THÈSE

présentée à

UNIVERSITÉ PARIS XII, Val de Marne

pour obtenir le titre de

DOCTEUR EN SCIENCES

Spécialité

Automatique

soutenue par

Masoud NAJAFI

Titre

Solveur Numérique pour les Systèmes Algébro-Différentiels Hybrides

(The Numerical Solver for the Simulation of the Hybrid Dynamical Systems)

Directeur de thèse : **Ramine Nikoukhah**

Jury

Mr.	**F.**	**Rocaries**	Président
Mr.	**S. L.**	**Campbell**	Rapporteurs
Mme.	**Z.**	**Benjelloun-Dabaghi**	Rapporteurs
Mr.	**S.**	**Steer**	Examinateur
Mr.	**Y.**	**Candau**	Examinateur
Mr.	**R.**	**Nikoukhah**	Directeur de thèse

A mes parents...

Remerciements

Je tiens à remercier en tout premier lieu Monsieur Ramine Nikoukhah, directeur de thèse à l'INRIA, pour son encadrement, sa compétence, ses conseils et surtout pour sa patience et sa disponibilité durant cette thèse.

J'exprime ma profonde gratitude à Monsieur Stephan L. Campbell à l'Université NCSU (North Carilina State university) pour accepter d'être rapporteur de ma thèse, et surtout pour son aide précieux, ainsi que pour ses commentaires sur mon mémoire.

Je remercie Madame Zakia Benjelloun-Dabaghi, ingénieur de recherche senior à l'IFP (Institut Français du Pétrole) d'avoir accepté d'être rapporteur de ma thèse et l'intérêt qu'elle a porté à mon travail.

Monsieur François Rocaries, professeur à la laboratoire A2SI (Algorithmique et Architecture des Systèmes Informatiques), pour m'avoir fait l'honneur de présider le jury de thèse.

Monsieur Yves Candau, professeur à la université de Paris XII val de Marne, pour avoir accepté de juger cette thèse et pour l'intérêt qu'ils ont porté à mon travail ainsi que pour ses jugements très pertinents sur mon manuscrit, tant sur le fond que sur la forme.

Monsieur Serge Steer, directeur de recherche à l'INRIA qui m'a fait l'honneur de participer au jury de cette thèse.

Je désire également remercier monsieur Daniel Bouskela, chercheur EDF (Électricité de France) et monsieur Sébastien Furic ingénieur de recherche IMAGINE pour leur aides précieux durant cette thèse.

Mes remerciements vont aussi à tous les membres actuels et anciens du projet METALAU et l'équipe Scilab et plus généralement tous ceux que j'ai côtoyés ces dernières années pour leurs gentillesses et la qualité des échanges, aussi bien techniques qu'humains, que nous avons eues.

Je n'oublierai pas les aides permanentes reçues du personnel de l'INRIA, Martine Verneuille, Eric Gallula, Faouzi Rahali.

Mes amitiés à mon collègue de bureau mon très proche copain Azzedine qui m'a supporté durant ces années et je remercie également Rachid pour ses soutien moral. Je n'oublie pas, bien sûr, de remercier Abdelaaziz, Amel, Anis, Djalel, Saadi, Sandrine pour la bonne ambiance de travail qui règne à l'INRIA.

Je termine par un grand remerciement à mes parents auxquels je dédie mon mémoire de thèse.

Solveur numérique pour les systèmes d'equations algébro-différentiels hybrides

Résumé

Les systèmes hybrides sont des systèmes composés de sous-systèmes temps-discret et de sous-systèmes temps-continu. Les travaux de cette thèse sont concentrés sur l'outil Scicos qui est un logiciel de simulation des systèmes hybrides. Une nouvelle extension de Scicos permet de modéliser des systèmes physiques en utilisant des composants. Afin de décrire les modèles des composants on a choisi le langage Modelica et pour faire la simulation quelques dispositifs ont été rajoutés à Scicos. Du point de vu du simulateur de Scicos, la différence principale, quand on modélise un système basé sur des composants, est que l'on obtient plus souvent un système algébro-différentiel. Pour résoudre numériquement ce type de systèmes d'équations, le solveur numérique Daskr, a été intégré dans Scicos. Mais la simulation n'est pas seulement le problème de l'interface du solveur et du simulateur. D'autres problèmes doivent être réglés afin d'obtenir des résultats satisfaisants. Dans cette thèse les solveurs et le simulateur de Scicos ont été modifiés ou développés pour mieux gérer les systèmes hybrides implicites.

Mot-clés: Simulation, Modélisation, Système dynamiques hybrides, Scilab, Scicos, Modelica, Solveur Numérique

The numerical solver for the simulation of the hybrid dynamical systems

Summary

Hybrid systems are characterized by the co-existence of continuous-time dynamics and discrete-time dynamics. In this thesis, we focus on Scicos as a hybrid systems modeler and simulation tool. A new extension of Scicos allows the natural modeling of physical systems using models of physical components or implicit blocks. To extend the capacity of Scicos to allow component based modeling, we adopted the Modelica language and to simulate the models containing the components some new features have been added to the Scicos simulator. From the Scicos simulator's viewpoint, the main difference when using component based modeling is that the resulting global system is very often given as a set of Differential-Algebraic Equations (DAE). To solve such systems, the DAE numerical solver Daskr has been included in Scicos. But simulation is not just the problem of linking the solver to the simulator; there are other problems that should be coped with to achieve a good simulation result. In this thesis, the development of the Scicos simulator to use the numerical solver efficiently and the modifications made in the numerical solvers for better handling of hybrid systems is presented.

Key words: Simulation, Modeling, Hybrid dynamical systemes, Scilab, Scicos, Modelica, Numerical solver

Contents

List of Figures

Abstract

A new research domain in the field of simulation of physical systems deals extensively with the new class of systems called hybrid systems. Hybrid systems are characterized by the co-existence of continuous-time dynamics and discrete-time dynamics. The simulation of this new class of systems, obtained by the combination of the two types of dynamics, requires new methodologies and approaches. In this thesis we focus on Scicos as a hybrid systems modeler and simulation tool. In particular, the Scicos formalism and its hybrid aspect will be discussed.

The continuous-time and discrete-time dynamics of hybrid systems have such significant interaction that they cannot be decoupled and must be analyzed simultaneously. So to have an efficient simulation, the discrete-time part should be developed by taking into account the properties of continuous-time part and vice versa. Simulation of the continuous-time part needs a numerical solver so it would be unrealistic to imagine that a simulator can be developed independent of the properties of the numerical solver. Then it is important to study these properties and identify precisely what properties are important and must be taken into account in the development of the formalism and the implementation of the modeler and the simulator. The reverse is also true, most numerical solvers are designed to integrate a system without any discontinuity in the variables and their derivatives. A hybrid system, on the other hand, consists of a piecewise continuous-time system. To detect and handle the discontinuities efficiently, the solver must be adapted to the hybrid nature of the system. In this thesis, the development of the Scicos simulator to use the numerical solver efficiently and the modifications made in the numerical solvers for better handling of discontinuities will be discussed.

Before simulation, a hybrid system should be modeled. Recently the Modelica language has been introduced to describe and model hybrid dynamical systems. Scicos has adopted it. This thesis will show how this language is used in Scicos and how the obtained model is simulated.

The thesis then describes some real hybrid models used in an electric power plant. The case studies presented here include, among others, a thermo-hydraulic system, and a heat transfer system. The thesis concludes with a brief overview of ongoing work and possible future research directions.

Chapter 1

Introduction

Hybrid systems are systems with mixed discrete-event/continuous-time subsystems. At discrete time instants some actions take place and between two consecutive discrete actions, the continuous-time subsystem evolves as a function of time. Typically, the continuous-time subsystem is modeled by differential equations, while the discrete-time events are modeled by finite/infinite state machines. Hybrid systems are everywhere. As an example, consider a water tank with a tap on the top. When the tap is open and water is flowing, the water level in the tank is a continuous function of time. The tap is closed when the tank reaches its full capacity. The actions of opening and closing the tap occur at discrete-times. Such a system that has both continuous-time and discrete-time behavior is an example of a hybrid system. There are many reasons for using hybrid system techniques in modeling and simulation of physical systems. The overall motivation for hybrid methods is significant interaction between the continuous-time and the discrete-time parts of a system, as can be the case in the discrete-time planning of continuous-time processes. Hybrid system theory also provides a convenient framework for modeling engineering systems with multiple time scales, where fast dynamics can be abstracted away and be treated as discrete-time changes affecting slower dynamics. The application areas of hybrid systems today range from process control, robotics, flexible manufacturing, avionics and automated highway systems [**?**, **?**, **?**].

The complexity of hybrid models can grow enormously. This is because purely continuous-time and purely discrete-time systems can already be rather complex and in hybrid systems these two world-views are present in a combined way. To predict the behavior of complex hybrid systems computer simulation is necessary. Nowadays we cannot imagine design and manufacturing of any new advanced technological system before its complete modeling and simulation. Obtaining a better and faster result strongly depends on the ability and accuracy of modeling and simulation of that system. In fact, to avoid the high cost of prototyping, the use of computer modeling and simulation is inevitable. A simulator is a collection of hardware and software systems which are used to show the behavior of phenomenon or to analyze and verify theoretical models. Modeling and simulation of a large and complex physical system consists in cutting up the system into several subsystems and then connecting them. Each subsystem has a physical behavior that is described by mathematical equations. Hybrid systems are in general difficult to simulate due to the non-smooth characteristics of the state evolution. There are several hybrid simulation softwares, such as Scicos, SystemBuild, Simulink, Shift, Dymola, etc. In this thesis we focus on Scicos.

Scicos is a software package for modeling and simulation of hybrid dynamical systems.

More specifically, Scicos is intended to be a simulation environment in which both continuous-time systems and discrete-time systems co-exist. Unlike many other existing hybrid system simulation software, Scicos has not been constructed by the extension of a continuous-time simulator or of a discrete-time simulator; Scicos has been developed based on a formalism that considers both aspects from the beginning. Scicos includes a graphical editor which can be used to build complex models by interconnecting blocks which represent either predefined basic functions available in Scicos libraries (palettes), or user defined functions. A large class of hybrid systems can be modeled and simulated this way.

A new extension of Scicos allows the natural modeling of physical systems using models of physical components or implicit blocks. In designing a model with components, there is no causality and components are connected to each other via their ports that are not labeled, a priori, as inputs and outputs. The component imposes a dynamical constraint on the values on its ports, contrary to regular blocks which compute explicitly their outputs as functions of their inputs. Physical components can be more naturally modeled in this way and not as a system with input and outputs, simply because physical laws are expressed in terms of mathematical equations not as mathematical assignments. To extend the capacity of Scicos to allow component based modeling, we needed a language for describing the algebraic-differential constraints on input/outputs of the components. We found the Modelica language to be an excellent choice.

Modelica is an object-oriented modeling language for dynamical systems. Modelica is primarily a modeling language, sometimes called hardware description language, for specifying mathematical models of physical systems, in particular for the purpose of computer simulation of hybrid systems. Modelica is a language that is based on equations instead of assignment statements. This makes it particularly useful for the purpose of component based modeling. This is not a surprise since Modelica is developed exactly for such applications.

To simulate component level models in Scicos, some new features have been added to Scicos. From the Scicos simulator's viewpoint, the main difference when using component based modeling is that the resulting global system is very often given as a set of Differential-Algebraic Equations (DAE). To solve such systems, the DAE numerical solver DASKR has been included in Scicos. DASKR was chosen both because of its ability to solve many DAEs and because of the root finding option which is important for hybrid system simulation. But simulation is not just the problem of linking the solver to the simulator. There are other problems that should be coped with to achieve a good simulation result. In a hybrid simulator, the numerical solver management should be performed automatically. The solver management may be a complicated task, because of the interaction between the continuous-time and the discrete-time dynamics. The discrete part can affect the system equations of the continuous part and the continuous part can generate an event affecting the discrete part.

The contribution of this thesis is in numerical simulation of component based models that are in fact hybrid dynamical systems. In this thesis, the numerical solvers and the simulator of Scicos were modified or developed to better handling the implicit hybrid systems. System level modeling in Scicos had already been available since the beginning of the this project. During this project, the Scicos semantics has been extended with component level modeling. The extension consists of using the Modelica language to describe the model of components, then a C code is automatically generated to produce the input/output behavior of the existing components of the model. Complexity of the new model brings in many difficulties in the simulation of models. These difficulties show up both in the simulator level and in the numerical solver level. To integrate a hybrid system efficiently, some modifications were made in the numerical solvers of Scicos. For example, the zero-crossing routines of the solvers were modified in order to indicate the direction of crossings. In addition, the numerical solver should receive a continuous smooth system of differential equations. But the main characteristic of a hybrid system is the fact the system may not be continuous at all. Thus simulator's task is to provide the solver a piecewise continuous smooth system.

1.1 Thesis Outline

In the chapter that follows, modeling and simulation of hybrid dynamical system will be introduced. Concepts, and simulation of hybrid systems and numerical solvers that are used in simulation, in particular DASKR, will be presented. Also, at the end, an overview of hybrid simulation software will be given.

In chapter 3, Scicos as a software tool for modeling and simulation of hybrid systems will be introduced. A Scicos block as the basic block of modeling of hybrid systems will be explained. Some examples of continuous-time, discrete-time, and hybrid systems will be given. Then, the internal architecture of an ordinary Scicos block will be studied in detail. At last, it will be explained how an interface between AMESim and Scicos softwares can be established. In chapter 4, the shortcomings of the current numerical solvers used in Scicos will be explained. Then the modifications that have been made as part of this research to adapt them to integration of hybrid system will be discussed.

In chapter 5, Scicos architecture will be the main subject. First, the Scicos compiler, then the Scicos simulator will be discussed. The difficulties that we came across in the simulation of hybrid systems will be studied in detail. Also, the whole Scicos simulator flowchart will be given.

In chapter 6, the implementation of a new extension of Scicos, the component level modeling, will be presented. System level and component level modeling approaches and their advantages and disadvantages will be discussed. Some numerical difficulties that were confronted and the implemention of the analytical Jacobian for these systems will be presented.

In chapter 7, some modeling case studies for component level modeling will be given. In particular, a thermo-hydraulic system, which has been modeled and simulated in Scicos, will be explained.

The thesis will conclude in chapter 8 with a discussion of the contributions made and some suggestions for further research.

1.2 Introduction (en français)

Les systèmes hybrides sont des systèmes composés de sous-systèmes temps-discret et de sous-systèmes temps-continu. Aux instants discrets plusieurs actions peuvent avoir lieu au même temps que l'activation temps-continu, en effet entre deux événements discrets le sous-système continu évolue en fonction du temps continu. Les systèmes temps-continus sont modélisés selon leurs nature soit par des systèmes différentiels ordinaires, soit par des systèmes algébro-différentiels, tandis que les systèmes temps-discrets sont modélisés par des systèmes machine à état fini ou infini. Les systèmes hybrides se retrouvent partout, par exemple, considérez un réservoir avec un robinet. Quand le robinet est ouvert le réservoir se remplit, le niveau d'eau est une fonction continue de temps. Quand le niveau d'eau arrive à certain seuil le robinet se ferme. L'action d'ouverture et de fermeture du robinet s'exécute dans un instant discret. Ce type de système qui a des comportements discrets et continus est un exemple de système hybride. La portée d'application des systèmes hybrides est très variée, comme par exemple, le contrôle des procédés, la robotique, l'aviation et les routes automatisées. Il y a beaucoup de raisons pour utiliser les systèmes hybrides dans la modélisation et la simulation de systèmes physiques. La motivation principale pour se servir de la méthode hybride est l'interaction signifiante qui existe entre la partie continue et la partie discrète. La théorie des systèmes hybrides fournit également une méthode pour modéliser des systèmes avec plusieurs échelles de temps, par exemple dans un système qui contient deux dynamiques lente et très rapide, à la place de la dynamique très rapide, on peut utiliser un changement d'état discret. Un tel exemple de ses systèmes, est de considérer l'horloge d'un ordinateur qui change continuellement, mais on le considère comme une horloge discrète.

Pour connaître le comportement d'un système temps-continu ou temps-discret, il faut résoudre les systèmes d'équations continus ou discrets du modèle. Si on modélise la partie continue en utilisant des systèmes algébro-différentiels et la partie discrète en utilisant des systèmes à état fini/infini et des événements, on pourra représenter une large classe des phénomènes physiques. La complexité des systèmes hybrides peut être augmentée, c'est parce que chaque partie continue ou discrète peut être très complexe ce qui rend le modèle hybride très compliqué. Alors, pour connaître le comportement d'un système hybride, une simulation doit être effectuée. Aujourd'hui, Il est impossible d'imaginer la conception et la réalisation des systèmes hautes technologies avant les modélisations et les simulations complètes du système. Pour obtenir un meilleur résultat dans un délai plus court, dans toutes les étapes de conception et de réalisation, il est indispensable d'utiliser des modélisations et des simulations bien précises. Une simulation est une collection de matériels et de logiciels qui pourrait servir à montrer le comportement d'un phénomène à analyser et à vérifier théoriquement, par exemple d'un model qui peut être très difficile ou impossible à réaliser dans le monde réel. La modélisation et la simulation d'un système physique compliqué consiste à de séparer le système en plusieurs sous-systèmes et ensuite les relier par des connecteurs. Chaque sous-système a un comportement physique qui se décrit par un ensemble d'équations mathématiques.

La simulation numérique est très importante dans l'analyse et la conception d'un système de commande hybride, parce que la complexité de ce genre de système limite l'application des méthodes analytiques. En général, il n'est pas facile de simuler numériquement un système hybride à cause des caractéristiques discontinues des systèmes hybrides. Il existe plusieurs logiciels de simulation des systèmes hybrides, comme par exemple Scicos, SystemBuild, Simulink, Shift, Ptolemey, etc. Les travaux de cette thèse sont concentrés sur l'outil Scilab/Scicos. Scicos (www.scicos.org) est une boite à outils du logiciel libre de calcul scientifique Scilab (www.scilab.org), dédiée à la modélisation et la simulation des systèmes dynamiques hybrides. Ces systèmes, représentés sous forme de schémas blocs, peuvent être potentiellement constitués d'éléments avec des fonctionnements de nature différente : continu, discret, événementiel; réalisant ainsi des systèmes hybrides.

Nous présentons dans ce mémoire une méthodologie pour la modélisation des systèmes dynamiques hybrides. Le principal objectif de cette modélisation est la simulation du système complet, c'est à dire constitué de tout l'environnement avec son système de commande, de façon à répondre aux besoins du monde industriel comme par exemple la validation des lois de commande ou la simulation d'un système thermo-hydraulique d'une centrale nucléaire. L'idée principale est le développement de Scicos dans l'objectif d'obtenir un simulateur bien défini qui permettra la modélisation et la simulation d'une large classe de systèmes dynamiques hybrides de manière plus naturelle et plus proche de la modélisation physique du système.

Scicos est composé d'un éditeur graphique qui sert à construire des modèles en reliant des blocs. Un bloc peut être déjà défini comme une fonction élémentaire disponible dans des palettes de Scicos ou défini par l'utilisateur. Ainsi, une grande classe des systèmes hybrides peut être modélisée et ensuite simulée dans Scicos.

Une nouvelle extension de Scicos permet de modéliser les systèmes physiques plus naturellement en utilisant des composants ou des blocs implicites. Dans la construction d'un modèle avec des composants, il n'y a pas de causalité et les composants se connectent par leur ports, qui ne sont pas libellés, à priori, comme des entrées ou des sorties. Contrairement aux blocs réguliers qui calculent explicitement leurs sorties en fonction de leurs entrées, les composants ou les blocs implicites imposent des contraintes dynamiques sur les valeurs de ses ports. Ces contraintes sont des lois physiques qui sont exprimées par des équations mathématiques et non pas avec des affectations mathématiques, ce qui est le cas avec les blocs explicites. Afin de décrire des contraintes algébro-différentielles sur les entrées/sorties des composants et d'étendre la portée de Scicos pour modéliser des systèmes en utilisant des composants, on a eu besoin d'un langage. On a choisi le langage Modelica. Modelica est un langage orienté objet pour modéliser des systèmes dynamiques. C'est un langage de modélisation pour spécifier des modèles mathématiques de systèmes physiques, en particulier, pour la simulation. Modelica est basé sur des équations et non pas des affectations mathématiques, cela est très utile pour la modélisation au niveau des composants.

