematical semantics of these operations instead of considering the detailed implementation. Furthermore, HADL enriches the functions of its computation kernels by providing constructs and operations more than these computation kernels, such as multi-dimensional list and its corresponding operations. These constructs and operations will be mapped to a collection of entities in the computation kernel rather than a direct mapping.

The advantage of using this language is that (i) algorithms are designed independently from implementation and hence can be used with any supported computation kernel; (ii) analysis algorithms can be reused for different systems; (iii) more complex algorithms can be constructed by using other existing algorithms.

3 Construction of ReachLab

In this section, the architecture of ReachLab is introduced and the construction issues related to model traversal and semantic mapping are presented.

3.1 ReachLab Architecture

By utilizing the language defined by HADL, a computation platform called ReachLab has been developed, and its architecture, as shown in Fig.3, is designed to separate the concerns of algorithm design from implementation details. The MIPS environment of ReachLab, facilitated by GME, provides support to build graphical algorithm and system models. Different graphical model entities and components are connected according to the rules specified by HADL meta-model. Therefore, models can be designed in ReachLab graphically according to HADL specification. Besides model design, the other key process is the use of translators to automatically translate the models into executable artifacts. This translation process requires mapping of the abstract entities into concrete implementations for the target domain of a computation kernel. In [7], the translation process has been summarized as a graph transformation: (i) *Creation of "input graph"* : The models with different interconnected components are implicitly represented by a graph structure. (ii) *Model traversal and Semantic mapping* : The translation process requires creation of a "target graph" (data structure for the executable artifact) from an "input graph". This requires the translator to traverse various objects in the "input graph", recognize their patterns and calculate attributes of output objects in the "target graph" using semantic mapping. The "target graph" corresponds to the data structure required to represent the output form of the executable artifacts.(iii) *Printing the product* : In this step, the translator serializes the "target graph" to generate executable artifacts pertaining to the related domain.

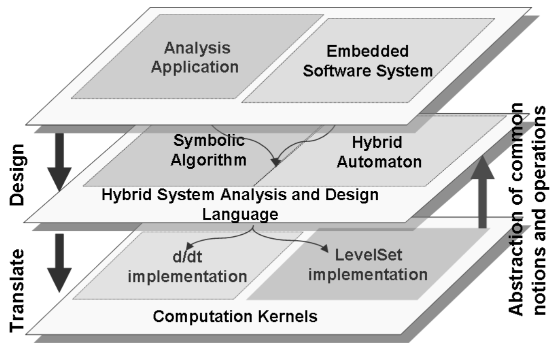


Fig. 3. The three-layer ReachLab architecture.

In ReachLab, the traversal process uses the data structures provided by GME to store the "input graph" along with necessary information. These data structures are very generic and remain the same for different translators. However, the data structures used to store the "target graph" vary due to implementation differences among different computation kernels.

In the next subsection, we will explain in detail how model traversal is done to fulfill model translation process.

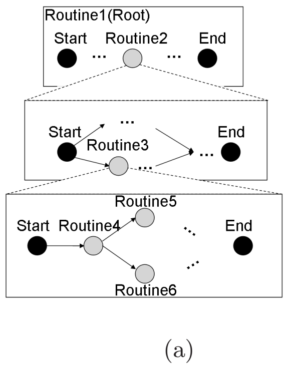
3.2 Model Traversal

Translators need to perform the traversal of all three aspects in order to understand the patterns and collect all useful information. This traversal process is based on graph search techniques such as depth first search [12]. The complete process can be broken down into four sub-tasks reviewed below.

Traversal of Data Aspect: All the data are defined in one single data folder as a list. Translator traverses this list in a linear fashion to collect all useful information about the data elements.

Traversal of System Aspect: The hybrid automaton model specified in the system aspect can be understood as a graph, in which the discrete modes are vertices and the discrete transitions of hybrid automaton are the edges. The translators traverse this graph by using depth first search starting from the initial discrete state to collect all useful information.

Traversal of Control Flow of Algorithms:



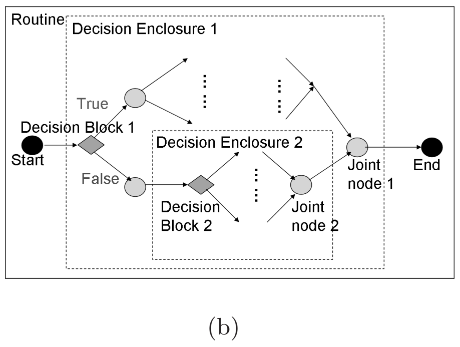


Fig. 4. (a) The components of a routine are interconnected as a DAG. Routines may be hierarchical leading to a hierarchical graph; (b) The decision enclosure is sub-graph starting from a decision block and ending at its corresponding joint-node.

The traversal of programming aspect is more complex. Every algorithm has a root *routine* which is the entry point to the algorithm. *Routines* can be hierarchical and may contain other sub-routines as shown in Fig.4(a). The control flow inside each *routine* routes from a *"start"* to an *"end"* . However, there might be other exit routes from a *routine* through *"break-exit"*, which is used in the same way as the break in many programming languages. For example, the constraint continuous successor set operation in bounded time T, denoted as $cPostc_T$, can be implemented by iterating $T/\Delta t$ times by calling $cPostc_{\Delta t}$, which is previously defined. Therefore, the routine to implement $cPostc_T$ can use the routine of $cPostc_{\Delta t}$ as its sub-routine. The language also provides a specialization of *routine* called *while routine* for implementation of looping constructs such as *do-while* which is traversed in the same manner as a *routine*. The control flow inside a routine is sequential, however it can have multiple branches due to *decision blocks*. Cycles in the control flow are disallowed to demote the use of sudden jumps such as "goto". Therefore, the control flow inside each routine is a directed acyclic graph (DAG) [12] with its directed edges depicting the route of control flow and each node depicting a block of algorithm. Since routines can contain other routines, the overall control flow of the complete algorithm is a hierarchical DAG. The translators traverse the graph structure of algorithms in

Table 2. Decision-Enclosure Algorithm

```
Input:
    $DecisionBlock$ = the starting node of the enclosure
Initialization:
    $InitPath := DecisionBlock$
    $Paths := \{InitPath\}$
Start:
While $true$ do
    For each $path$ in $Paths$ do
        $Fringe :=$ the tail of $path$
        $Succ :=$ successor nodes of $Fringe$
        If $Succ \neq \phi$ then
            Add $Succ[0]$ to the fringe of $path$
            $Succ := Succ - Succ[0]$
            For each $s$ in $Succ$ do
                $path' := path$
                Add $s$ to the fringe of $path'$
                Add $path'$ to $Paths$
            End For
            If $\exists s \in Succ\ s.t.\ \forall p \in Paths, s \in p$ then
                Return $s$ as the joint-node
            End If
        End If
    End For
End While
```

a depth-first search manner to extract information. In each routine, the traversal starts from *"start"* block and follows the directed edges. If any of the traversed entity is hierarchical, translators will traverse its subcomponents in a depth-first manner. *Decision blocks* are used inside routines to design a logical branching in the control flow sequence. For each of these blocks, the branching starts from itself, and finally merges at a *joint-node*. The sub-graph enclosed by the *decision block* and the joint-node in the DAG is called a *decision-enclosure*. This is illustrated by Fig. 4(b). The traversal algorithm has to recognize the *"if true"* and *"if false"* part of each *decision block* so that they can be mapped to the corresponding decision logic in the implementation. This requires knowledge of its *decision-enclosure*. Table 2 gives an algorithm based on breadth first search technique for determining *decision-enclosure* of each *decision block*. This algorithm has a complexity of $O(n^2)$, where n is the number of blocks in the DAG.

The key of this algorithm is to find the joint-node, and since a joint-node is where all branches from the *decision block* merge, by using breath-first search and keeping all branching paths from the *decision block*, the first block that belongs to every recorded branching paths is the joint-node.

Traversal of Operators: Operators are used for data manipulation. Every assignment expression forms a tree structure, with the left-hand-side data variable as the root of the tree. All data variables on the right-hand-size of the expression correspond to the leaves of this tree, and operators on the right-hand-side correspond to the internal nodes of the tree. The expression itself can be restored to reverse-polish notation by post-order traverse.

The operators have different semantic meanings depending on the input data types. And since HADL is "strong-typed", the data types of the tree leaves, which are predefined, will finally determine the input data type of the operator connected to the root data. Therefore, it is important to propagate the data type information from leaves to the root in a post-order manner [12].