Afin de pouvoir modéliser les systèmes en utilisant des blocs implicites, quelques dispositifs ont été rajoutés à Scicos. Du point de vu du simulateur de Scicos, la différence principale, quand on modélise un système basé sur des composants, est que l'on obtient souvent un système algébro-différentiel. Pour résoudre numériquement ce type de systèmes d'équations, le solveur numérique DASKR, a été intégré dans Scicos. DASKR a été choisi, d'abord, pour ses capacité à résoudre une large classe de DAE, ensuite pour l'option de détection des traversées de zéro qui est importante pour la gestion des discontinuités. Mais la simulation n'est pas seulement le problème de l'interface du solveur et du simulateur. D'autres problèmes doivent être réglés afin d'obtenir des résultats satisfaisants. Dans un contexte hybride, le contrôle du solveur doit se faire automatiquement et sans que l'utilisateur s'en aperçoive. Ce contrôle est assez compliqué à cause de l'interaction qui existe entre la dynamique continue et le reste du système. D'un côté la partie discrète peut affecter les systèmes d'équations de la partie continue, de l'autre côté, la partie continue peut générer un évènement affectant la partie discrète.

L'objet de cette thèse réside dans la simulation numérique des modèles basés sur des composants, qui sont en effet des systèmes hybrides. Dans cette thèse les solveurs et le simulateur de Scicos ont été modifiés ou développés pour mieux gérer les systèmes hybrides implicites. La modélisation basée sur les blocs explicites était déjà utilisable quand cette thèse a commencé. Ce document présente la façon dont la sémantique de Scicos a été étendue pour permettre la modélisation au niveau des composants. Cette extension consiste à utiliser le langage Modelica pour décrire les modèles des composants, ensuite générer un code C pour réaliser le comportement entrée sortie de système composé des blocs implicites. La complexité de ces nouveaux modèles entraîne beaucoup de difficultés au niveau de la simulation. Ces difficultés se manifestent au niveau du simulateur ainsi qu'au niveau du solveur numérique. Afin de simuler un système hybride efficacement quelques modifications ont été effectuées dans le solveur numérique de Scicos. Par exemple, les routines de traversée de zéro du solveur ont été modifiées pour indiquer la direction des traversées de zéro. De plus, le solveur numérique doit recevoir un système d'équations différentielles bien lisse. Mais la principale caractéristique d'un système hybride est qu'il n'est pas nécessairement continu et lisse, il peut être discontinu. La tache du simulateur est donc de fournir au solveur un système continu et lisse par morceau.

Ce document s'achève avec quelques exemples physiques, en particulier, une application basée sur équation de la chaleur et un système de refroidissement de centrale de production d'électricité. Les deux applications ont été modélisées et simulées dans Scicos.

Chapter 2

Review and Theoretical Background

2.1 Physical System Modeling and Simulation

A physical system is a set of interacting components, or entities, that generally work together to achieve some objectives. These systems are large and complex, and would be difficult and expensive to experiment with directly. A model is an abstract and simplified representation of a system that represents the most important system components, and the way in which they interact. Simulation is using the model to predict the behavior of the system without directly experimenting with the system. Simulation has been used to analyze a very large variety of systems. For evaluating the performance of manufacturing systems, simulation has come to be the dominant methodology, and special modeling tools and simulation software have been developed for these systems. Simulation is not specific to any particular application area, but can be applied to any system that can be modeled using the modeling concepts.

The term *simulation* has been used to mean quite a number of things. Usually it refers to a realization of a representation of some larger, more complex activity. For example, engineers build simulations of physical systems such as a ship's flow through water. Any simulations use a model to represent the behavior of a system that may or may not exist and that is generally much larger, costlier and more complex than the model. The model may be physical, as in the cases of an aircraft simulator, or it may just be represented as a computer program. In all cases, the key idea is that the simulation is an alternative realization that approximates the system, and in all cases the purpose of the simulation is to analyze and understand the system's behavior under various alternative actions or decisions.

In order to simulate a physical system, modeling is the first step. There are several other steps that should be done to achieve a reliable simulation result. Typical phases in a simulation study are shown in Fig. 2.1.1. They are data collection, mathematical modeling, model verification, model validation, and finally simulation to gain the results. The main concern of this thesis is the modeling and simulation approaches.

2.2 System Modeling

The system state is a collection of variables and possibly other necessary information to run over time. The major task in simulating a system is to come up with a model that captures the

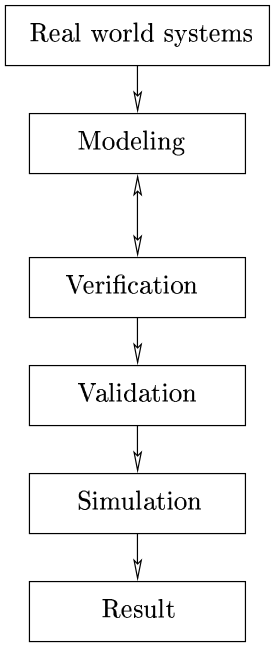


Figure 2.1.1: Simulation steps.

behavior of the system. Thus, the modeler must first select a representation for the system states. In some simple models, the system state could be a collection of variables such as temperature, speed, etc. More realistic models, however, need a much richer representation for the system state in order to capture the complexity of the model. Maybe the first important factor in modeling is the independent variable selection. The modeler has two choices about the independent variable (time); it can advance either continuously, or it jumps. In other words, time can be represented as a continuous-time variable or as a discrete-time variable. This choice makes the system continuous-time or discrete-time, respectively.

• If time advances continuously, the system states change continuously in time as well. In this modeling approach, time is treated as a continuous-time variable and as a result, system changes are expressed in terms of a set of differential equations involving the system state variables. In this case, the simulation program numerically integrates the differential equations to compute the solution of the system state. A typical plot of a continuous-time state in a continuous-time simulation is depicted in Fig. 2.2.1.

• If time advances discretely, the system states change only at discrete-time points, called *event times*. When an event occurs, states may change abruptly (sudden changes) with respect to time. This type of model is called a discrete-time or discrete-event system model. This approach might be used to model the processing of parts in a factory. In this model type, the state does not change between two events. The graphic in Fig. 2.2.2 shows the plot of a state of a discrete-time system with respect to time in a discrete-time simulation.

• It is possible to have a combined discrete-time and continuous-time system model in which the values of some variables are controlled by differential equations and for others the values are changed at the moments that events occur. For example, in a steel mill, an event occurs to cause the steel to begin heating, but the temperature of the steel is determined by a set of differential equations that depend upon the amount of power applied, the starting

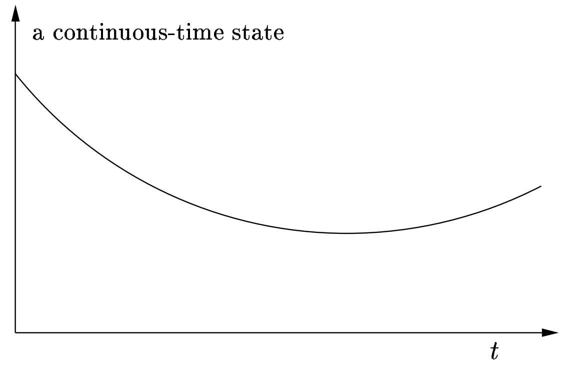


Figure 2.2.1: Continuous-time system output example.

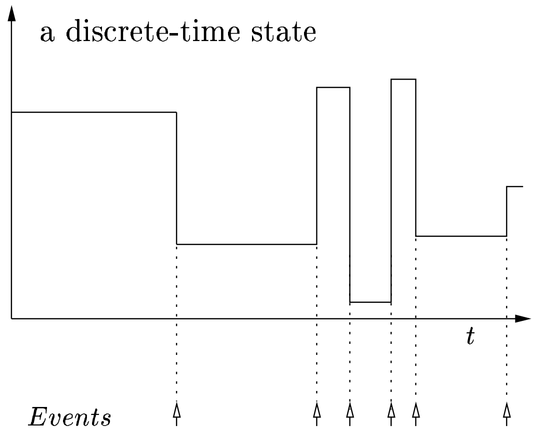


Figure 2.2.2: Discrete-time system output example.

temperature and some random variables. The event involving pouring the steel cannot occur until the steel exceeds a specified temperature. Thus, the variables that change continuously over time and those that change only in events are interdependent. We call this category of systems **hybrid systems**. The graphic in Fig. 2.2.3 shows a possible plot of a state with respect to time in a hybrid simulation.

2.3 Hybrid Modeling

A hybrid system is a dynamical system that cannot be represented and analyzed with sufficient precision either by the method of the continuous-time systems theory or by the method of the discrete-time systems theory. It is modeled using continuous-time components as well as discrete-time components. As an example consider a bouncing ball. It is a hybrid system, since its state variables (position and speed) vary continuously according to Newton's laws when falling, but have a discrete-time change (speed is reversed) when entering in collision with the ground. Note that the discrete-time change is due by the way we model the collision (for simplicity); in reality, everything is continuous-time for this example.

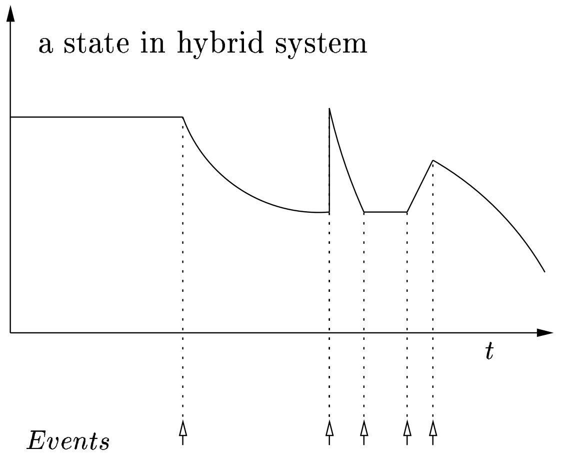


Figure 2.2.3: Combined system output example.

The state jumps and the discontinuities are the basic phenomena that cannot be represented and analyzed by methods elaborated either in continuous-time or in discrete-time system theory. Hence, a system has to be considered as a hybrid system if both the continuous-time movement and the state jumps are important to be taken into account. In the following sections, the specification of continuous-time and discrete-time sub-systems, and their interaction in a hybrid system will be discussed in detail.

2.3.1 Discrete-time Systems

From a classical mechanical point of view, everything is continuous-time in nature: for example, the collision of the ball with the ground is not an instantaneous event, but in reality, is the deformation of the ball (like a spring) which will have the effect at the end of having inverted its speed. This continuous-time world is no more true in Quantum Mechanics, but this point is not relevant at our level of modeling since Quantum Mechanics is rarely present in engineering situations. There are several reasons to use discrete-time modeling and simulation:

First, The real world is very complicated. To deal with this complexity, we have to make approximations and abstract out the non-important phenomena. The bouncing ball is a good example. We usually do not know how the ball will deform itself when colliding with the ground and we usually do not care anyway. A good first approximation is to assume that the collision is not elastic, i.e., some kinetic energy is transformed in thermal energy due to deformation of the ball. This approximation business is closely related to the time scale we are interested in. For example, if we want to study effects on the ball which occur in a microsecond period, then we will have to study in detail its collision with the ground and not consider it as instantaneous. Fig. 2.3.1 shows the effect of the time scale on the continuity of the downward velocity of ball. Depending on the time scale under study, a phenomenon will appear continuous-time or discrete-time.

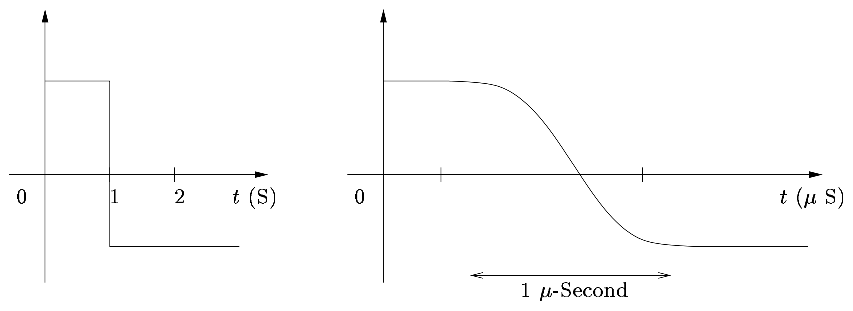


Figure 2.3.1: Approximation due to time scale.

Furthermore, discrete-time behavior happens often in control systems, where switches are usually used (electrical switch, for example). A canonical example of a control system used in the modeling world is bang-bang control. This can be used, for example, to keep the temperature of a room inside a predefined range with a simple thermostat. When the temperature exceeds a high threshold ($Tmax$), the control mechanism stops heating; when the temperature falls below a lower threshold ($Tmin$), it starts heating. So the heating undergoes a discontinuous change in a bang-bang model.

Finally, discrete-time systems are simulated much more efficiently and much faster than continuous-time ones. For these purposes we often use discrete-time systems to model physical systems. A discrete-time system is mathematically modeled by a set of difference equations expressed as:

$$\begin{aligned} x(k+1) &= f(x(k), u(k), t(k)) \\ y(k+1) &= h(x(k), u(k), t(k)), \end{aligned} \tag{2.3.1}$$

where $x(k)$ is the discrete-time state vector, $t(k)$ is the time, k is the index corresponding to the time point $t(k+1)$ at which the state takes on the new value $x(k+1)$, $y(k+1)$ is the output vector, and $u(k)$ is the input vector. The time $t(k)$ is usually, but not always, uniformly spaced ($t(k) = t(0) + k \cdot T_s$) where T_s is the *sampling time.* In any case we assume that these times points are known in advance. This time points at which the system states are changed are referred to as *events.*

Events

In discrete-time systems, the connections between two system parts is carried out with signals that consist of events placed on a time axes. An event is characterized by a source and an activation time. This is different from continuous-time signals, where the signal is continuous in time. In the discrete-time models, the set of events are located discretely on the time axis. The components in a system respond to input events and produce output events (instantaneously or in the future) that may trigger other parts of the system. As an example of a discrete-time system, the arrival of patient to a hospital can be considered as an event that activates the emergency ward, and then a surgery is called (a new event).

Discrete-time Modeling and Simulation

There are three basic ways to implement discrete-time modeling in practice: the *event scheduling* approach, the *activity scanning* approach and the *process interaction* approach [?, ?, ?, ?]. While each of these methods has their own special way of looking at system dynamics, every discrete-time system can be modeled using any of the three methods. For the sake of brevity, we will only explain the event scheduling approach that will be used in this thesis.

In the event scheduling approach an event is the only thing that changes the system states. The essential idea of event scheduling is to move along the time scale until an event occurs, and then depending on the event, modify the system state and possibly schedule new events (which can occur as a result of the arriving event). Each event specification then consists of one determined event and a collection of events that may be caused by it. In the event-scheduling approach, simulation is executed as a sequence of events ordered in time; but no time elapses within an event. Simulated time (the simulation clock) jumps from one event time to the next event time. At each event, we need to change the state and possibly the list of future events. The simulation can end either at a fixed time (e.g., 10 seconds) or event count (e.g., 1000 job completions) or a special event (e.g., user stop request).

A typical discrete-time simulator operates by maintaining an event queue, in which the events are sorted by their activation-time. During the simulation, the output events from all model parts will be fed into the queue. At each iteration of the execution, the event with the smallest activation-time is removed from the queue, and the model parts concerning to this event are activated. The activation time of the event that is just removed from the queue is defined as the current time of the simulation.

Managing the *simultaneous events* (events with the same activation-time) and *synchronous events* (events having the same origin) is a key issue in designing a discrete-time simulator. At any event arrival, all the consequent simultaneous or synchronous events should be treated or programed. In this thesis, we do not enter into the details of discrete-time system analysis. For a complete analysis of the discrete-time model, please see [?, ?, ?, ?].

2.3.2 Continuous-Time System Modeling

In contrast to discrete-time systems, the signal flow in all parts of a continuous-time systems are usually continuous-time signals. In order to model the continuous-time part of a hybrid system or a continuous-time system, we use the differential equations. In this section we are going to explain how the differential equation of a model is obtained then different types of differential equations will be introduced.

Modeling a physical system is to give a description of the system and its states (physical properties) and the state transition mechanism (physical laws). Simulating a physical system, on the other hand, is to describe its behavior when specific inputs are given to the system (like initial conditions, driving forces, etc.) (see [?]). Modeling a system consists of three main Steps:

1. Determine the goal of the model. This gives the framework of the subsequent modeling and simulation and influences the assumptions to take.

2. Determine the state variables (such as the temperature of the system, the position and velocity of its constituents, etc.)

3. Determine the laws and interactions in the system which describe its evolution in time and the constraints relating its state variables.

4. Then determine the parameters of the system which should remain constant after starting the simulation such as mass, lengths, etc.

Consider, for instance, the simple ideal pendulum shown in Fig. 2.3.2 with a "massless" string of length l and a bob of mass m.

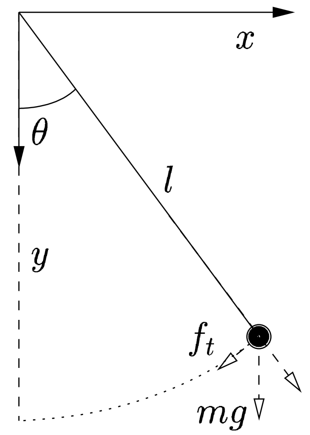


Figure 2.3.2: Pendulum.

The primary forces acting on the bob are the gravitational force that makes it move in the first place and the force exerted by the string to keep it moving along a circular path. In addition, there may be a damping force from friction at the pivot or air resistance or both that we do not consider at first in this example. We construct a model to describe how the angle of the pendulum varies as a function of time t. Let $s(t)$ be the distance along the arc from the lowest point to the position of the bob at time t, with displacement to the right considered positive. Let $\theta(t)$ be the corresponding angle with respect to the vertical. The gravitational force is directed downward and has magnitude mg, where g is the gravitational acceleration constant. Thus, the force acting in the tangential direction is $f_t = -mg\sin(\theta)$ (The negative sign is because this force is in the negative direction when θ is positive and vice versa). Since this force is mass times acceleration, it follows that

$$\frac{d^2s}{dt^2} = -g\sin(\theta).$$

Now s and θ are related as arc length and central angle in a circle of radius l, i.e., $s = l\theta$. Thus, the second derivative of s is l times the second derivative of θ. That brings us to our undamped model differential equation with a single dependent variable, the angular displacement θ:

$$\frac{d^2\theta}{dt^2} = -\frac{g\sin(\theta)}{l}.$$

Next, to make the model more precise, we can introduce damping to the model. We make the simplest possible assumption about the damping force, that it is proportional to velocity.

Since arc length and central angle are themselves proportional (with proportionality constant l), it makes no difference whether we use linear or angular velocity. Having selected θ as our dependent variable, we will represent the damping as proportional to angular velocity, say, $-b(\frac{d\theta}{dt})$. The negative sign is because the damping force has to be opposite the direction of motion. When we include this term in the model, our equation becomes

$$\frac{d^2\theta}{dt^2} + \frac{b}{m}\frac{d\theta}{dt} + \frac{g\sin(\theta)}{l} = 0,$$

and with $\alpha = \dfrac{d\theta}{dt}$ we have

$$\begin{aligned}\dot{\theta} &= \alpha \\ \dot{\alpha} &= -\frac{b}{m}\alpha - \frac{g\sin(\theta)}{l},\end{aligned}$$

which is, in fact, a differential equation set in the form of ordinary differential equation.