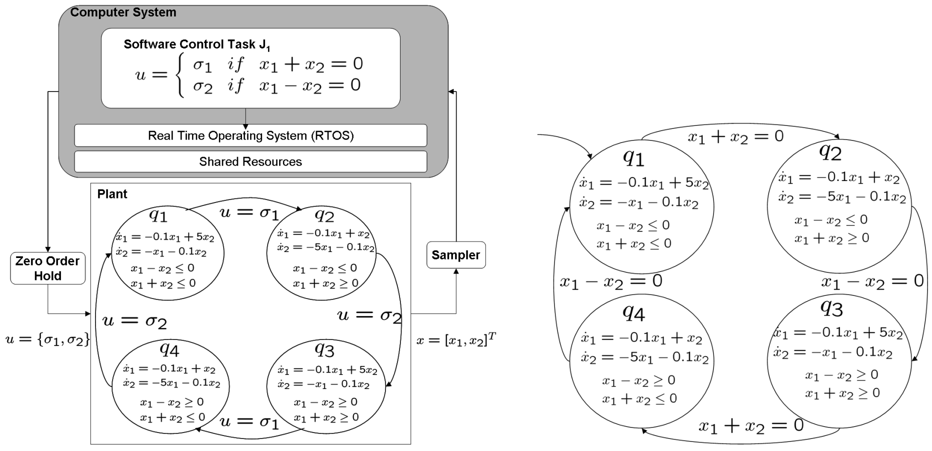
3.3 Semantic Mapping

Since the semantics of a computation kernel are anchored to the semantics of HADL, we can find a corresponding implementation for HADL constructs in the computation kernel. These constructs include sequential programming features, boolean operations on state sets, as well as the reachable set operations. However, in some cases, the operations, such as data structure manipulation operations, are not directly supported by the computation kernel and have to be specifically added to the computation kernel as new functions. The process of associating the HADL constructs to its implementation in computation kernel is akin to providing a meaning to them and is therefore referred to as semantic mapping.

We will illustrate some of the aspects of the semantic mapping process by using the example of Level Set kernel. Level Set kernel has been implemented as Matlab functions. It supports all the basic data types in HADL except the multi-dimensional list structure, which we have to specifically implement along with the relevant operations in Matlab. The hybrid system specific data types such as *discrete mode* and *continuous set* are mapped to Matlab *struct* and *mesh* on analysis space, respectively. This *mesh* is an internal structure used by Level Set kernel. The control flow inside a routine is mapped to the sequential flow of logical commands inside a function. We use *"if-else-end"* statement in Matlab to implement branching and *"while-end"* statement in Matlab to implement looping. Boolean operations on state sets and reachable set operations are mapped to their corresponding implementation in Level Set kernel. However, for some of the operations defined in HADL, there are no straight-forward mappings, therefore we have to write specialized functions for them by using operations provided by the kernel.

4 Design and Implementation Process in ReachLab

In this section, we will illustrate the design and implementation process for analysis algorithms in ReachLab by designing a forward reachability analysis algorithm for the embedded software system shown in Fig.5(a).



(a) (b)

Fig. 5. (a) An embedded software system. The plant on the bottom has four running modes with different continuous dynamics, controlled by the software control task J_1; (b) Hybrid automaton model for the control task and plant. It has four discrete modes corresponding to the four running modes of the plant, and one continuous state $x = [x_1, x_2]^T \in \mathbb{R}^2$.

Depending on the current state of the plant, it determines input $u \in \{\sigma_1, \sigma_2\}$. By considering the direct interaction between the control task and the plant, we can model the system as a hybrid automaton as shown in Fig.5(b). Multiple tasks which share common resource with the control task, the scheduler and the interface elements such as sampler and the zero order hold can be modeled by a more complex hybrid automaton.

It has been shown in [15] that this system is stable in the sense of Lyapunov. Starting from anywhere in the continuous state space, the continuous state of the automaton moves toward the origin in a flower-like trajectory. For this system, we are interested in computing forward reachable set using symbolic methods based algorithms, in order to verify that starting from certain initial state, whether or not the system can eventually enter some desired set.

Table 3. Algorithm for computing forward reachable set

```
Input:
    $H_A$, $Q_S, X_S$, $Q_F, X_F$, $X_B$, where
    $Q_S$ is list of initial discrete modes, $X_S$ is list of initial continuous sets, $Q_F$ is
    list of final discrete modes, $X_F$ is list of final continuous sets, and $X_B$ is bad set.
Constant:
    $T$ as time limit for $cPost_Tc$, $M$ as search depth limit
Initialization:
    $Reach = X_S, List = \{\}, Successors = \{\}, R = \phi, Queue = Q_S$
    $Depth = 1, i = 0, j = 0$
Start:
    While $\neg$Empty($Queue$) do
        $List$ = PopAll $Queue$
        For $i = 1$ : Size($List$) do
            $R = cPostc_T(List(i), Reach(i))$
            $Successors = Postd(List(i))$
            For $j = 1$ : (Size($Successors$) do
                If $R \cap Guard_{List(i),Successors(j)} \neq \phi$ Then
                    Push $Successors(j) \rightarrow Queue$
                    Append $R \cap Guard_{List(i),Successors(j)} \rightarrow Reach$
                End If
            End For
        End For
        $Depth = Depth + 1$
        If $Depth > M$ Then
            Stop
        End If
        Pop first Size($List$) elements of $Reach$
    End While
```

Table 3 gives the specification of a generic forward reachability algorithm for hybrid automaton. It uses the concepts of both discrete and continuous successor set and finds the reachable set starting from a given initial set. This algorithm unfolds the hybrid automaton into a tree like structure and explores it by using breadth first search. Termination of this algorithm is guaranteed because of the limit M on the depth of this tree. The data structure *Reach* is used to store the reachable set. It can be noted that this specification does not delve into the actual implementation method of reachable set operations. However, the process of semantic mapping will relate those operations to a specific implementation method based on the concerned computation kernel. This algorithm can be used to verify if the system would ever execute into some desired state. In order to perform verification, the algorithm systematically explore the hybrid state space and check if the forward reachable set overlaps with the desired set. The main concern with this type of algorithms is termination. But if we perform the computation in an *Eulerian* framework (one in which the underlying coordinate system is fixed) within a bounded continuous state space, the algorithms will terminate due to the fact that the partition of state space has finite number of representative elements.

4.1 Design Steps

To analyze the safety property of the hybrid automaton model in Fig.5(b) by using the forward reachability algorithm, we need to design its hybrid automata model in the system aspect and design the algorithm in the programming aspect. The data used in both of the system model and the algorithm are defined in the data aspect. The entire process can be summarized into three steps:

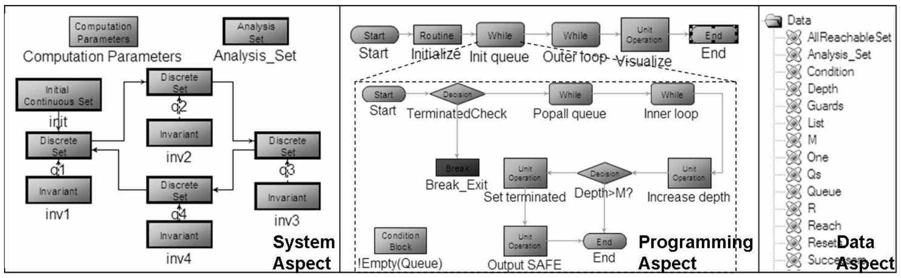


Fig. 6. Hybrid automaton model for the corresponding plant in the system aspect, forward reachability analysis algorithm model in the programming aspect, and data used in the data aspect of ReachLab.

1. **Obtaining system model and algorithm specification**:
   Fig.5(b) and Table 3 provide the hybrid automaton and analysis algorithm specifications for this example.
2. **Design phase**:
   - Design of the system model: A hybrid automaton is drawn in the system aspect with discrete transitions connecting discrete modes, as in Fig.6.
   - Design of the analysis algorithm: The analysis algorithm, which is hierarchical in nature, is modeled in the programming aspect by using ReachLab library elements. Fig.6 also gives part of the algorithm model for the algorithm given in Table 3, and the data required by both the hybrid automaton and the algorithm model.
   - Specification of computation parameters: Input parameters to the algorithm and the computation kernels have to be specified before translation, such as the bounded time (T) for $cPost_{cT}$ operator, the analysis region, and how the analysis region is partitioned into finite number of representative elements.
3. **Implementation phase**:
   Translators are used to convert the designed models into implementation for

a certain computation kernel. For this example, we translate the system and algorithm model into the d/dt implementation. Fig.7 shows the computation result. This result can be used to examine system behaviors, such as approaching the origin while evolving. It can also be used to verify system stability properties by testing intersection between the reachable set and the desired set.

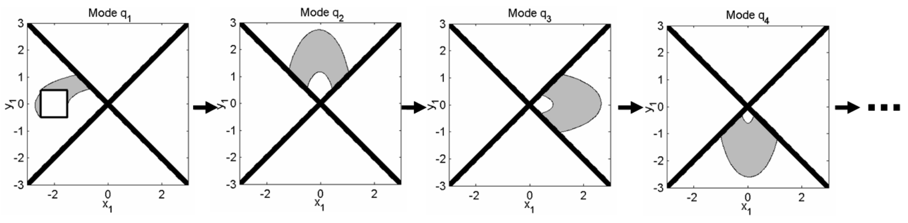


Fig. 7. The reachable set computed by using d/dt kernel. The white box is the initial set, [-2.5,-1.5]x[-0.5, 0.5]. Each sub-figure denotes the reachable set in the corresponding discrete mode. Eventually, the reachable set will reach the origin. The analysis region is $[-3, 3] \times [-3, 3]$, the size of a representative elements in each dimension is 0.001, and the bounded time T is 5 seconds. Time taken for execution: 180 minutes on Pentium IV 2.59 GHz machine with 2 GB RAM.