Ordinary Differential Equation

Modeling most dynamical systems ends up with a finite number of coupled ordinary differential equations:

$$\begin{aligned}\dot{x}_1 &= f_1(x_1, ..., x_n, u_1, ..., u_p, t) \\ \dot{x}_2 &= f_2(x_1, ..., x_n, u_1, ..., u_p, t) \\ &\vdots \\ \dot{x}_n &= f_n(x_1, ..., x_n, u_1, ..., u_p, t),\end{aligned}$$

where $\dot{x}_i$ denotes the derivative of x_i with respect to the independent variable t, and u_i are specified input variables. We call the variable x_1, x_2, ...,x_n the state variables, that represent the memory that the dynamical system has of its past. Vector notation is usually used to write the above equation in a compact form. That is,

(2.3.2) $$\dot{x} = f(x, u, t).$$

Then (2.3.2) is called the state equation, and x and u are referred to as the *state variables* and the *input* respectively. Sometimes, another equation is associated with (2.3.2), i.e.,

(2.3.3) $$y = h(x, u, t),$$

which defines an *output* vector that comprises variables of particular interest in the analysis of the dynamical system, like variables which can be physically measured or variables which are required to behave in a specified manner. The output equation (2.3.3) with the equation (2.3.2) together are referred to as the *state-space* model. Often, the state equation is used without explicit presence of an input u, that is, the so-called unforced state equation

(2.3.4) $$\dot{x} = f(x, t).$$

Working with an unforced state equation does not necessarily means that the input to the system is zero. It could be that the input has been specified as a given function of time,

$u = g(t)$, a given feedback function $u = g(x)$, or both, $u = g(t, x)$. Substitution of $u = g$ in (2.3.2) eliminates u and yields an unforced state equation.

Although equation (2.3.2) is a general equation, there are certain phenomena that cannot be described with it. In fact, the modeling a physical system does not always end up with an ODE. There are two other important differential equation forms, i.e., *partial differential equations* and *differential Algebraic Equations* that will be discussed in the following sections.

Partial Differential Equation

If in modeling a physical system we have a single independent variable, we may end up with an ODE, i.e., (2.3.2). But if we have two or more independent variables, we will obtain a *Partial Differential Equation* (PDE). A PDE is an equation relating a function of two or more independent variables and its partial derivatives, such as the following equation

$$u_{xx} + u_{yy} + u_{zz} = f,$$

where u is a functions of x, y and z. For example, the distribution of temperature in an iron bar, heated at its both ends with different heating rates, is a function of time and distance from the end points. PDEs are used everywhere in science, as they describe phenomena such as fluid flow, gravitational fields, and electromagnetic fields. They are important in fields such as aircraft simulation, computer graphics, and weather prediction. In this thesis PDE's are not addressed, so interested readers are referred to e.g., [**?**, **?**].

Differential Algebraic Equations

This is an explicit ODE

$$\dot{x} = f(x, t).$$

A more general form is an implicit ODE

$$0 = F(\dot{x}, x, t),$$

where the $\frac{\partial F}{\partial \dot{x}}$ matrix is assumed to be nonsingular for all argument values in an appropriate domain. In principal, it is then often possible to solve for $\dot{x}$ in terms of t and x, obtaining the explicit ODE form. However, this transformation may not always be numerically easy or cheap to realize, so in many cases it is preferred to leave the ODE in the implicit form. This form of differential equation appears in modeling of a physical system involving some constraints such as conservation laws (mass and energy balance), constitutive equations (equations of state, pressure drops, heat transfer...), and design constraints (desired operations...). In fact, modeling a system with constraints usually leads to a sort of ODE's and some algebraic equations, i.e.,

$$\dot{x} = f(x, y, t) \tag{2.3.5}$$

$$0 = g(x, y, t), \tag{2.3.6}$$

where (2.3.5) is the differential part and (2.3.6) is the constraints or the algebraic part of the DAE. This equation set is also referred to as *an ODE with constraints.* Here the ODE (2.3.5) for x depends on an additional algebraic variable y, and the solution is forced to satisfy the algebraic constraints (2.3.6). Consider again the pendulum in Fig. 2.3.2, but instead of θ, select the bob's position in Cartesian coordinates as system states. Then Newton's law for x and y coordinates are

$$\begin{aligned} m\ddot{x} &= f_x &&= -\frac{x}{l}f \\ m\ddot{y} &= -mg + f_y &&= -mg - \frac{y}{l}f, \end{aligned}$$

where f_x and f_y are the x and y components of the force and $-mg$ is the additional downward force due to gravity acting on the pendulum. In addition to equations for Newton's law, there is a constraint that the position must sum to the length of the pendulum:

$$x^2 + y^2 = l^2.$$

After introducing additional variables $u = \dot{x}$ and $v = \dot{y}$ we have

$$\begin{aligned} \dot{x} &= u \\ \dot{u} &= -\frac{x}{ml}f \\ \dot{y} &= v \\ \dot{v} &= -g + \frac{y}{ml}f \\ 0 &= x^2 + y^2 - l^2, \end{aligned}$$

which is composed of four ODEs and an algebraic equation. This is no longer an ODE, it's a *Differential Algebraic Equation* or simply a DAE. In this very simple example of a DAE system, the change of variables $x = l\sin(\theta)$ and $y = l\cos(\theta)$ allows elimination of f by simply multiplying the ODE for x by y and the ODE for y by x and subtraction. This yields the simple ODE

$$\ddot{\theta} = -\frac{g}{l}\sin(\theta),$$

as we saw in section 2.3.2. Such a simple elimination is usually impossible in more general situations to obtain an ODE from a DAE. DAEs arise in a variety of applications, such as constrained mechanical systems, electrical circuits and chemical reaction kinetics. Therefore their analysis and numerical treatment plays an important role in modern modeling and simulation. DAEs present analytical and numerical difficulties that are quite different from those encountered in ODEs [?]. The difference between ODE and DAE is fundamental. In the following sections, we will start with some definitions and then we will explain some characteristics of DAEs.

Semi-Explicit vs Fully Implicit DAE

The (2.3.5-2.3.6) equation set where the ODE and the algebraic part are decomposed is a *semi-explicit* DAE. In this DAE form, equation (2.3.5) is referred to as the *differential equations* and the equation (2.3.6) that is purely algebraic is called the *algebraic equations.*

If we cast (2.3.5-2.3.6) in the form of an implicit ODE for the unknown vector $z = \begin{pmatrix} x \\ y \end{pmatrix}$; however, we have:

$$0 = F(\dot{z}, z, t), \tag{2.3.7}$$

where $\frac{\partial F}{\partial \dot{z}}$ is no longer nonsingular. This system is called a *fully implicit DAE.* Note that if $\frac{\partial F}{\partial \dot{z}}$ is non-singular, then it is possible to formally solve $\dot{z}$ for z in order to obtain an ODE. However, if it is singular, this is no longer possible and the solution z has to satisfy certain algebraic constraints. As an example, this is an fully implicit DAE:

$$\begin{aligned} 0 &= x_1 + \dot{x}_2^2 + sin(x_2 + x_1) \\ 0 &= x_2 - x_1 + cos(\dot{x}_2). \end{aligned}$$

DAE Index

DAEs are characterized by their index. Here, the definition of the differential index is given, as defined by Gear [**?**].

Definition: The *index* of a equation (2.3.7) is m, if m is the smallest number such that the system of differential equations

$$\begin{aligned} F(\dot{x}, x, t) &= 0 \\ \frac{dF(\dot{x}, x, t)}{dt} &= 0 \\ \vdots \;\; &\;\; \vdots \\ \frac{d^{(m)}F(\dot{x}, x, t)}{dt^{(m)}} &= 0, \end{aligned}$$

can be transformed into an ODE by algebraic manipulations. Obviously, the index of an ODE is zero. In general, the higher the index, the greater the numerical difficulty one is going to encounter, when trying to solve the system numerically. Systems with indexes greater than one are particularly difficult to solve. These are called high index DAEs. The first method for solving DAEs was proposed by Gear [**?**, **?**]. This is based on a backward difference formula (BDF method). Several codes have been written to solve DAEs that exploit this method. For example, LSODI [**?**], DASSL, and DASKR [**?**, **?**]. These codes are capable of solving DAEs of index 0 and 1. The direct solution of high index systems, except for some rather specific forms, however, is not possible using classical numerical methods. Some numerical techniques are under development for general DAEs, See works by Kunkel and Mehrmann [**?**, **?**, **?**, **?**] and Campbell [**?**, **?**, **?**, **?**, **?**]. High index problems are mostly solved by first reducing the index to 0 or 1, then the resulting system is solved by available solvers.

Consistent Initial conditions

One issue which is quite different for DAEs compared to ODEs is the specification of initial conditions. For ODEs a set of initial conditions uniquely determines a solution. In other word, we are free to choose all the variables. If the ODE size is N, our degree of freedom in N. One of the key differences between DAEs and ODEs is that in DAEs not all variables can freely be initialized. This problem is far from being trivial even if the DAE is a smooth function. To better understand this issue, consider the following index-2 DAE:

$$\begin{aligned} x_1 &= \sin(t) \\ \dot{x}_1 &= x_2, \end{aligned} \tag{2.3.8}$$

where the exact solution of (2.3.8) is

$$\begin{aligned} x_1 &= \sin(t) \\ x_2 &= \cos(t). \end{aligned}$$

So the initial conditions should be

$$z_0 = \begin{pmatrix} \sin(0) \\ \cos(0) \end{pmatrix}.$$

For this example we do not have any degree of freedom. If integration start from a point other than this initial condition set, say z_1, then the difference between the consistent initial condition set (z_0) and the proposed initial condition (z_1) will introduce a constant value $e_0 = z_0 - z_1$ in the local error in the first step of integration. Since the local error will not vanish but rather approach the value e_0 when the step-size reduces, the error control strategies implemented in solvers will fail to meet the error criteria. If the initial conditions do not satisfy the system equations, in the first step the the numeric solver may fail to converge due to a large error independent of step size [**?**].

The initial values of variables $\dot{x}$, x and y, denoted by $\dot{x}(0)$, $x(0)$, and $y(0)$, must satisfy equation (2.3.7) at time $t = 0$:

$$f(\dot{x}(0), x(0), y(0), 0) = 0.$$

If they do so, then the initial values are consistent. But it is not always enough, a further property of DAEs is that the initial values may have to satisfy additional algebraic constraints. These are hidden constraints that are not explicitly present in the original DAEs. This is particularly true for high index DAEs. These conditions can be revealed by executing the m differentiations in (2.3.8) [**?**]. In this way, however, all equations are differentiated, possibly unnecessarily. Pantelides proposes an algorithm [**?**] to identify the subset of the equations that must be differentiated in order to reveal such hidden constraints. In the preceding example (2.3.8), differentiating the first equation yields

$$x_2 = \cos(t),$$

this means, in addition to constraint $x_1(t) = \sin(t)$, that there is also a *hidden constraint* $x_2(t) = \cos(t)$, which the solution must satisfy at any point t, so the only consistent initial conditions are $x_1(0) = 0$, $x_2(0) = 1$, [**?**].

Differential vs Algebraic Variables

Another important definition in DAE theory is the *differential* and *algebraic variables.* For the semi-explicit index-1 and fully implicit index-1 DAEs, we can distinguish between *differential variables* and *algebraic variables.* Variables whose derivatives appear in DAE are called differential variables and the other variables are algebraic, i.e x in (2.3.5) is differential and y in (2.3.6) is algebraic. The algebraic variables may be less smooth than the differential variables by one derivative (e.g., the algebraic variable may be non-differentiable) [**?**].

The computation of consistent initial conditions of a DAE requires a classification of its state's components. Suppose an index one DAE is in semi-explicit form, that is, it is in the form:

$$\begin{cases} f(\dot{x}, x, y) = 0 \\ g(x, y) = 0, \end{cases} \tag{2.3.9}$$

where $\left|\frac{\partial f}{\partial \dot{x}}\right| \neq 0$ and $\left|\frac{\partial g}{\partial y}\right| \neq 0$. Then the initial x is free and must be supplied and the computation of a consistent initial condition consists of finding the initial values of y and $\dot{x}$. The initial value of $\dot{y}$ is not needed because it does not appear in the Jacobian and is not really needed to compute the estimate for the next value of y since a small step is used. In developing initialization for DASKR it was found that the solution was insensitive to the value of $\dot{y}$ and a value of $\dot{y}(t_0) = 0$ was found to work well [**?**, **?**].

In this case, x and y are called respectively the differential and algebraic variables or states. x is called differential because $\dot{x}$ appears explicitly in the system, similarly y is called algebraic because $\dot{y}$ does not appear explicitly. In the general index one case, the situation is somewhat more complex and the notions of differential and algebraic cannot be used in exactly the same way. Consider for example the following DAE:

$$\begin{cases} \dot{x} + \dot{y} = 0 \\ x = 1. \end{cases}$$

If we apply the above definitions, x and y would both be differential. However, clearly the initial condition of x is not free in contrast to that of y. In this case, there are two options: we can consider both states as differential providing x and y and asking Scicos to compute consistent $\dot{x}$ and $\dot{y}$, or provide y and ask Scicos to find consistent x (which of course in this case is clearly equal to one). In the first option, it is up to the user to make sure that they furnish a consistent (x, y). On the other hand, in the second option, the part of the state that needs to be furnished is not always uniquely defined. Consider for example:

$$\begin{cases} \dot{x} + \dot{y} = 0 \\ x - y = 1. \end{cases}$$

In this example both x and y can be freely initialized but not simultaneously. Note that in all cases, the system equations do not characterize completely $\dot{x}$ and $\dot{y}$. In the above example the only equation characterizing them is $\dot{x} + \dot{y} = 0$. It turns out, however, in this example that the solver does not need them as long as it has a pair of $\dot{x}$ and $\dot{y}$ satisfying the system equations. The reason is that this system is linear in the derivative so that the Jacobian depends only on the value of $\dot{x} + \dot{y}$. After one step we have $x(t_i) - y(t_i) = 1$ for $i = 0, 1$ and we pick up the additional information needed to get unique estimates for $\dot{x}$ and $\dot{y}$. In this thesis, we do not treat these special cases, and we assume that all DAE are in the form of (2.3.9). Hence, variables whose derivative appears in the DAE are called differential variables and the rest are called algebraic variables.

Discontinuity in ODE/DAE

The functions f in (2.3.2), and F in (2.3.7) are allowed to be discontinuous at a countable number of discrete-time points, called the discontinuity points [**?**]. These discontinuity points may be caused by the discontinuity of input signal u, or by an intrinsic property of the ODE/DAE functions. In theory, the solutions at these discontinuous points are not well defined. But the left and right limits at these points are. So instead of solving the ODE/DAE at those points, we would actually try to find the left and right limit. A discontinuity point may be known beforehand, in which case it is called a predictable discontinuity. For example, for a square wave signal source, we can predict its next flip time. This information can be used to control the discretization of time. A discontinuity point can also be unpredictable, which means it is unknown until the time it occurs. An example is a system whose functionality is varied when an input signal crosses a threshold. In the first case, integration can be carried out until the left limit of the discontinuity point, but in the latter case, the discontinuity point should be found and then integrate until its left limit. This means there is more work to do to handle this kind of discontinuity.

2.3.3 Discrete-Time and Continuous-Time System Interaction

Hybrid systems arise from the interaction between continuous-time systems (i.e., systems that can be described by a system of differential equations) and discrete-time systems (i.e., asynchronous systems where the state transitions are initiated by events. In a hybrid system composed of continuous-time and discrete-time sub-systems, each sub-system can interact with the other, see Fig. 2.3.3.

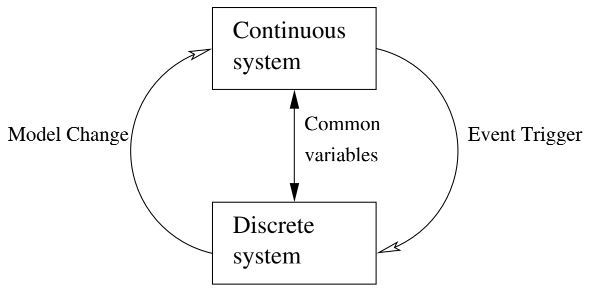


Figure 2.3.3: Continuous-Time and discrete-time parts interaction in a hybrid system.

The discrete-time components influence the continuous-time behavior by changing the values of the continuous-time variables by discrete-time actions. This can be accomplished in the following ways:

- A discrete-time variable used in a differential equation is changed. As a consequence, all the continuous-time variables that depend on this value are changed implicitly. As a result we may have a discontinuity.
- One set of equations is replaced by another set. We assume that the number of states remains the same. In this case we may have a discontinuity as well.

In the same way, a continuous-time sub-system can influence the discrete-time behavior in the following ways:

- The value of a continuous-time variable is used in a discrete-time action.
- A boolean condition that depends on the values of some continuous-time variables becomes true and triggers a discrete-time action.

Discrete-time and continuous-time systems have distinct types of signals. For continuous-time systems, all output signals are continuous-time waveforms, while for discrete-time systems, the activation signals are discrete-time and the output signals are piecewise continuous. When putting together these two domains, the signals at the boundaries must be treated carefully. A Zero-Order Hold (ZOH) converts discrete-time into a continuous waveform to have a value at the time points where no events occur. Zero-order hold's output is a piecewise constant function.

The discrete-time part is activated by events that may come from several sources such as intervention by human operators or from input control or may be programed to be generated autonomously. They are called control switches and autonomous jumps [**?**, **?**]. There are two event types that are more important. They are events that occur at a given time, i.e., a *time-event*, and those who appear when a variable reaches a certain threshold value, i.e., a *state event*. The continuous-time part can make a change in the discrete-time part only if it generates a *state event*.

Time-Events

Time-events occur at a specified future time that is known when the event is scheduled, such as the sampling-time of a discrete-time controller. Time-events are most easy to handle as they may be specified directly before the simulation begins. The integration thereby may be stopped and restarted exactly at the possible discontinuity point. An example is a process where an operator resets a valve opening at a specified time.

Zero-crossing Event

It is possible that a hybrid system have several possible configurations whereby in each configuration the behavior of the system is described by a system of differential equations. Such a system switches from one configuration to another based on a condition. This condition may depend on the input control of the system (such as human operators intervention), or may depend on system states [**?**, **?**]. For instance, in [**?**] several flight modes of a helicopter such as *Hover, Cruise, Acc/AH, Dec/AH, Climb*, and *Descend*, where *AH* and *Acc* stands for "Altitude Hold", and "Acceleration" respectively, have been represented. Each mode represents a different configuration of operation of the helicopter that correspond to different variables and differential equation set in the system dynamic, see Fig. 2.3.4.

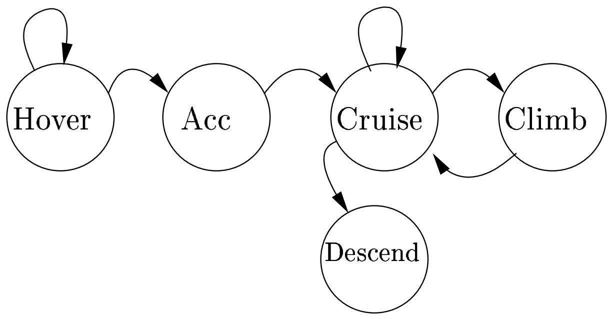


Figure 2.3.4: Helicopter flight mode switches.

In this example, the switching between configurations is done by the pilot, i.e., by input control. There are, however, systems for which switching between configurations is automatic or inherent in the system. Consider, for example, the constrained pendulum of Fig. 2.3.5, where a pin is placed at $(x = 0, y = H)$, to constrain the string's movement. The bob trajectory changes at $\theta = \theta_0$, and another equation set should be used for the trajectory of the bob.

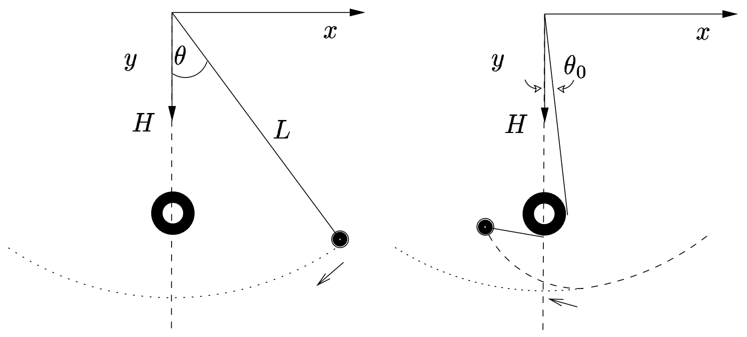


Figure 2.3.5: Constrained pendulum.

For this system, it is no longer possible to define the whole trajectory with a single differential equation set. In this case it should be defined as follows:

$$0 = \begin{cases} f_1(\dot{x}, x, u, t) & \text{if } \theta > \theta_0 \\ f_2(\dot{x}, x, u, t) & \text{if } \theta \leq \theta_0. \end{cases}$$

This is an example of a system with two configuration or two *modes.*

A transition between two modes occurs when some conditions on the continuous-time state are fulfilled (e.g., a temperature reaches a threshold and implements the discrete-time state change of turning off a heater). Thus the timing of transition is a function of the solution of the differential equations governing the system. The transition time is given by a transition condition:

$$g(\dot{x}, x, u, t) > 0,$$

where x is the continuous-time state vector and u is the input, and t is the time. Whenever $g(.)$ crosses zero, this condition switches its logical value. An *event* is defined as the earliest time at which one of the currently pending transition conditions becomes true. This event is a so called **zero-crossing event** or **state-event** in contrast with the events that whose activation time are known in advance, i.e., they depend only on time (**time-event**).

Zero-crossing events are the mechanism whereby the state of the continuous-time subsystem influences the discrete-time subsystem. Zero-crossing events which are dependent on state variables of the system are more difficult to handle accurately in time than the time-events. To locate the instant in time where the discontinuity appears, an additional function is introduced i.e., a zero-crossing function, which changes its sign at the discontinuity. By calculating the value of this function, it is possible to detect the discontinuity. This function is also called the *zero-crossings surface* or *root function.*

2.4 Hybrid System Simulation

A hybrid system is composed of discrete-time components and continuous-time components, and a hybrid signal looks like the plot in Fig. 2.2.3 where at discrete-time times, or simply at events, the discrete-time part is activated and their state values remain unchanged until the next event arrives. Between every two discrete-time, a differential equation defines the system behavior.

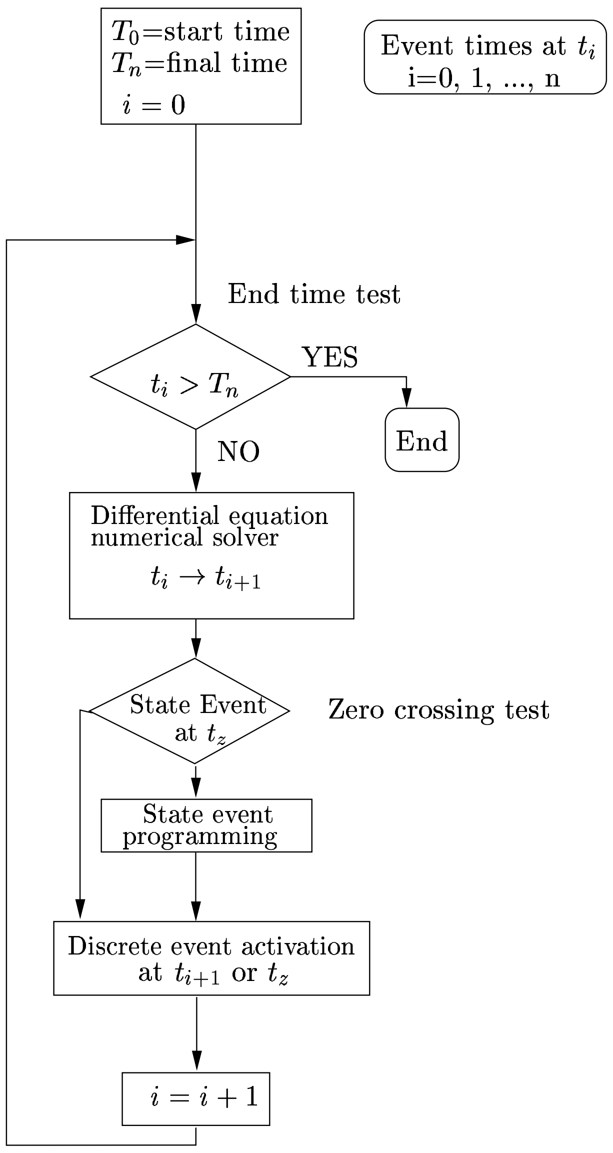


Figure 2.4.1: Basic hybrid system simulator.

As a result, the simulation is composed of two interlaced phases, *continuous-time-phase* and *discrete-time-phase*. In discrete-time phase, several events may be activated, but the simulation time does not advance. In continuous-time phase, a numerical integrator (numerical solver) integrates the active differential equation and the simulation time is advanced. The diagram in Fig. 2.4.1 shows a simple architecture for a hybrid system simulator.

In continuous-time phase, depending on the nature of the differential equation an ODE solver or a DAE solver is called. To integrate a differential equation, there are several numerical method such as explicit Euler, implicit Euler, Runge-Kutta, BDF, etc. In the next section, a BDF method for numerical integration and then DASKR which is a numerical solver based on BDF will be introduced.

2.4.1 BDF Based Solvers

Simulating continuous-time systems requires solving the initial value ODE/DAE problems numerically. A widely used class of algorithms, called time-marching algorithms, discretizes the continuous-time-line into an increasing set of discrete-time instants, and numerically computes state variable values at these time instants in increasing order. The discretization of time reflects the trade-off between speed and accuracy of a simulation, and is handled based on the error tolerance of the solutions and the order of the algorithms.

There are several integration methods, such as one-step methods (e.g., the simple explicit Euler and Runge-kutta) and multi-step methods (e.g., the BDF method). BDF (Backward Differentiation Formula) methods are implicit linear multi-step methods that depend on multiple previous solution points to generate a new approximate solution point. The BDF method of order 1, is the implicit Euler method, i.e., to integrate $\dot{x} = f(x,t)$, the DAE is discretized as

$$\frac{x_{n+1} - x_n}{h_{n+1}} = f(x_{n+1}, t_{n+1}),$$

and is solved for x_{n+1}. In a BDF method of order n, the solution is advanced at each step by interpolating n previous solution points along with the as yet unknown new solution point, differentiating that interpolant, and requiring the derivative to match the ODE at the new point. Specifically, for an ODE $\dot{x} = f(x,t)$ and approximate solution points $(t_{k-n+1}, x_{k-n+1}), ..., (t_k, x_k)$, the approximate solution value x_{k+1} at time $t_{k+1} = t_k + h_k$ is determined by solving the implicit equation $\dot{w}(t_{k+1}) = f(t_{k+1}, x_{k+1})$ for x_{k+1} where $w(t)$ is the unique polynomial of degree n that interpolates $(t_{k-n+1}, x_{k-n+1}), ..., (t_k, x_k), (t_{k+1}, x_{k+1})$. BDF methods have relatively large stability regions, so they are particularly suitable for solving stiff ODEs. The stability region decreases as order increases.

BDF method has been used for ODEs successfully, for DAEs certain precautions should be taken into account. The basic idea for solving DAE systems using numerical ODE methods, originating with Gear [?] is to replace the derivative in (2.4.1) by a difference approximation, and then to solve the resulting system for the solution at the current time t_{n+1} using Newton's method. Suppose the DAE is

$$F(\dot{x}, x, t) = 0, \tag{2.4.1}$$

replacing the derivative in (2.4.1) by the first order backward difference, we obtain the implicit Euler formula

$$F\left(\frac{x_{n+1} - x_n}{h_{n+1}}, x_{n+1}, t_{n+1}\right) = 0, \tag{2.4.2}$$

where $h_{n+1} = t_{n+1} - t_n$. This nonlinear system is then usually solved using some variant of Newton's method for x_{n+1} at t_{n+1}, for example,

$$x_{n+1}^{m+1} = x_{n+1}^{m} - J^{-1} F\left(\frac{x_{n+1}^{m} - x_n}{h_{n+1}}, x_{n+1}^{m}, t_{n+1}\right),$$

where m is the iteration index, and the Jacobian matrix J is defined as

$$J = \frac{1}{h_n}\frac{\partial F}{\partial \dot{x}} + \frac{\partial F}{\partial x}.$$

In some numerical solvers such as DASKR and LSODAR, the k^{th} backward difference formula is used, where k may range from 1 to 5. The step-size h_n and the order k varies, depending on the behavior of the solution. The k-step BDF method on the n^{th} integration step can be represented as

$$\sum_{j=0}^{k} \alpha_j \, x_{n+1+j-k} = h_{n+1} \, \beta_k \, \dot{x}_{n+1},$$

where the past values of x, $(x_{n+1+j-k}\ ,j = 0,\dots\ ,k-1)$ are known. This equation may be re-arranged to give:

$$\dot{x}_{n+1} = \frac{1}{h_{n+1}\,\beta_k}\left(\alpha_k \, x_{n+1} + \sum_{j=0}^{k} \alpha_j \, x_{n+1+j-k}\right),$$

which can be rewritten as

$$\dot{x}_{n+1} = \frac{1}{h_{n+1}\,\beta_k}\left(\alpha_k \, x_{n+1} + \rho(k, n+1)\right),$$

where $\rho(k, n+1)$ is the collection of terms in the previous step. Substituting this into the (2.4.1), yields

$$F\left(\frac{1}{h_{n+1}\,\beta_k}\left(\alpha_k \, x_{n+1} + \rho(k, n+1)\right), x_{n+1}, t_{n+1}\right) = 0,$$

This equation is then solved for x_{n+1}. As the step-size varies, computed past values of x may be unequally spaced, whereas in the above formula, equally spaced past values are used. This problem is solved using the fixed leading-coefficient strategy. The Jacobian Matrix that should be provided is

$$J = \alpha\frac{\partial F}{\partial \dot{x}} + \frac{\partial F}{\partial x},$$

where $\alpha = \dfrac{\alpha_k}{h_n \beta_k}$ is given by the solver. The Jacobian matrix either can be provided by the user or can be computed numerically.

Continuity Criteria for Numerical Solvers

During a simulation of a dynamic system, an equation or its derivatives may be discontinuous at a discrete value of the independent variable (at an event time). In such a situation special care has to be taken. Since the BDF methods are based on polynomial backward time-expansions of each variable, beyond a discontinuity point, a BDF method normally results in repeated step failure and reduction of the integration step-size to obtain a coherent derivative and to meet the accuracy requirements. For some problems the solver gives up with an error message when the step gets too small, and in some other cases the solver may continue with an erroneous result [**?**].

Numerical solvers cannot integrate discontinuous functions, but some numerical solvers, however, can integrate functions with discontinuous first derivatives. Although this is with a cost of low order and small step-sizes. As a result, simulation slows down in the vicinity of the discontinuities. Numerical solvers assume that input signals and their derivatives are continuous. So a simulator should provide the solver a smooth function and prevent the solver from attempting to integrate through discontinuities. In case of discontinuity, the solver must be restarted at the discontinuity point.

Continuity Assumption at Initialization

Differential variables typically represent conserved quantities like mass, energy, entropy or other quantities that are closely related to conserved ones. These quantities are conserved in closed physical systems; only external actions change them. Mostly, their initial value can be freely chosen, so that they represent degrees of freedom. The values of the differential variables can be freely chosen at the beginning of the simulation. Note that the value of a differential variable cannot be freely chosen if it is a dependent variable and/or there are hidden constraints in the equations. After an event or a discontinuity, by default, whose values are assumed to be continuous before and after the discrete-time. Suppose that the consistent initial conditions of (2.4.3) after a discontinuity at t_k should be calculated. Suppose we have the DAE

$$f(\dot{x}(t_k),\, x(t_k),\, y(t_k),\, t_k) = 0. \tag{2.4.3}$$

Let $x(t_k^-)$ and $x(t_k^+)$ denote the differential states (2.4.3) just before and just after an event, respectively. Then the default continuity assumption holds

$$x(t_k^+) = x(t_k^-). \tag{2.4.4}$$

In other words, in an initialization phase when solving (2.4.3), x is constant (equal to $x(t_k^-)$ by defalut). Note, that in the continuity assumption of differential variables (2.4.4) does not preclude the possibility of modeling instantaneous changes to them. For instance, the model of a bouncing ball is

$$\begin{aligned} \dot{y} &= v \\ \dot{v} &= -g, \end{aligned}$$

where y and v are vertical position and vertical velocity correspondingly. Although both are differential states, the velocity state is changed at a discontinuity point, i.e., $v(t_0^+) = -\rho v(t_0^-)$. In most models such a change denotes an interaction with the model environment. For example, some substance is added to a chemical process or charge is added to an electrical circuit by closing a switch.

Role of Derivative in Initialization

In DAE initializations, state derivatives play an important role that will be discussed here. A general DAE takes the form $F(\dot{x}, x, t) = 0$. Initialization at time t_0 requires values of $x(t_0)$, and $\dot{x}(t_0)$. In subsequent steps the unknowns are only the next value of x and predictions are based on past values of x. In Jacobian evaluation, the $\dot{x}$ is estimated from a combination of state values. Thus the estimated value of $\dot{x}(t_0)$ has three uses. The first is to evaluate the Jacobian during the initialization. Secondly it will continue to be used for the Jacobian evaluation until the first time the Jacobian is updated. After that it will have no direct effect. One of the advantages of implicit methods like DASKR and other BDF based methods is that it is not necessary to have the exact Jacobian and Jacobians are kept as long as the Newton iterations converge fast enough at each time step.

There is a third role of $\dot{x}(t_0)$, that is the value of $\dot{x}(t_0)$ determines which solution the integrator will track. Consider the implicit ODE

$$\tan(\dot{x}) = 0, \quad t_0 = 0.$$

There are an infinite number of families of solutions of the form of $x(t) = x(0) + n\pi t$ for each $n = 0, 1, \ldots$. It is the value of $\dot{x}(t_0) = n\pi$ which determines which solution the simulation will track. It does this by generating the estimate of $\dot{x}(t_1)$ which is found to be $n\pi$. After finding $x(t_1)$, we have two values of $x(t) = x(0) + n\pi t$ and further initial guesses of $\dot{x}(t_i)$ will again be $n\pi$.

2.4.2 DASSL Family of Solvers

DASSL is a multi-step, implicit solver developed for large systems of stiff ODEs and fully-implicit index-1 and semi-explicit index-2 DAEs [**?**]. DASKR is a new version of DASSL, DASPK and DASRT(DASSL with root finder) that performs root finding and also solving the initial state problem [**?**, **?**, **?**, **?**].

These solvers are widely used in many industrial and military applications. DASXX is available from NETLIB[1] as part of the ODEPACK. DASXX can be obtained as either a serial (Fortran 77) or parallel (Fortran 90) subroutine. This solver implements the multi-step BDF of orders 1-5. The formula order and step-size are selected adaptively to maintain stability and efficiency. The code includes many diagnostic features and user-defined arguments for customizing and optimizing routines in the code for a particular application. The potential strength of this method lies in the ability to take larger, stable time steps (yielding fewer iterations) for a level of accuracy comparable to other methods.

DASSL requires two user-provided subroutines: a residual routine in which the system of DAE's is defined; and a root-finding routine in which the zero-crossing surfaces are defined (for DASRT and DASKR). DASSL should be called by a calling routine in which the initial conditions (except for DASKR) and control parameters are defined. Runtime parameters (some are optional) include maximum step size, matrix banding, maximum order, and the absolute and relative error tolerances. Outputs from DASSL include reports on the order of the method, step size of the last successful step, number of residual evaluations, number of Jacobian evaluations, and number of convergence test failures.

In this chapter, these solvers, their advantages and their shortcomings with emphasis on their impact on hybrid simulation will be discussed.

In the next sections, several aspects of DASSL/DASKR that are relevant to its integration and use in a hybrid simulator are discussed. These are step-size selection, convergence and accuracy, discontinuity handling, root finding, consistent initial condition finding, and condition number improving. Many practical tips about the use of DASSL can be found in [**?**].

[1] www.netlib.org

Some Definitions Used in DASSL family

- **Current step size:** The step-size that is used in the current integration step.
- **Suggested next step-size:** DASSL has an error control capability and the integrators is able to predict an estimation for the next step size based on the local error of the current step. Suggested next step-size value is the minimum of all the predictions from the integrators.
- **Initial step size:** This is the step-size that the user specifies as the desired starting step size. For fixed step-size solvers, this step-size will be used in all iterations, But for DASSL, this is only a reference.
- **Minimum step size:** The lower bound for adjusting step-sizes. If this step-size is used and the errors are still not tolerable, the step is considered failed, and the simulation aborts.
- **Minimum time resolution:** This controls the comparison of time. Since the simulator works with double-precision real numbers, it is sometime impossible to reach or step at a specific time point. If two time points are within this resolution, then they are considered identical.
- **Value resolution:** This is used in the fixed point iterations. If in two successive iterations the states are within this amount, then the fixed point is considered reached.
- **Maximum number of iteration by step:** This is used to avoid the infinite loops in the fixed point iterations. If the number of iterations exceed this value but the fixed point is still not found, then the fixed point procedure is considered failed. The simulation then aborts.
- **Info vector:** This is an integer array used to communicate details of how the solution is to be carried out, such as tolerance type, matrix structure, step-size and order limits, and choice of nonlinear system method.
- **Rwork, Iwork vector:** Real and integer work arrays which provide the code with needed storage space. These arrays are also used to provide the necessary information to code quantities such as minimum step-size, maximum integration order, etc.
- **IDID variable:** This scalar integer is an indicator reporting what the code did on return. It should be monitored to decide what action to take next.

Zero-Crossing and Discontinuity Handling

The objective of state event modeling is to circumvent attempts by the numerical integration code to integrate across sudden changes (that is, instantaneous changes of state values in system equations) in the system. Usually such attempts cause the local error criterion of the integration algorithm to fail or have convergence difficulties. These problems are most acute when the time the event occurs is not known a priori. If state events are handled properly, observe that they do not disturb the numerical integration of the continuous-time system because they are handled outside of the integrators. To detect and handle the state events a transition condition should be checked. The transition condition becomes true if its associated function crosses zero. That is why the transition associated function is also called *zero-crossing* function.

Most modern numerical solvers, including DASKR/LSODAR, are able to find the point where the zero-crossing function crosses zero to high accuracy. Then the solver returns to the calling program, and then the associated state-event can be generated. Besides the use of zero-crossing functions to generate the state-events, another important functionality of zero-crossing is handling the discontinuities in ODE/DAEs. If an ODE/DAE function contains an inherent discontinuity, at discontinuity points the derivatives of the signals are not defined, so the integration formula is not applicable. That means the discontinuity points cannot be crossed by one integration step. In particular, suppose the current time is t and the intended next time point is $t + h$. If there is a breakpoint at $t + \delta$, where $\delta < h$, then the next step-size should be reduced to $t + \delta$. For a predictable breakpoint, like a time-event, the main program (simulator) can adjust the step size accordingly before starting an integration step. However, for an unpredictable discontinuity, the simulator should be able to discard its last step and restart with a smaller step-size to locate the actual discontinuity. Most of the time, discontinuities are defined with transition condition or zero-crossing function. As a result, they can be detected like state-events.

To use DASKR/LSODAR as an event detector, the number of crossing functions, as well as the name of an external routine to compute the crossing-functions values should be provided.