5 Conclusion

In this paper, we presented the computation platform called ReachLab for enabling automatic analysis of embedded software systems modeled as hybrid automata. It implements the meta-model based language HADL whose abstract entities allow users to model their algorithms and the system in an implementation independent manner. These models are then translated to implementations for different computation kernels. Translation is performed by using model traversal and sematic mapping. Currently, d/dt kernel and Level Set kernel are supported by ReachLab. In the future, we will expand this platform to other computation kernels for more effective and efficient computation. In order to model networked hybrid automata, shared variable could be introduced to ReachLab for specifying communication protocols between hybrid automata. We are also interested in expanding the capabilities of HADL to capture a larger class of embedded software systems so that more sophisticated system features can be described.

6 Acknowledgments

This work is supported by NSF's Faculty Early Career Development (CAREER) Program, Award No. 0448234, and Department of EECS, Vanderbilt University.

References

1. T. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, (1996), pp. 278–292.
2. R. Alur, D. L. Dill. A theory of timed automata. *Theoretical Computer Science 126*, (1994), pp. 183–235.
3. T.A. Henzinger, R. Majumdar. A classification of symbolic transition systems. In *Proceedings of the 17th International Conference on Theoretical Aspects of Computer Science* (2000), pp. 13–34.
4. I. Mitchell, J. A. Templeton. A toolbox of Hamilton-Jacobi solvers for analysis of nondeterministic continuous and hybrid systems. In *Hybrid Systems: Computation and Control*, (2005), pp. 480–494.
5. E. Asarin, T. Dang, O. Maler. The d/dt tool for verification of hybrid systems. In *Computer Aided Verification*, (2002), vol. 2404 of *LNCS*, Springer-Verlag, pp. 365–370.
6. G. Karsai, A. Agrawal, A. Ledeczi. A metamodel-driven MDA process and its tools. Workshop in Software Model Engineering, (2003).
7. G. Karsai, J. Sztipanovits, A. Ledeczi, T. Bapty. Model-integrated development of embedded software. In *Proceedings of the IEEE*, (2003), pp. 145–164.
8. A. Pinto, A. L. Sangiovanni-Vincentelli, L. P. Carloni, R. Passerone. Interchange formats for hybrid systems: Review and proposal. In *Hybrid Systems: Computation and Control*, (2005), pp. 526–541.
9. A. Ledeczi, M. Maroti, A. Bakay, et al. Generic modeling environment. In *International Workshop on Intelligent Signal Processing*, (2001).
10. J. Sztipanovits, G Karsai, C. Biegl, T. Bapty, A. Ledeczi, D. Malloy. Multigraph: an architecture for model-integrated computing. In *Proceedings of the 1st International Conference on Engineering of Complex Computer Systems*, (1995), pp. 361–368.
11. J. Sztipanovits, G. Karsai, H. Franke. Model-integrated program synthesis environment. In *Proceedings of the IEEE Symposium and Workshop on Engineering of Computer Based Systems*, (1996), pp. 348–355.
12. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein . Introduction to Algorithms, Second Edition, (2001), The MIT PRESS.
13. J. Lygeros. Lecture Notes on Hybrid Systems. Cambridge, 2003.
14. S. Osher, R. Fedkiw. Level Set Methods and Dynamic Implicit Surfaces. Springer, 2003.
15. A.Rantzer, M. Johansson. Piecewise linear quadratic optimal control. In *IEEE Transactions on Automatic Control*, (2000), pp. 629–637.
16. T Clark, A Evans, S Kent, and P Sammut. The mmf approach to engineering object-oriented design languages. In *Workshop on Language Descriptions, Tools and Applications.LDTA*, Genova, Italy, 2001. Available via http://www.puml.org.
17. Kai Chen, Janos Sztipanovits, and Sandeep Neema. Toward a semantic anchoring infrastructure for domain-specific modeling languages. Fifth International Conference on Embedded Software (EMSOFT05), Jersey City, New Jersey, September 2005. (Accepted for publication).
18. et a.l, R. S. C. *Object Constraint Language Specification ver 1.1*, Sept 1997.