Algebraic and Differential Variable Declaration

The equations solved by DASKR are in the form

$$F(\dot{x}, x, t) = 0,$$

where $\dot{x}$, x are N dimensional vectors and F is a vector valued function. Note that no distinction is made between differential and algebraic variables because DASSL/DASKR implements a backward differentiation formula (BDF) method. But in DASKR there are two new options to help the use to find the consistent initial condition and exclude the algebraic variables from the error test. For these options, DASKR should be informed about the differential and algebraic variables. This is done via initializing `Iwork(Lid+I)` before running DASKR (`Lid` is an address index). To identify which variables are the differential and which are the algebraic components (algebraic components are components whose derivatives do not appear explicitly in the function $F(\dot{x}, x, t)$). The user must set:

- `Iwork(Lid+I) = +1` if $x(i)$ is a differential variable
- `Iwork(Lid+I) = -1` if $x(i)$ is an algebraic variable,

Consistent Initial Calculation

When using either of the solvers DASSL or DASPK, the integration must be started with a consistent set of initial conditions. Consistency requires, in particular, that $F(\dot{z}_0, z_0, t_0) = 0$. Usually, not all of the components of $\dot{z}_0$ and z_0 are known directly from the original problem specification. The problem of finding consistent initial values can be a challenging task. DASKR, the latest version of DASSL, provides the facility to compute the initial condition of index-1 DAE, in two cases. Let

$$0 = F(\dot{z}, z, t),$$

where $z = \begin{pmatrix} x \\ y \end{pmatrix}$ and x and y are differential and algebraic variables respectively. Then the two cases are:

- x is considered as known and y and $\dot{x}$ are calculated.
- $\dot{z}$ is considered as known and z is calculated.

In the first case, if $v = \begin{pmatrix} y \\ \dot{x} \end{pmatrix}$, then $\frac{\partial F}{\partial v}$ should be square and nonsingular. In the second case, $\frac{\partial F}{\partial z}$ should be non-singular. In most models of industrial systems the default continuity assumption together with the set of valid DAEs can uniquely define the initial condition, i.e., the differential variables are assumed to be known, so the first initialization method will be able to find the unknown part of initial states. However, there are cases when the default continuity assumption does not hold for some of the differential variables. The most common situation is when a process is initialized to its steady-state. Steady-state initialization is possible by using the second initialization method and assigning $\dot{z} = 0$. This problem does not involve a split of the z vector into differential and algebraic parts or a semi-explicit form for the equations.

DASKR has an integer array argument `Info` which is used to specify a variety of options. In the latest version of DASKR, the user is to set `Info(11)` to have DASKR solve one of the two initialization problems [?].

- `Info(11) = 0` means initial values are already consistent (the default).
- `Info(11) = 1` means solve the first initialization problem. In this case, the user must identify the differential and algebraic components of z.

Some times, specially when system initialization is not just finding the consistent initial condition, we need to return to the main program after the initial condition calculation, before proceeding to the integration of the DAE of the system. This possibility has been introduced in DASKR, i.e., DASKR is called with `INFO(11)=1`, and if the initialization succeeded, it returns with `IDID = 4` indicating successful initialization. For the next call or start integration, the user should reset `INFO(14)` to `0` to prevent the solver from repeating the initialization (and to avoid an infinite loop).

Step-Size and Order Selection

In order to initiate the BDF method, it is required to obtain a number of solution points equal to the order of the method. This may be done either by using a lower order multi-step method with the drawback of smaller step sizes or by using single-step methods with the drawback of having to implement these separately. In DASSL the first possibility has been used by implementing an increasing order BDF which works well due to the variable step size method. On the first step, DASSL takes the first order BDF formula, and a small step. The initial step-size is calculated from the formula

$$h_0 = sign(T_{out} - T) \cdot \min(10^{-3} \left| T_{out} - T \right|, \frac{1}{2} \|\dot{x}\|^{-1}).$$

After the first step, the order and the step-size are gradually increased. DASSL uses the order and step-size selection strategy described in [?]. At each step, the order and the step-size of the next step is determined. When the step-size is increased, it is increased by factor of 2, when it is decreased it is decreased by a factor between 0.9 and 0.25. The default maximum order is 5, however it can be modified by the user. In stiff region, the step-size reduces to meet the accuracy requirement, but it cannot become less than h_{min} which is calculated from

$$h_{min} = 4.0\, u\ \max(|t|, |t_{out}|),$$

where u is the unit round of error of the computer (for a double precision floating point numbers to represent reals, $u = 2^{-52}$ so that $u = 10^{-16}$), t is the current integration time, and t_{out} is the user requested output point. The simulator takes the minimum step-size into consideration. If the next discrete-time phase is closer to the current time than the minimum step-size, then the simulation time is not advanced, and the next discrete-time phase is scheduled at the current time.

Accuracy, Error Tolerances

The true (global) error is the difference between the true solution of the initial value problem and the computed approximation. Practically all present day codes, including DASSL, control the local error at each step and do not attempt to control the global error directly. Usually, but not always, the true accuracy of the computed x is comparable to the error tolerances. DASSL will usually, but not always, deliver a more accurate solution if the user reduces the tolerances and integrates again. By comparing two such solutions the user can get a fairly reliable idea of the true error in the solution at the looser tolerances.

In DASSL it is possible to control the error and reach a desired accuracy. It is quite useful to be able trade off simulation time for accuracy. The requested accuracy in the solution is specified by the error tolerances `Rtol` and `Atol`. `Rtol` and `Atol` represent absolute and relative error tolerances (on local error) which the user provides to indicate how accurately user wishes the solution to be computed. Each variable should have a distinct tolerance because they may have a different magnitude and accuracy. So `Rtol` and `Atol` should be defined as vector. In DASSL it is possible to define scaler (one tolerance for all variables) as well as a vectorial tolerances. The tolerances are used by the code in a local error test at each step which requires roughly that

$$\|\texttt{local error in } x_i\| \leq \texttt{Rtol} * |x_i| + \texttt{Atol},$$

for each vector component. More specifically, a root-mean-square norm is used to measure the size of vectors, and the error test uses the magnitude of the solution at the beginning of the step.

Setting (`Atol=0`) results in a pure relative error test on that component. Setting (`Rtol=0`). results in a pure absolute error test on that component. A mixed test with non-zero `Rtol` and `Atol` corresponds roughly to a relative error test when the solution component is much bigger than `Atol` and to an absolute error test when the solution component is smaller than the threshold `Atol`. DASSL will not attempt to compute a solution at an accuracy that is unreasonable for the machine being used. It warns the user if too much accuracy has been requested.

Cold/Hot-Start of Solver

There are two methods for initializing DASSL: a *cold-start* and a *hot-start*. A cold start is the first call to DASSL for a given problem. In this case, many cumbersome memory setting is done, then as there is no memory about the solution, a very small step-size and a first order method should be used at the beginning. A hot start, on the other hand, is a call to the solver that is not the first call for the given problem. In this case, assuming no change in problem or discontinuity, the solver does not performs the memory settings and can use all past information about solution. In this case the solver can start integration with previous [big] order and [long] step-size. This makes the hot-start much more efficient than a cold start. It should be noted that, if there is any change in model or any discontinuity in the solution, solution history before discontinuity or model change are useless in approximating the derivative of x after the discontinuity. The solver should resolve the new initial conditions and start the solving process as if it is at the starting point, i.e., it should be cold-restarted.

Forward Looking in Time (Tstop)

An important property of LSODAR and DASKR is the possibility to set constraints on the solver. For example, constraints on the relative and absolute error tolerances can be imposed globally or on each state variable separately. A time constraint can be imposed to forbid the solver to advance the time beyond a given time called the *stopping time*. Normally the solver is allowed to step beyond the final integration time and return the value at the final time by interpolation. This increases the efficiency when the integration is restarted.

Consider a situation where the user wants to integrate a DAE from t_0 to t_f and many intermediate output points are required. It is not efficient to run DASSL from one output point to the next one and then return to the main program to print the output points, even if DASSL is called with a hot-start option. To handle this situation efficiently, DASSL may integrate past the output points and interpolate back to obtain the results at output points. This increase the efficiency, but sometimes it is not possible to integrate beyond some point t_s because the equation changes there or simply because the solution or its derivative is not defined past t_s.

To overcome this obstacle, DASSL has another important property, i.e., the possibility to set a time constraint on the solver. Time constraint can be imposed to forbid the solver to advance time beyond a given time called the stopping time. This stopping time (`Tstop`) can be set via `Info(4)`.

- `Info(4) = 0`: There is no restriction on time.
- `Info(4) = 1`: The integration should be carried out with restrictions on the independent variable(time). In this case, the `Tstop` is given by setting `Rwork(1) = Tstop`.

Jacobian Matrix Calculation

In section 2.4.1, we saw that in order to integrate the DAE

$$F(\dot{x}, x, t) = 0,$$

DASSL needs the Jacobian matrix

$$J(x) = \alpha \frac{\partial F}{\partial \dot{x}} + \frac{\partial F}{\partial x},$$

where α is a scaler given by the solver. An analytical expression of the Jacobian obtained through symbolic computation is more accurate and leads to better simulation performance.

So if the system description is available symbolically, Jacobian evaluation by symbolic manipulation should be performed if possible. There are situations, however, where symbolic information is not available or practical. This happens in many situations such as when the dynamics of the system is defined by black box computer programs.Furthermore, for small and/or simple systems of equations, it may be possible to write out manually the analytical Jacobian of F. For even moderately sized problems, however, it may be next to impossible (e.g., for n=10, the full Jacobian contains 100 elements). In these cases, we will need to compute the Jacobian numerically, i.e., we will perform a numerical differentiation. As an aside, it should be mentioned that large scale analytical differentiation can be performed using symbolic math packages such as MAPLE or MATHEMATICA. But for certain Scicos blocks we can only evaluate the function and we do not have access to their analytic derivatives, so it is not a good solution.

A convenient technique for computing the $J(x)$ is to use *difference approximation* at x that can be performed by finite differencing, i.e., we make a small perturbation to one of the x elements (keeping the others constant), evaluate the function, and use the difference in function value over the perturbation as a measure of the differential with respect to this x element. If σ_j is the size of the j'th perturbation and e_j is the unit vector with one in the j'th element and zero elsewhere, then the Jacobian elements at x_n are calculated as

$$J(x_n) \approx D_j = \frac{F(x_n + \sigma_j e_j) - F(x_n)}{\sigma_j} \; j = 1, ..., n, \tag{2.4.5}$$

where D_j is the j^{th} column of J.

In DASSL/LSODAR, the Jacobian matrix can be either provided by the user or computed numerically by DASSL.

- `Info(5) = 0`: The Jacobian is computed automatically by DASSL via difference approximation
- `Info(5) = 1`: DASSL uses the provided Jacobian matrix

Condition Number Improvement

Consider a linear system of equations

$$Ax = b$$

where A is a square matrix. Here we look at effect of round-off error or error, in general, in b on the stability of solutions. By stability, we are interested in how the solution x changes when a small change or error is made in b. We say that the solution is stable if a small error in the data leads to a small error in the solution. Let us first derive some relationship between the perturbation of the data and resulting change in the solution. Consider a perturbation of the right hand side of the above equation, i.e.,

$$A(x + \delta x) = b + \delta b \Longrightarrow \delta x = A^{-1}\delta b,$$

and using some suitable norms we get

$$\|\delta x\| \leq \|A^{-1}\| \cdot \|\delta b\|.$$

Also since

$$\|b\| = \|Ax\| \leq \|A\| \cdot \|x\| \Longrightarrow \frac{1}{\|x\|} \leq \|A\| \frac{1}{\|b\|},$$

multiplying the previous two equations we get

$$\frac{\|\delta x\|}{\|x\|} \leq \|A\| \cdot \|A^{-1}\| \frac{\|\delta b\|}{\|b\|}.$$

This equation gives an upper limit to the error in the solution in terms of the error/perturbation in b. What makes this important is that the inequality is strict, i.e., there exist b and δb for which equality holds. The quantity

$$\kappa(A) = \|A\| \cdot \|A^{-1}\|,$$

is called the *condition number* of matrix A. The condition number shows how much the round-off error is magnified and appears in the solution. If the condition number of the Jacobian matrix is large, then there is a possibility of a large error in the solution. This problem shows up in the solution of index-1 and more seriously for index-2 DAE systems. In DAE integration, a linear system is solved at each Newton iteration. For explicit ODE, as $h_n \to 0$, the Jacobian matrix tends to the identity. For index-p DAEs, the condition number of the Jacobian matrix is $O(h_n^{-p})$. To illustrate this, consider the backward Euler method applied to the following semi-explicit index-1 DAE

$$\begin{aligned} \dot{x} &= f(x, y, t) \\ 0 &= g(x, y, t). \end{aligned}$$

The Jacobian matrix is

$$J = \begin{pmatrix} \alpha I - f_x & -f_y \\ -g_x & -g_y, \end{pmatrix},$$

where α is proportional to $\frac{1}{h_n}$. The condition number of this matrix is $O(\alpha)$. For big α (small h_n), this can lead to failure of the Newton iteration. However, scaling can improve the situation. In this case, multiplying the algebraic equation by α yields a Jacobian matrix whose condition number no longer depends on α in this way.

Note that multiplying the differential equations by α^{-1} does not help, because it affects the accuracy. The termination criteria of iteration in the solver is based on the size of the error in residual. If the differential equations are multiplied by h_n, then the solver may meet the tolerances without really having an accurate enough solution.

In DASKR, this new option has been introduced to increase the accuracy and convergence. In the calling sequence of the external routine that reads the residuals, DASKR provides the scalar α. If the user can distinguish the differential and algebraic equations, multiplying algebraic ones by α can help in some stiff situations.

2.4.3 DASSL Family Shortcomings

Hybrid systems have some features, that the user should to be aware of. For complex physical systems, often end up with an ordinary differential equation. As a result, a numerical differential equation solver should be used to solve it. The solver introduces numerical inaccuracy in the results, which might even lead to incorrect behavior. The solver has requirements that have to be fulfilled; some of them are not checked by the compiler, so that the user has to take care of them. In the following section several shortcomings of DASKR/LSODAR that we encountered during simulation of some thermo hydraulic applications will be explained.

Zero-Crossing Starting Around Zero

DASKR can be used to detect state events. For example, a conditional statement like $x > 1$, is delivered to DASKR as a zero-crossing function $g(x) = x - 1$. The problem is that DASKR sometimes did not find the root, especially if $g(x)$ is zero or around zero at the beginning of the integration step. For instance, in the above example if we start the solver when x is very close to 1, say $1 - 10^{-6}$, and $\dot{x}$ is positive, the solver often fails to detect the zero.

Zero-Crossing and Time Derivative of Variables

In some applications, a state event depends on the derivative of a state variable. In DASKR Zero-crossing statements could not contain a time derivative of a variable. For instance,

$$\begin{aligned} 0 &= F(\dot{x}, x\, t) \\ G_{zc} &= g(\dot{x}), \end{aligned}$$

where G_{zc} is the vector of zero-crossing functions, was not acceptable for DASKR. Note that, if the x is a differential variable, in theory it is possible to substitute $\dot{x}$ with a function of other variable. But if it is a algebraic variable, it is no longer possible. Since, in the latter case, introducing a new variable for the time derivative of variable will increase the DAE index by 1.

Zero-Crossing Direction

In some hybrid system applications, there are situations where the crossing-time of the zero-crossing is not enough; it is also important to know in which direction the crossing has occurred. LSODAR/DASKR did not provide this information.

Zero-Sticking

Zero-crossings are not just used to generate state-events in order to communicate with the discrete-time part of the model. In most cases zero-crossings are used to properly control the solver to simulate non-smooth continuous-time dynamics. In some hybrid applications, it can very well happen that a zero-crossing function remains at zero over a period of time. This means, from the solver point of view, that one of the zero-crossings is stuck on the value zero. This situation is not properly dealt with in the solvers and, for example in DASKR, it stops the integration.

Minimum Step-Size (Hmin)

In the integration of a DAE, to meet the requested accuracy, DASKR reduces the step-size when it encounters a stiff region. The minimum step size is limited by the H_{min} variable, defined in DASKR. H_{min}, which is computed as a function of integration time, may cause problems in some stiff problems specially at big simulation times.

Initial Condition Calculation

the integration of a DAE can be started only with consistent initial states. Although DASKR can be used to compute the consistent initial states of DAEs, there are still some problems to be addressed in initialization, specially in Jacobian matrix computing and its evaluation rate.

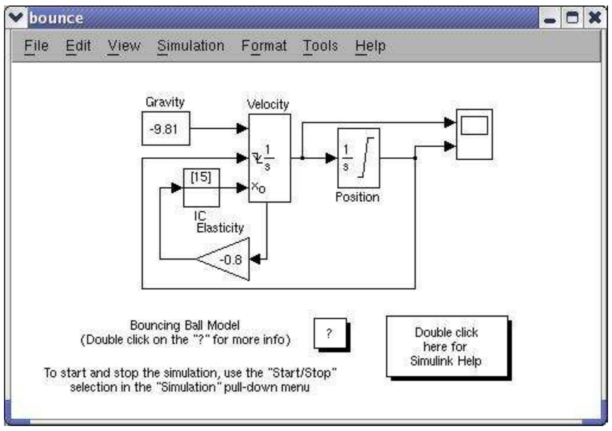
2.5 Survey Of Hybrid Simulators

Interactions between the discrete-time and the continuous-time components results in new simulation phenomena that are typical of hybrid systems. A hybrid simulator, therefore, must be able to handle these. There is, however, no general agreement as to what extent a simulator should be able to handle these phenomena to qualify as a hybrid simulator. In [?], [?] two surveys of some hybrid simulation packages have been presented. Here an overview on the simulators which have been used as hybrid system modeling and simulation tool will be given.

Matlab, Simulink, Stateflow

Simulink [?], trademark of MathWorks, is a software package for modeling, simulating, and analyzing dynamic systems. Simulink is part of the Matlab tool family and implements a dataflow-oriented, graphical language consisting of diagrams with blocks representing data transformations and connecting lines representing data signals. It supports linear and non-linear systems, modeled hierarchically in continuous-time, sampled time, or a hybrid of the two. Systems can also be multirate, i.e., have different parts that are sampled or updated at different rates. The dataflow notation lends itself very well to modeling mathematical equations and Simulink offers a library of basic function blocks for this purpose. With this library, the characteristic differential equations of feedback control systems can be modeled.

State-flow [?] is an interactive design and simulation tool for event-driven systems. State-flow provides the language elements required to describe complex logic in a natural, readable, and understandable form. It is tightly integrated with MATLAB and Simulink, providing an environment for designing embedded systems that contain control, supervisory, and mode logic. In Fig. 2.5.1, a Simulink block diagram modeling of a bouncing ball system has been shown. The simulation result is shown in Fig. 2.5.2.


bounce
File Edit View Simulation Format Tools Help
Gravity
-9.81
Velocity
1/s
x_o
[15]
IC
Elasticity
-0.8
1/s
Position
?
Double click here for Simulink Help

Bouncing Ball Model
(Double click on the "?" for more info)

To start and stop the simulation, use the "Start/Stop" selection in the "Simulation" pull-down menu

Figure 2.5.1: A hybrid system modeled in Simulink.

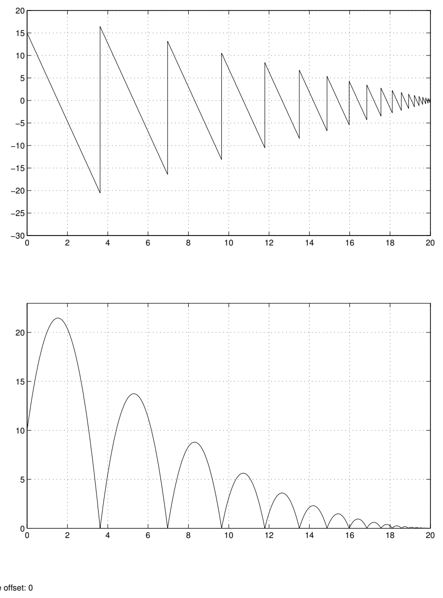
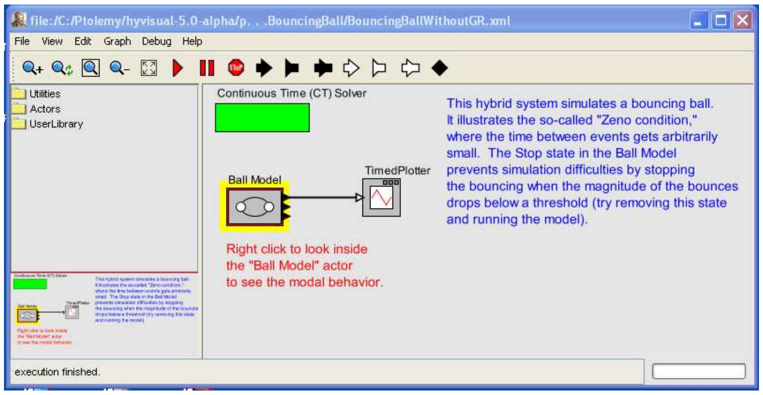


Figure 2.5.2: Simulation result of model in Fig. 2.5.1 in Simulink.

HyVisual, Ptolemy II

The Hybrid System Visual Modeler (HyVisual) is a block-diagram editor and simulator for continuous-time dynamical systems and hybrid systems. It is a Java-based component assembly framework with a graphical user interface. HyVisual is built on top of Ptolemy II, a framework that supports the construction of such domain specific tools, and can be freely downloaded from [?]. The Ptolemy project studies modeling, simulation, and design of concurrent, real-time, embedded systems. The focus is on assembly of concurrent components. The key underlying principle in the project is the use of well-defined models of computation that govern the interactions between components. A major problem area being addressed is the use of heterogeneous mixtures of models of computation. The diagram in Fig. 2.5.3 show how a Ptolemy model looks like, the simulation result of this model is given in Fig. 2.5.4.

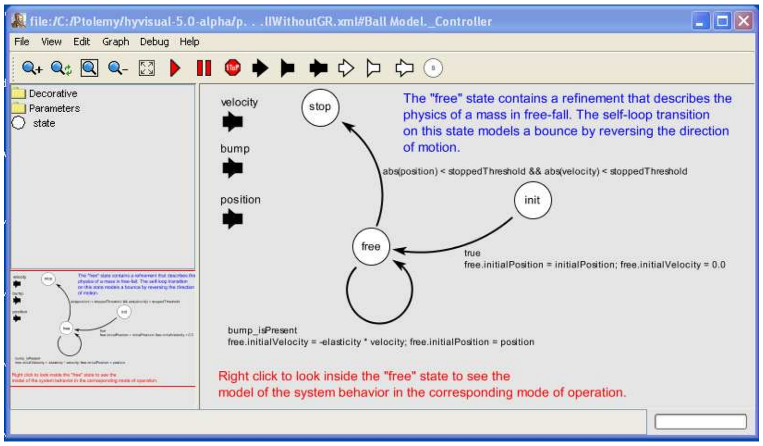

file:/C:/Ptolemy/hyvisual-5.0-alpha/p. . .BouncingBall/BouncingBallWithoutGR.xml
File View Edit Graph Debug Help
Utilities
Actors
UserLibrary
Continuous Time (CT) Solver
Ball Model
TimedPlotter

This hybrid system simulates a bouncing ball. It illustrates the so-called "Zeno condition," where the time between events gets arbitrarily small. The Stop state in the Ball Model prevents simulation difficulties by stopping the bouncing when the magnitude of the bounces drops below a threshold (try removing this state and running the model).

Right click to look inside the "Ball Model" actor to see the modal behavior.

execution finished.

Figure 2.5.3: Bouncing ball system modeled in Ptolemy.


file:/C:/Ptolemy/hyvisual-5.0-alpha/p. . .llWithoutGR.xml#Ball Model._Controller
File View Edit Graph Debug Help
Decorative
Parameters
state
velocity
bump
position
stop
init
free

The "free" state contains a refinement that describes the physics of a mass in free-fall. The self-loop transition on this state models a bounce by reversing the direction of motion.

abs(position) < stoppedThreshold && abs(velocity) < stoppedThreshold

true
free.initialPosition = initialPosition; free.initialVelocity = 0.0

bump_isPresent
free.initialVelocity = -elasticity * velocity; free.initialPosition = position

Right click to look inside the "free" state to see the model of the system behavior in the corresponding mode of operation.

Figure 2.5.4: State machine inside the model in Fig. 2.5.3.

NI-MATRIXx, SystemBuild, NI-LabVIEW

SystemBuild [?] is the core of the NI-MATRIXx [?] products. It is a graphical environment for rapid model development and simulation of complex dynamic systems, as well as specifying and testing control and software algorithms. SystemBuild has a hierarchical block-diagram modeling paradigm designed for simple development of complex models based on an extensive library of primitive blocks. SystemBuild also has a graphical user interface layer that creates interactive simulations incorporating user input, in addition to standard batch simulations. System build uses data logging capabilities for postprocessing, analysis, and interactive visualization of simulation data. On can extend the SystemBuild modeling capabilities through additional advanced analysis and design modules. Model options include fuzzy logic design, neural network design, optimization and control, aerospace libraries, configuration management for use with source code control software, state transition diagram editors, and more.

NI-LabVIEW [?], another trademark of National Instrument, is the graphical development environment for creating test, measurement, and control applications. LabVIEW is an open environment designed to interface with measurement hardware.

Dymola

Dymola uses hierarchical object-oriented modeling to describe the systems, subsystems and components of a model. There are several libraries available for Electronics, Hydraulics, Thermal, etc. The models are written in the object-oriented modeling language Modelica. Models are usually built hierarchically and graphically by dragging component models from libraries and connecting them. Equations are used at the lowest level, facilitating true reuse of models in different contexts. Model details are given by (ODE) and algebraic equations, that is DAEs. Discontinuous equations are handled by translation to discrete-time events as required by numerical integration routines.

Symbolic processing is used to make simulations efficient. Dymola converts the differential-algebraic system of equations symbolically to state-space form, i.e., solves for the derivatives. The equations are then, if possible, solved symbolically or code for efficient numeric solution is generated. Higher index DAEs, typically obtained because of constraints between sub-models, are handled by symbolically differentiating equations [?].

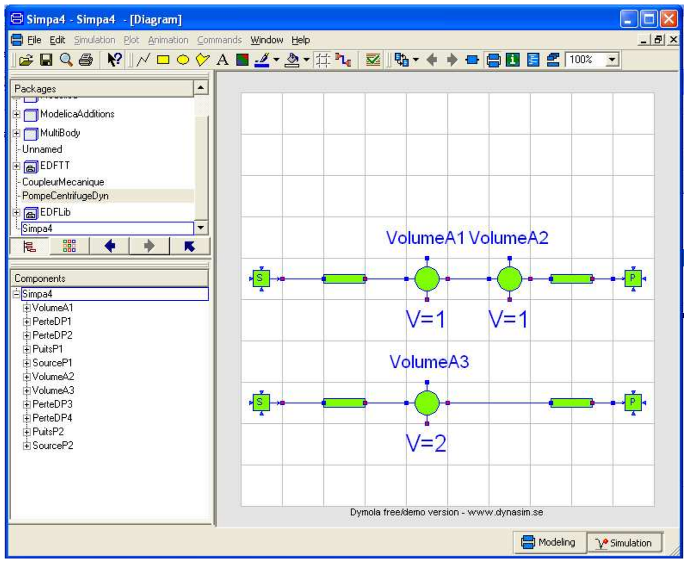


Figure 2.5.5: A physical system modeled in Dymola.

Abacuss II

ABACUSS (Advanced Batch And Continuous Unsteady-State Simulator) [?], developed for chemical engineering systems, supports hybrid models, model inheritance, and hierarchical model decomposition. It facilitates guaranteed state event location, batch process simulation, solution of high-index differential algebraic equations, dynamic and steady-state optimization, and dynamic sensitivity and uncertainty analysis. ABACUSS II is the next generation open modeling environment and simulator. Designed from the ground up to be as flexible as possible, it can be used standalone or embedded within another application. Alternatively, ABACUSS II can be easily and seamlessly embedded within another application (e.g., Microsoft Excel or an automation software system).

20-SIM

20-SIM ("Twent Sim") [?] is a modeling and simulation program that runs under Windows and Sun-Unix. 20-SIM is used to simulate the behavior of dynamic systems, such as electrical, mechanical and hydraulic systems or any combination of these systems. It supports graphical modeling, allowing the user to design and analyze dynamic systems. It supports also the use of components that allows for building a model by choosing components from the library and interconnecting them. Fig. 2.5.6 shows a sample model built in 20-SIM.

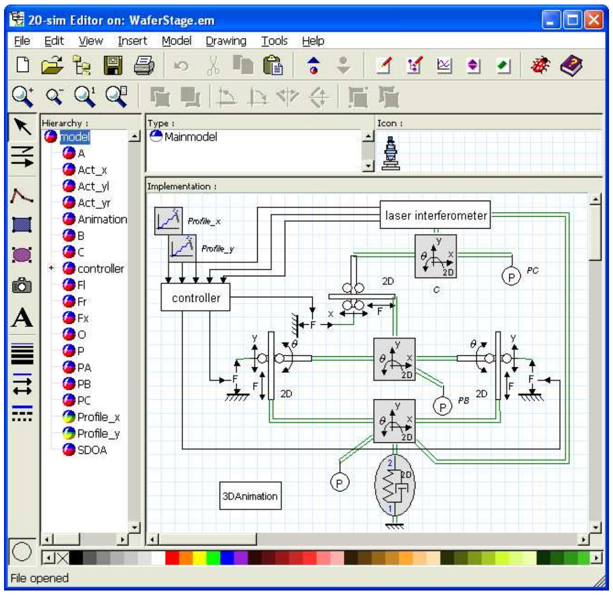


Figure 2.5.6: A hybrid system modeled in 20-SIM.

Chi (χ)

The Chi [?] design system is a concurrent programing language based on the (Timed) CSP [?] formalism. The Chi language can be used to express models of industrial systems. A model expressed in Chi is simulated using the Chi simulator. In Chi, a hybrid process may have a continuous-time part (initialization of variables determining the continuous-time state of the process, links, DAEs), a discrete-time part (initialization of other variables, discrete-time statements) or a combination of both. A continuous-time type declaration looks like:

```
  type name = [SIunits]
```

A hybrid process programed in Chi looks like:

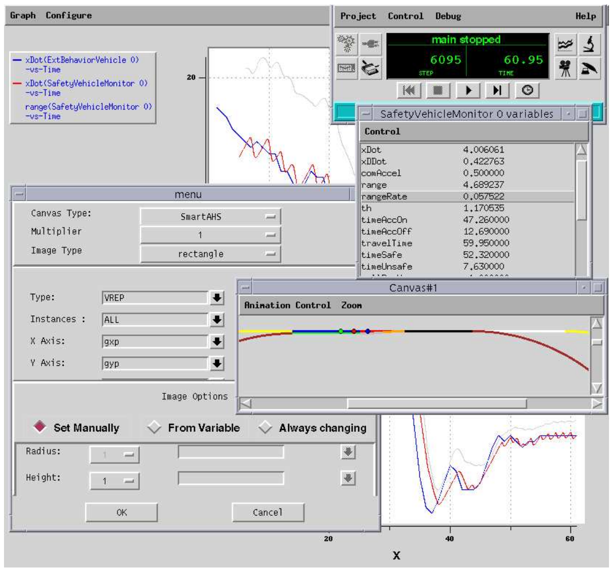
```
proc name(parameters) =
|[ local variables
; initialisations
| links
% equations
| discrete-event statements
]|
```

SDX

SDX [?] is Fortran based Problem Solving Environment for dynamics (continuous-time, discrete-time, hybrid) related applications in science and engineering. It can be used for modeling and simulation of dynamic systems characterized by differential, difference and algebraic equations.

Shift and SmartAHS

Shift [?] is a programing language for describing dynamic networks of hybrid automata. Such systems consist of components which can be created, interconnected and destroyed as the system evolves. Components exhibit hybrid behavior, consisting of continuous-time phases separated by discrete-time transitions. SmartAHS is a specification, simulation and evaluation framework for modeling, control and evaluation of Automated Highway Systems (AHS). SmartAHS is developed using Shift.


Graph
Configure
xDot(ExtBehaviorVehicle 0) -vs-Time
xDot(SafetyVehicleMonitor 0) -vs-Time
range(SafetyVehicleMonitor 0) -vs-Time
20
Project
Control
Debug
Help
main stopped
STEP
TIME
SafetyVehicleMonitor 0 variables
Control
menu
Canvas#1
Animation Control
Zoom
Image Options
OK
Cancel
20
40
60
X

main stopped: 6095 STEP, 60.95 TIME

```
xDot           4.006061
xDDot          0.422763
comAccel       0.500000
range          4.689237
rangeRate      0.057522
th             1.170535
timeAccOn      47.260000
timeAccOff     12.690000
travelTime     59.950000
timeSafe       52.320000
timeUnsafe     7.630000
```

Canvas Type: SmartAHS
Multiplier: 1
Image Type: rectangle
Type: VREP
Instances : ALL
X Axis: gxp
Y Axis: gyp
[✓ Set Manually] From Variable Always changing
Radius: 1
Height: 1

Figure 2.5.7: Screenshots from Shift/SmartAHS Simulations.

Sildex

Sildex [?], trademark of TNI-Valiosys, is an integrated toolset for formally specifying and designing control and data-oriented real-time embedded systems. In particular, Sildex targets safety-critical embedded software applications whose failure may lead to disastrous consequences. It is used in the aerospace, automotive, energy, telecom, and defense industries. The Sildex approach is based on the synchronous paradigm and the synchronous language Signal. Simulation and code generation all rely on formal semantics of Signal language. The software aspects of the Signal language are transparent to the users of Sildex. In fact, they are not necessarily required to master the language to be able to manipulate Sildex design specifications. Instead, they can simply use the graphical design editors while benefiting from the formal semantics that lies in the background.

Simcreator

SimCreator [?] is a graphical, hierarchical, real time simulation and modeling system. SimCreator's GUI interface allows the user to develop distributed simulation models. It is a block diagram modeling simulator. Fig. 2.5.8 shows how a model in simcreator looks like.

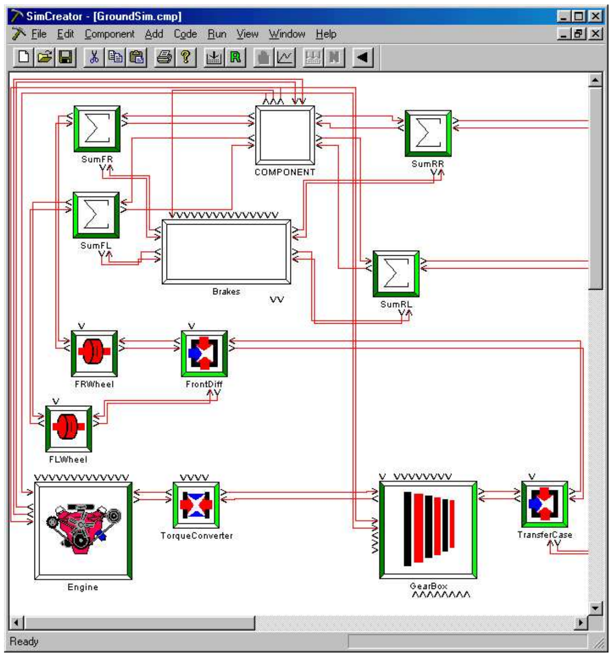


Figure 2.5.8: Simcreator system simulator.

EASY5

EASY5 [?], developed by Boeing corporation, offers a set of Application Libraries that are targeted to a specific application with pre-built models of physical devices such as hydraulic valves and actuators, internal combustion engines, electric motors, gears, clutches, heat exchangers, fans, and evaporators. These physical subsystems can be used to construct a dynamic system model. Fig. 2.5.9 shows how a model in EASY5 looks like.

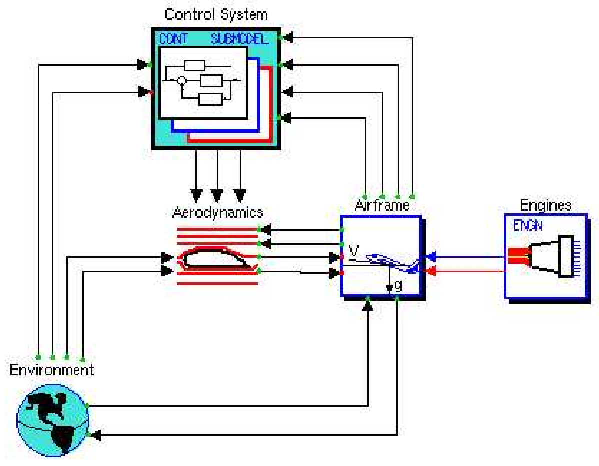


Figure 2.5.9: EASY5 system simulator.

Chapter 3

Scicos

The name Scilab [?] stands for **Sci**entific **lab**oratory. Scilab is a language for scientific computing. It provides an environment to calculate, visualize, and program the problems. Scilab was initially devoted to matrix operations, and scientific and engineering applications were its main target. In fact, it was written to provide an easy access to matrix software such as LINPACK, EISPACK, LAPACK and BLAS. But over time, it has considerably evolved and currently the Scilab language includes powerful operators for manipulating a large class of basic objects [?, ?].

Scilab is an interactive system whose data objects do not need to be declared or allocated. This allows the user to solve many technical computing problems, especially those with matrix and vector formulations, requiring less time then it would take to write a program in a language such as C or Fortran. Scilab includes hundreds of mathematical functions with the possibility to add interactively programs from various languages (C, Fortran...). It has sophisticated data structures (including lists, polynomials, rational functions, linear systems...), an interpreter, and a high level programing language. Scilab has been designed to be an open system where the user can define new data types and operations on these data types by using overloading. Scilab contains several specialized toolboxes that allow the user to extend the Scilab environment to work with particular classes of problems. A number of toolboxes are available with the system:

- 2-D and 3-D graphics, animation
- Linear algebra, sparse matrices
- Polynomials and rational functions
- Simulation: ODE solver and DAE solver
- Classic and robust control, LMI optimization
- Differentiable and non-differentiable optimization
- Signal processing
- Metanet: graphs and networks
- Parallel Scilab using PVM

- Statistics
- Interface with Computer Algebra: Maple package for Scilab code generation, MuPAD 3.0 includes Scilab
- Interface with Tcl/Tk
- **Scicos**: a hybrid dynamic systems modeler and simulator

Scilab recognizes several data types. Scalar objects are constants, booleans, polynomials, strings and rationals (quotients of polynomials). These objects in turn allow the user to define matrices which admit these scalars as entries. As an example, to define a complex matrix, it is enough to type the following command:

```
--> b=23;
--> j=sqrt(-36);
--> A=[2 b 4;5 -8 2*j];
```

The row elements are separated by commas or spaces and column elements by semi-colons. Scilab shows the result as follows:

```
 A         =
!   2.    23.   4. !
!   5.  - 8.    12.i !
```

In addition to internal functions, the user can define their own functions. Functions are collections of commands which are executed in a new environment thus isolating function variables from the original environments variables. Functions can pass arguments, have programing features such as conditionals and loops, and can be recursively called. Functions can be arguments to other functions and can be elements in lists. The most useful way of creating functions is by using a text editor, however, functions can be created directly in the Scilab environment using the `deff` primitive. This is an example of a nested function definition and its execution from the Scilab window:

```
function y=foo(x)
  a=sin(x)
  function y=sq(x),
    y=x^2,
  endfunction
  y=sq(a)+1
endfunction

-->foo(%pi/3)
 ans  =

    1.75
```

3.1 Scicos

Scicos [?] (**Sci**lab **C**onnected **O**bject **S**imulator) is a toolbox of Scilab and provides an environment for modeling and simulation of dynamical systems. The underlying formalism in Scicos allows for modeling very general dynamical systems: systems including continuous-time, discrete-time and event based behaviors. Scicos could have been an independent software for modeling and simulation, but having access to Scilab and its functionalities brings about a lot of flexibility and widens the range of modeling capabilities. For example, in signal processing, it is quite easier to use Scilab functions and write a small program than writing a code for basic signal processing functions. Furthermore, Scilab sees the Scicos model as a function. When Scilab calls Scicos, the model is returned. This is useful when a Scicos batch process should be run.

Scicos can be considered as a graphical editor for constructing block diagram models by interconnecting blocks, representing predefined or user defined functions. But what is a block diagram model?

3.1.1 Block Diagram Modeling in Scicos

The most fundamental concept for control systems engineering is the block diagram. A classic block diagram model of a dynamic system graphically consists of blocks and links (that represent the continuous-time signals and discrete-time events). This concept comes from engineering fields such as Feedback Control Theory and Signal Processing. A block within a block diagram defines a dynamic system in itself. The relationships between each elementary dynamic system in a block diagram are illustrated by the use of signals connecting the blocks. Collectively the blocks and lines in a block diagram describe an overall dynamic system, for example see Fig 3.1.1.

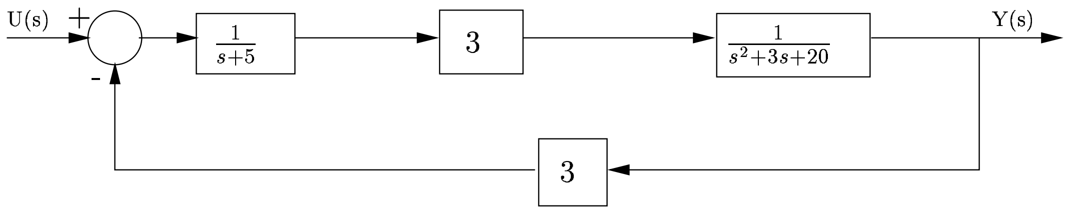


Figure 3.1.1: A typical control systems.

The simplest element of a block diagram system is a block. A block takes the input and generates the output through some dynamical actions. The details of the internal workings of each block looks like a black box to the rest of the world. In other word, all the model is concerned with is the relationship between the input and output of the block. Each block is connected to the others via signal lines that may represent discrete-time or continuous-time signals. A continuous-time signal represents a quantity that changes over time and is defined for all points in time during simulation, while a discrete-time signal is a piecewise continuous signal that only an event changes its value. An event represents a distinct point on the time axes that causes a change in signals.

The relationship between input and output of a block is defined by a set of continuous-time and discrete-time equations. These equations define a relationship between the input signals, output signals, and the internal state variables.

Discrete-time events play an important role in the design of hybrid systems. In order to explicitly define the discrete-time events, a special link is used in Scicos. They are called *activation links*, in contrast to *regular links*. A discrete-time event comes from an event source and can activates several continuous-time or discrete-time blocks. Events indicate activation at certain time points. A Scicos block diagram model can have several event sources. There are two important definitions about events, i.e., *Simultaneous Events* and *Synchronous Events*.

- Two events are simultaneous only if they are activated at the same time
- Two events are synchronized only if they are activated by the same block

Two simultaneous events are not necessarily synchronized, and two synchronized events are not necessarily simultaneous.

3.1.2 Scicos Blocks

A Scicos block is the basic constructing element of a Scicos diagram and can be a complex entity. It can have several inputs and outputs, continuous-time states, discrete-time states, zero-crossing functions, etc. But note that it is not necessary to have all these elements at the same time. A Scicos block can have two types of inputs and outputs:

- *Regular inputs*, to receive data from other blocks through regular links.
- *Regular outputs*, to send data to other blocks through regular links.
- *Activation inputs*, to receive control information (events).
- *Activation outputs*, to transmit control information (events).

Activation inputs and outputs are connected by activation links. Note that it is not possible to treat an event link as a regular (continuous-time or discrete-time) link or to connect them.

Scicos provides a graphical editor that allows creating and connecting block types selected from block palettes. A block corresponds to an operation and by interconnecting blocks through links, we can construct a model, or an algorithm. These blocks represent elementary systems that can be used as model building blocks. To get an idea what a Scicos model looks like, the model of Fig. 3.1.1, has been implemented in Scicos, see Fig. 3.1.2.

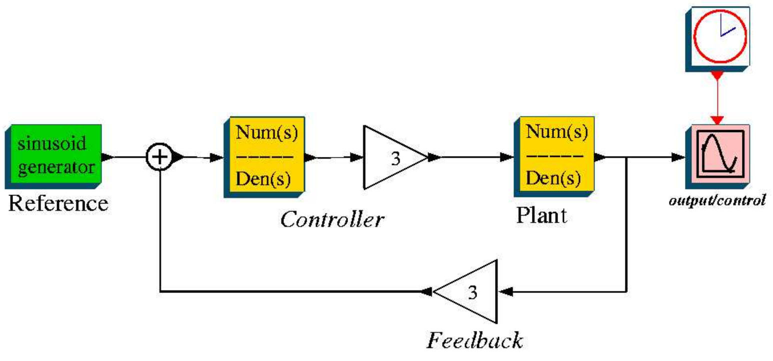


Figure 3.1.2: Scicos implementation of control model of Fig. 3.1.1.

In the Scicos block diagram of Fig. 3.1.2, these block have been used:

- a `Linear system block` that provides a transfer function ($\frac{Num(s)}{Den(s)}$) between its input (u) and its output (y),
- two `Gain block` that multiply the input by a constant value (gain),
- an `Adder block` to add its inputs and put the result in its outputs
- a `Sinewave block` that provides a sinusoid wave,
- an `Event Clock block` that generates discrete-time events (time events) to activate the Scope block (Scope's sample time),
- and a `Scope block` that displays its input with respect to simulation time.

In Fig. 3.1.2, The link between these last two blocks is an *activation link*.
These blocks are taken from Scicos libraries (palettes). Scicos provides several block libraries, some of them are:

- **Sinks:** blocks used to display, write into a file,..
- **Linear:** linear blocks normally used in control system modeling, such as gain, linear system, adder, etc.
- **Non_linear:** nonlinear blocks, such as saturation, absolute value, divider, and multiplier.
- **Sources:** the signal or event source blocks, such as absolute time, read from file, step function.
- **Events:** discrete-time events related blocks, such as switch, event adder, event select, event clock, and If then else block.
- **Threshold:** state event generators, i.e., zero-crossing blocks.
- **Branching:** signal and event routers, such as multiplex, and selector blocks.
- **Other blocks:** some general blocks such as debugger, backlash, constraint, etc.

Scicos users can also create their own block types and use the Scicos editor to create instances of them in a diagram. We can group these blocks into two categories: basic functional blocks and special blocks. In the following sections, we will briefly explain some of important blocks.

Elementary Blocks

Scicos provides many blocks representing elementary systems that can be used as building blocks. Most of the blocks are basic functional blocks, such as integrator block, gain, constant, etc. In these blocks the following information is assumed to be known:

- Block inputs signals
- Block inputs events (time or discrete-time events)
- States (discrete-time or continuous-time states)

- Parameters (information from the block dialog box)

and these values are computed:

- Block output signals
- Block event outputs
- Block states

The elementary blocks perform a computation function, the outputs are computed based on internal states, input signals and input events of the block. An example is a `Gain` block, which multiplies the inputs by some fixed value and passes the result to the output. These blocks may have both discrete-time and continuous-time states and they may generate output events, but they are not synchronized with input events. Fig. 3.1.3 shows the internal structure of an elementary block.

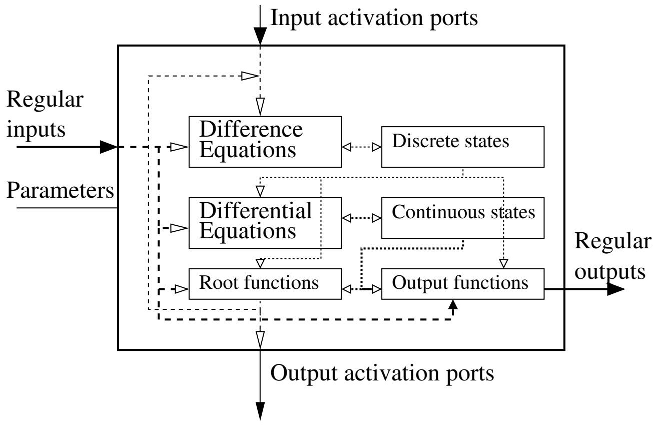


Figure 3.1.3: A classic Scicos block.

Time-Event Generator Blocks

A block generates a time-event either it is programed to generate an event (only one) at a specified time, or it can delay an input event. Delaying an event means receiving an event in and transmitting it out with a positive or zero delay. In the later case, it should be noted that the generated event is not synchronous with the original event. The `event clock` block uses these two options to generate a serie of events. It is simply a block with its output event port fed to its own input event port with an initial pre-programed event. In Fig. 3.1.4, `Delay`, `Event clock generator`, and `Time delay` block have been shown.

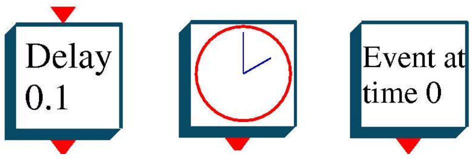


Figure 3.1.4: Some time-event generators in Scicos.

Zero-Crossing Blocks

Zero-crossing events are introduced to overcome the difficulty in the modeling of continuous-time systems when there are some discontinuities in the system. Because discontinuities may cause problem on the operation of continuous-time integration methods. The numerical solvers of Scicos, offer an option to detect the state-events. Scicos uses this option in the blocks. For ordinary uses, Scicos has introduced the `Zero-Crossing` block. The purpose of this block or zero-crossing feature is to handle discontinuities in state values and switching between system equations. Applications for zero-crossing events arise in hybrid system for unpredictable events. For example, monitoring the liquid level in a tank and closing the intake tap as soon as the liquid level exceeds a certain level. This can be done by use of a `Zero-Crossing` block.

In some applications, in addition to the crossing time, we need to know the crossing direction. For these purposes Scicos has provided two other blocks `'+ to -'` block and `'- to +'` block that generate a zero-crossing event whenever the input signal crosses the zero in positive or negative direction. In Fig. 3.1.5, `Zero crossing`, `- to +`, and `+ to -` block have been shown.

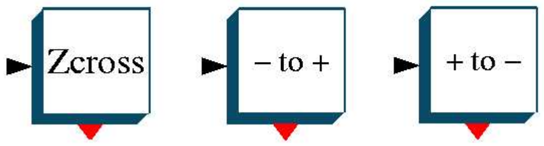


Figure 3.1.5: Zero-crossing event generators in Scicos.

Conditional Blocks

The basic functional blocks perform some computational functions, whereas the *Conditional blocks* control the execution in the model. These blocks can conditionally activate other blocks in a Scicos model. The `If-then-Else` and `Event select` blocks, see Fig. 3.1.6, provide frameworks for conditional activation.

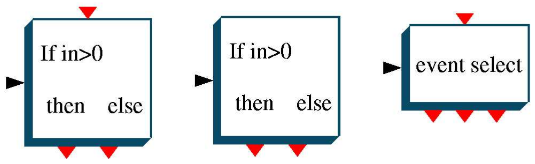


Figure 3.1.6: Conditional blocks in Scicos.

The main difference between the two blocks is the number of their output events. They are the counterpart of the `If then Else` and `Switch: Case` statements in imperative programing languages like C. Whenever these blocks receive an event, they immediately generate a synchronous event at their event output port as a function of the value of their regular input. The Condition block controls the path of events. Depending on the inputs of the condition block, output activation goes out either from the `If` port or from the `Else` port. The two activation outputs are mutually exclusive and their sum (union of activation times) equals its input activation.

The condition activating can be used to conditionally activate other discrete-time or continuous-time blocks. Here is an example to show how the `If-then-Else` block is used to conditionally select the input to another system (see Fig. 3.1.7).

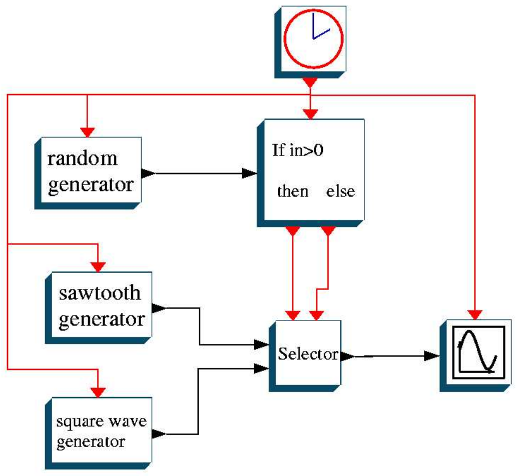


Figure 3.1.7: Conditional activation.

The block `Selector` has two input activation ports. The block can be activated by only one of them at a moment. Whenever `Selector` is activated it knows the activating port and uses this information to update its output the first or the second input depending on the port through which it has been activated. The output of the `Selector` is the sawtooth signal generated by `Sawtooth Generator` block as long as the signal is positive. If it is negative or zero, the output of the Selector is a `Square Wave` signal.

If an `If-then-Else` block does not have an input event port, see Fig. 3.1.6 (middle), it controls the continuous-time blocks that receive the activation. In this case, the numerical solver sees only the blocks that are in the active branch of `If-then-Else`.

These blocks are also referred to as *Synchro blocks*. Synchro blocks have built-in zero-crossing surfaces that generate an event when the activated output port changes. These conditional blocks are very special blocks, they do not have a computational function and internal states, they have only one input and only one output event is activated at a moment.

Superblocks

Scicos has a hierarchical architecture, i.e., we can use it to define SuperBlocks. The SuperBlock mechanism provides an interactive design environment for creating and editing complicated block diagrams. A block diagram may contain several blocks and Superblocks, and each Superblock may contain blocks and other Superblocks. Clicking on a Superblock, displays its contents in a separate Scicos editor window. Fig. 3.1.8 shows a SuperBlock which is used to simplify the Scicos diagram of Fig. 3.1.2.

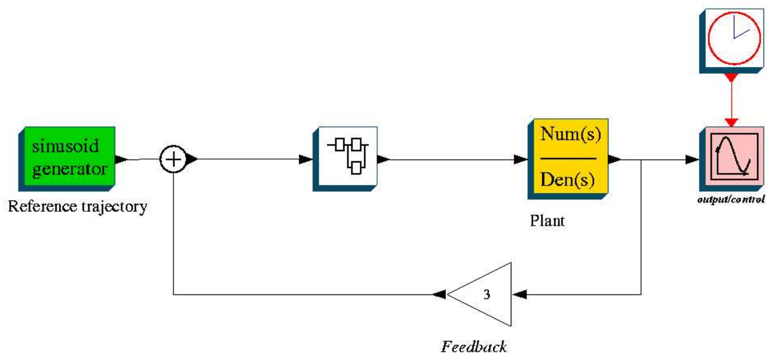


Figure 3.1.8: SuperBlock mechanism in Scicos.

Note that a SuperBlocks is just a graphical representation to facilitate the modeling. It is not a real block and it does not change the simulation at all.

3.2 Modeling Hybrid Dynamical Systems in Scicos

A Scicos diagram is obtained by the interconnection of Scicos blocks. If all inputs are connected and no algebraic loop exists, a Scicos diagram defines a Scicos model. Models created in Scicos can be broadly categorized as follows:

- Discrete-time models
- Continuous-Time models
- Hybrid models (a combination of continuous-time and discrete-time subsystems).

In this section we will explain how these models can be modeled in Scicos.

3.2.1 Modeling Discrete-time Systems

In the Scicos libraries there are several blocks for modeling discrete-time systems, such as `unit delay (1/Z)`. These systems need to be activated by events. For example the sampling time of a discrete-time system may be defined with a `event clock` block. Suppose that we want to model this discrete-time system with sampling rate `T=0.5 sec`:

$$y(k+1) = 0.8\, y(k) + 1.2.$$

Fig. 3.2.1 shows how this system can be modeled in Scicos.

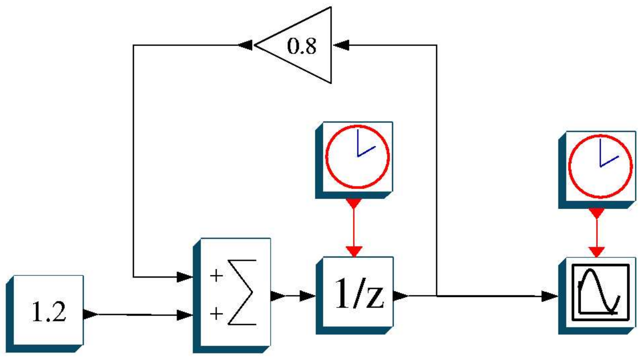


Figure 3.2.1: Modeling a discrete-time System.

In this model, the discrete-time state $y(k)$ is modeled by a `1/Z` block and an `Event clock` block is used to generate discrete-time events to update the discrete-time state. This block has been programed to generate its first event at $t = 1$ and a periodic event series with sampling time $T_s = 0.5$.

For discrete-time systems, given user-defined initial conditions and input vector, the procedure of simulating the discrete-time Scicos model or obtaining a sequence of solutions to the system equations is fairly straightforward. Starting from the given initial conditions, the discrete-time state equations are iterated until the specified final time. The simulation result for the initial condition $y(0) = 1$, and final time $T_f = 10$ is given in Fig. 3.2.2.

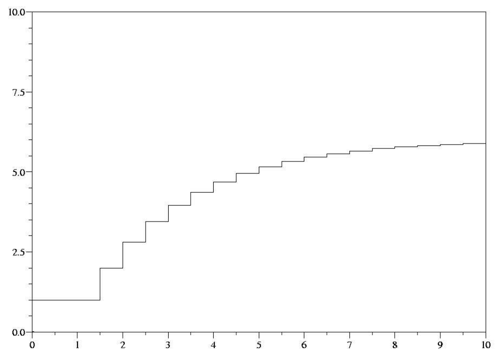


Figure 3.2.2: Simulation result of model of Fig. 3.2.1 in Scicos.

3.2.2 Modeling Continuous-time Systems

A Scicos block diagram model is a graphical representation of a mathematical model of a dynamic system. A mathematical model of a dynamic system is described by a set of differential equations. For instance, Fig. 3.2.3, shows how this continuous-time system:

$$\dot{x} = \sin(t) - 0.9\,x,$$

is modeled in Scicos.

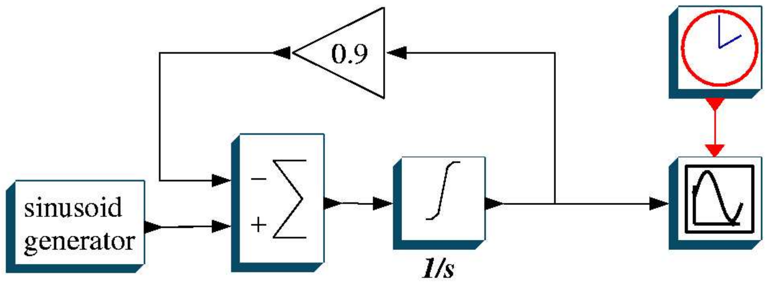


Figure 3.2.3: Modeling a continuous-time System.

In this model an `Integrator`, a `Summation`, a `Gain`, a `Sine generator`, a `Scope`, and finally, an `Event clock` have been used. The `Integrator` block contains the continuous-time state of system (x), so its input should be equal to ($\dot{x}$). In this model `Event clock` is used to periodically activate the `Scopes` block to plot the result on Scope's screen. So all the connections are carrying the continuous-time signals, except for connection between `Scope` and `Event clock`, which carries the activation signal. The simulation result for the initial condition $x(0) = 10$ is given in Fig. 3.2.4.

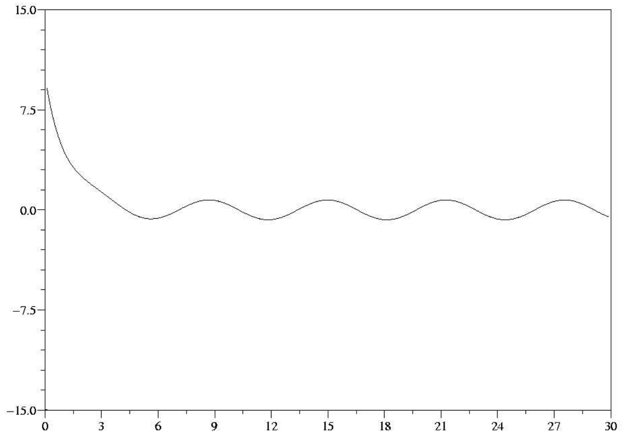


Figure 3.2.4: Simulation result of model of Fig. 3.2.3 in Scicos.

3.2.3 Modeling Hybrid Systems

After modeling two purely continuous-time and purely discrete-time systems, let us model some hybrid systems in Scicos. Continuous-time operations and discrete-time event dependent operations can interact in different ways. First, continuous-time and discrete-time signals can be inputs to a one block. In fact, fundamentally, there is no difference between a discrete-time signal and a continuous-time signal. In fact a Scicos signal can have discrete-time property over a period of time and later continuous-time property. This means that in Scicos we can perform operations (such as addition of) continuous-time and discrete-time signals. Then, Continuous-time signals can generate events through zero-crossing blocks. Finally, events can create jumps in continuous-time states as well as in discrete-time states. Consider the model of the bouncing ball system. In this case, the continuous-time state changes abruptly with an event (discontinuity in velocity). The bouncing ball system has been modeled in Fig 3.2.5.

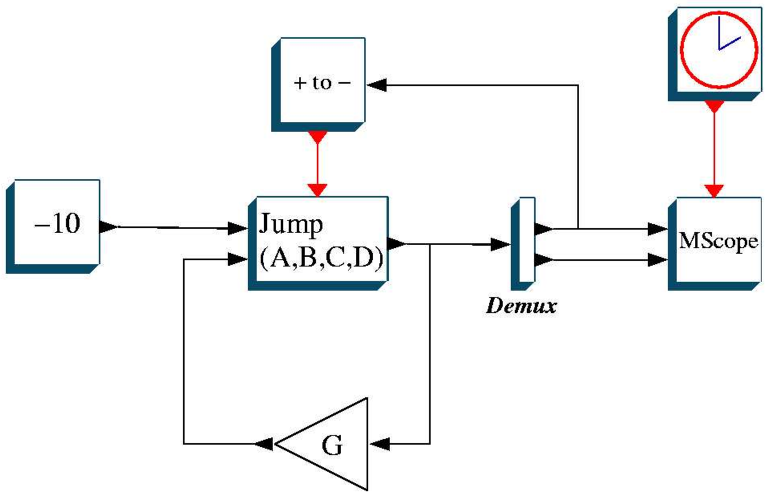


Figure 3.2.5: Modeling a hybrid system (Bouncing ball) in Scicos.

In this model we have used a `Linear system with jump possibility` block, a `Gain` block to return and reduce the velocity after collection, a `Constant` block to provide the downward acceleration (`-10`), and a `Zero-cross` block to generate an event and reverse the velocity. The `Zero-cross` block monitors the *altitude* of the ball whenever it crosses zero, an event is generated. This event is fed to `linear system` to make a discontinuous jump in states. The simulation result has been shown in Fig. 3.2.6. Note the discontinuity in velocity in the bottom plot.

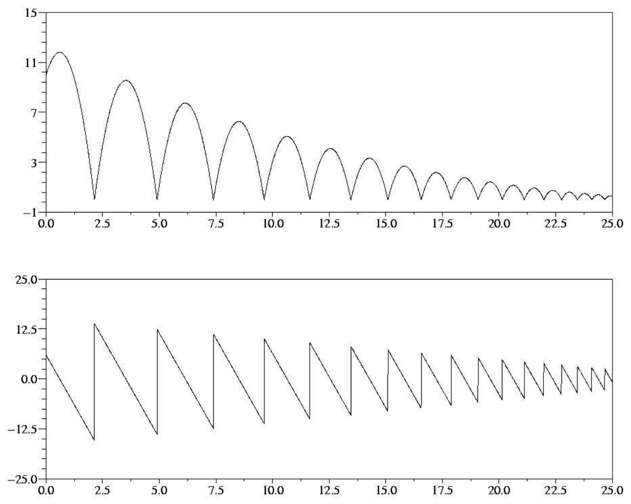


Figure 3.2.6: Simulation result of model of Fig. 3.2.5 in Scicos.

Another interesting model is activating a discrete-time system with a state event. Consider again the example in Fig. 3.2.1, but instead of `event clock`, we use a `+ to - zero-crossing` block to generate the events to activate the `delay` block, the result has been shown in Fig. 3.2.7.

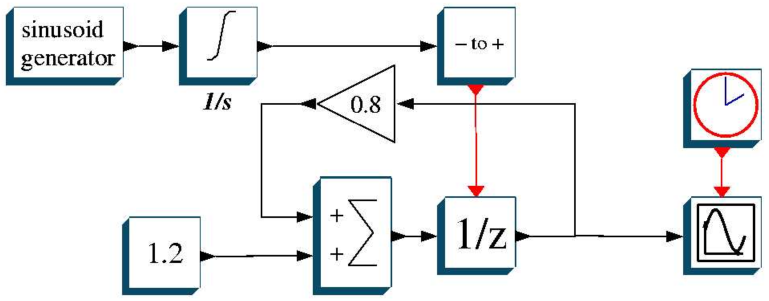


Figure 3.2.7: Activating a discrete-time system of Fig. 3.2.1via zero-crossing events.

3.2.4 Simulation Parameter Setting

Scicos can simulate the model from a $t = 0$ to a specified final time. Depending on the model, it uses an ODE solver for ordinary differential equation, a DAE solver for differential algebraic equations, and if the model is discrete-time, none of these solvers is used. Scicos is a general simulator, but each problem has its own particular specifications. That needs to introduce some flexibility in simulation and treating the models. This section discusses the simulation parameters, which the user should specify before running the simulation.

- **Final Simulation Time:** the start time is always $t_0 = 0$. The final time for the simulation is set by entering new values in the `final integration time` fields. Its default time is 10000 seconds.

- **Realtime Scaling:** Simulation time and actual clock time are not the same. For example, running a simulation for 10 seconds usually does not take 10 seconds. The amount of time it takes to run a simulation depends on many factors, including the model's complexity, the solver's step sizes, and the computer's speed. Most of the time the simulation time is less than the real time. This option increases the real time simulation by setting Scicos unit of time to 1 second.

- **Tolerances:** Relative tolerance measures the error relative to the size of each state, i.e., a percentage of the state's value. The default, 10^{-6}, means that the computed state is accurate to within 10^{-4} percent. Absolute tolerance, on the other hand, is a threshold error value. This tolerance represents the acceptable error as the value of the measured state approaches zero. Its default value is 10^{-4}.

- **Maximum Step-size:** The Maximum step-size parameter controls the largest integration step the solver can take. The default value is 100001 (inf). This option is important to limit the integration steps, in order to prevent the solver from taking large step-sizes and missing two close zero-crossings.

- **Maximum Integration Time Interval:** The maximum time interval for each call to solver, it must be reduced if the error message ”too many calls” is encountered (error -1).

- **Tolerance on Time:** The smallest time interval for which the numerical solver is used to update continuous-time states.

3.3 Scicos Block's Architecture

Each Scicos block is defined by two functions. The first one, which must be in the Scilab language, handles the iterations with the editor. This function specifies the geometry of block, number of inputs and outputs the block type, etc. It is also this function that handles user interface (updating block parameters and initialization the states). This function is referred to as the *interfacing function* [**?**]. The second function, which is called the *Computational function*, is normally written in C, but can also be in Scilab. This function defines the behavior of the block during the simulation, see Fog. 3.3.1.

Each block can only have one interfacing function, but it may have several computational functions. These functions, declared in the interfacing function, are selected by the user. During simulation, Scicos can only use one of the declared computational functions.

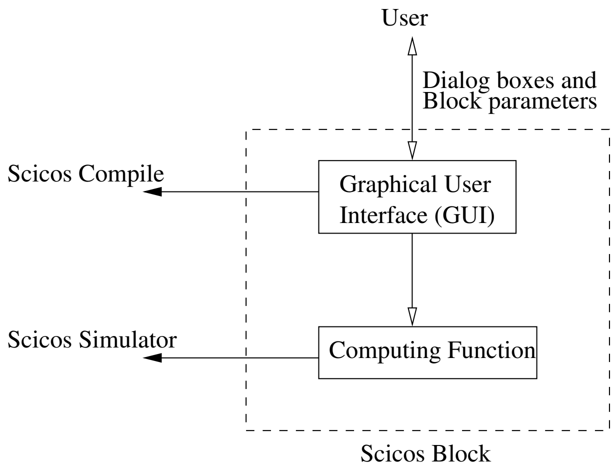


Figure 3.3.1: A Scicos Block.

3.3.1 Block Interfacing Function

Before explaining what this function exactly does, the block data structure should be introduced. The data structure of the block is a Scilab list that contains all the necessary information to compile and simulate the Scicos diagram. The `scicos_model()` function initializes this list. It is composed of the following elements:

- `sim`: which is a list by itself, containing the computation function name and the block-type (computation function calling sequence type).
- `state`: initial state vector for explicit blocks, and initial vector of state and state derivatives for block with DAE dynamics.
- `dstate`: initial discrete-time state vector.
- `in`: vector of block's input, e.g., `[3;2]`, means two inputs, the size of the first input is 3, and the size of the second one is 2.
- `out`: vector of block's output.
- `evtin`: vector of block's event input.
- `evtout`: vector of block's event output.
- `rpar`: vector of real valued parameters
- `ipar`: vector of integer valued parameters
- `blocktype`: 'c' means the computation function is written in C, and 'f' for Fortran.
- `firing`: initial event activation, if a block has some output event ports
- `nzero`: number of zero-crossing surfaces
- `nmode`: number of mode variables

- `label`: blocks' label
- `dep_ut`: input-output and time dependence. If block output can be directly affected by a change in input, the block is `dep_u=Ture`. It block needs the absolute simulation time, it is `dep_t=True`, it is also call always active, because it is always active via continuous-time.
- `equation`: for symbolic representation of equations (will be explained in the next chapters).

To set these values for each Scicos block, the Scicos editor uses the interfacing function. In addition to initializing these values, the interfacing function is used to draw the block in the Scicos window, define/change the block's icon, and to modify the block's parameters. As the interfacing function performs several tasks, in order to specify what it should do, Scicos calls it with an input flag `job`. The calling sequence is:

`[x,y,type]=block(job,arg1,arg2)`,

where `Job` can take several values such as `plot`, `getinputs`, `getoutputs`, `getorigin`, `set`, `define`, with each of these keywords, a special task can be done. Here, we only briefly explain `set`, and `define`.

- **job='define'**: The code fragment provides initialization of the block's data structure. The block type, number of inputs, outputs, states, etc should be set. In return, `x` is the block's data structure.
- **job='set'**: After clicking on the block, block is called with this flag, so this part of code can be used to open a dialog box for block parameters acquisition. Scilab `getvalue` function is used. The interfacing function should return the block's data structure of the block. In return, `x` is the new block's data structure.

3.3.2 Block Computational Function

The computational function is called by the simulator to perform the numerical computations such as computing the state derivatives, updating outputs, evaluating the zero crossing functions, etc. A Scicos block may contain continuous-time states, discrete-time states, differential equations, difference equations, root-finder functions, continuous-time signal input, input events, continuous-time signal outputs, event outputs, and a parameter set that may come from the simulator or belongs to the block. As the number of variables is relatively high, it will be cumbersome if they appear in the calling sequence of the computation function. Instead, the variables that belong to the block are collected in a structure, and the address of the structure is passed to the function. As a result we have a compact calling sequence:

void block_toto(**scicos_block** *blk, **int** flag)

where **scicos_block** is the block data structure, and **flag** variable indicates the computational task to be fulfilled by the block.

Computation Function Data-Structure

The **scicos_block** Data-Structure contains several variables and pointers.

```
typedef struct {
  int nevprt;     /* incoming discrete-time event coding (-1 for internal events)       */
  voidg funpt;    /* pointer to the computational function                              */
  int type;       /* type of interfacing function (calling sequence type)              */
  int scsptr;     /* not use for C computation function.                               */
  int nz;         /* number of discrete-time states                                    */
  double *z;      /* pointer to vector of discrete-time states                         */
  int nx;         /* number of continuous-time states                                  */
  double *x;      /* pointer to vector of continuous-time states                       */
  double *xd;     /* pointer to vector of derivatives of continuous-time states        */
  double *res;    /* pointer to vector of residuals of DAE                             */
  int nin;        /* number of inputs                                                  */
  int *insz;      /* pointer to  input                                                 */
  double **inptr; /* double pointer to block inputs                                    */
  int nout;       /* number of block outputs                                           */
  int *outsz;     /* pointer to  block outputs                                         */
  double **outptr;/* double pointer to outputs                                         */
  int nevout;     /* number of output events                                           */
  double *evout;  /* pointer to output events                                          */
  int nrpar;      /* number of block real-valued input parameters                      */
  double *rpar;   /* pointer to vector of block real-valued parameters.                */
  int nipar;      /* number of block integer-valued parameters                         */
  int *ipar;      /* pointer to vector of block integer-valued parameters.             */
  int ng;         /* number of zero-crossing functions                                 */
  double *g;      /* pointer to vector of zero-crossing functions.                     */
  int ztyp;       /* indicates if the block has zero-crossing function.                */
  int *jroot;     /* vector indicating the crossing zero and the crossing direction.   */
  char *label;    /* pointer to char, block's label                                    */
  void **work;    /* a pointer to workspace, used for dynamic memory allocation in block.*/
  int nmode;      /* number of modes.                                                  */
  int *mode;      /* pointer to vector of modes                                        */
  } scicos_block;
```

Block Dynamics Representation

The design of the Scicos block interface emphasizes a paradigm of accepting inputs and returning updates of states and required outputs. In continuous-time systems, a Scicos block may represent a set of ordinary differential equations (ODE) of the form:

$$\begin{aligned}\dot{x} &= f(x,u,t)\\ y &= g(x,u,t).\end{aligned}$$

In the above equations, u is the input vector, x is the state vector, y is the output vector, and t is the time. The Scicos block function is called from the simulator to compute $\dot{x}$ and y. The simulator sends x to the Scicos block function. This value should not be modified by the block. There are two exceptions: x can be modified during the discrete-time event calls and during initialization. If the dynamics of block is expressed with an ODE, the block `type` will be `4`.

A Scicos block's dynamics can also be represented as a differential algebraic equations (DAE). In the most general form, DAE systems are mathematically described by equations of the form:

$$\begin{aligned}0 &= F(\dot{x},x,u,t)\\ y &= G(\dot{x},x,u,t),\end{aligned}$$

where F is a vector-valued function with dimension equal to the number of states, and G is the output equation vector, with dimension equal to the number of outputs. In the this case, algebraic states of DAE should be declared, and initial value of differential states and guess values of algebraic states and derivatives of differential states may be given. It is interesting to note that, an ODE can be integrated with a DAE solver. So Scicos should transform the ODE to an DAE, with reformulating the ODE as:

$$\begin{aligned} 0 &= \dot{x} - f(x,u,t) \\ y &= g(x,u,t). \end{aligned}$$

Notice that the DAE integrator calculates both x and $\dot{x}$. The Scicos block function evaluates the implicit equation $F(\dot{x},x,u,t)$ with the supplied x and $\dot{x}$ values. The integrator uses F as the local residual error and attempts to maintain it below a certain threshold. If the dynamics of the block are expressed with an DAE, the block `type` will be `10004`.

Simulation Time Variable

The absolute simulation time, has not been included in block data structure, since it is a global variable. And it is not included in the calling sequence, since it is used only if the continuous-time block needs the time variable explicitly. To access the current simulation time, the external function:

```
double get_scicos_time();
```

can be used in the block to obtain the current simulation time. To use this function this header file should be declared in the block computation file.

#include `<scicos/scicos_block.h>;`

Simulation Phase Indicator

A block can be called either by the numerical integrator or by the simulator due to a discrete-time event. Sometimes we need to know who calls the block, either integrator, or simulator. This global variable has value `2` If the call is made in the numerical integration, and it will be `1` if it is due to an event. To obtain the value of this variable, the external function:

```
int get_phase_simulation();
```

is called. The need for introducing this variable will be discussed in succeeding chapters.

Continuous-time States

During integration, the numerical solver needs the value of state derivatives for explicit systems, and value of residuals for implicit systems. Whenever the solver request this value, the blocks that contain continuous-time states are called with `flag=0`. During this call, the computational function should compute either the state derivative or residuals, form the block data structure. and put it either in `xd`, or `res` vectors.

Discrete-time States

In discrete-time systems, the block represents a set of difference equations of the form:

$$\begin{aligned} x_{k+1} &= f(x_k,u_k,t_k) \\ y_k &= g(x_k,u_k,t_k). \end{aligned}$$

Whenever an event activates the block, the simulator calls the block with `flag=2`. A Scicos block may have several input events that come from several discrete-time event sources. When a discrete-time event arrives, the activated event port has been coded in binary in `nevprt` variable. During this call, the computational function should decode the `nevprt` variable to know the activating event to perform the associated action or state update. The addresses of discrete-time and continuous-time state vectors are in the block's data structure.

Mode variable and Zero-Crossing Functions

If a dynamical system has several configurations, each one is defined with a mode. There is a close relationship between mode variables and zero-crossings functions. The mode variable is used to handle the discontinuities and provide a smooth dynamical system for the numerical solver. At the moment where the system configuration is changed, there is a discontinuity in continuous-time states or their derivatives. So the numerical solver should be called up to this moment. This moment is the time where a function, so called zero-crossing, crosses zero. Provided the zero-crossing function, the numerical solver finds the exact crossing times.

The zero-crossing functions can be computed and delivered to the solver, when the block is called with `flag=9`. The mode variables can be updated whenever the block is called with `flag=9` and `phase=1`.

If the zero-crossing function can also be used to trigger some internal actions. In this case, if a zero-crossing function crosses zero, the block is called with `flag=2` (since this is an event) and `nevprt=-1` (to distinguish it from external events).

Algebraic vs Differential States Declaration

If the DASKR solver does a cold-restart, it normally requires the consistent initial condition to start the integration. If x_d and x_a denote the differential and the algebraic variables respectively, DASKR can be called to obtain the initial condition. To inform DASKR about the initial conditions computing option, INFO(11) should be set to:

- INFO(11) = 0: If provided initial conditions are consistent
- INFO(11) = 1: If initial conditions should be calculated with given x_d, and computing x_a and $\dot{x}_d$,
- INFO(11) = 2: If initial conditions should be calculated with given $\dot{x}_d$ and computing x_d x_a. This case has not been implemented in Scicos.

A block whose behavior is characterized by a DAE should provide a vector to identify the differential and the algebraic components of the DAE give in its `flag=0` code part. In Scicos a `property` vector is used to carry this information. This vector is filled in `flag=7` as follows:

`properties[i]=+1`, if `x[i]` is a differential variable
`properties[i]=-1`, if `x[i]` is an algebraic variable.

Then function

void `set_pointer_xproperty(int* pointer);`

is called to set a global vector in simulator holding overall algebraic and differential information.

Block Initialization

When the first option for initial value calculation is selected, the initial value of differential variables should be given in the `flag=4` part of the computing function of the block. In this case the given values for algebraic variables are considered as guess values during consistent initial computing. In complicated DAEs we need, however, to provide a precise enough solution to help DASKR find the consistent initial conditions.

At the beginning of simulation, the Scicos simulator calls all the blocks with `flag=4` to initialize them. Then the continuous-time and discrete-time states can be reinitialized (since they have been already initialized in the interfacing function). In addition, if the block should use an external file, it can open it during this block call. And finally, if the block needs extra memory, it can dynamically allocate it here.

Block Job Termination

At the end of simulation, the user stop request, or in case of error, the Scicos simulator calls all blocks with `flag=5`, and then exits. During this call, the opened files can be closed, and the dynamically allocated memories can be freed.

Block Regular Output Updating

Whenever the simulator needs the block outputs, it calls the block with `flag=1`. During this call, the computation function should update the output registers `**outptr`, based on current time, states, inputs, etc, existing in the block data structure.

Block Output Event Updating

A block may generate events, it may be a time-event (like a `event-clock generator`), or it may be s state-event (like a `zero-crossing` block). The events can only be generated with time delay greater or equal than zero. The generated events can also be programed synchronously (but with positive delay) with the incoming event. The outgoing event can be programed whenever the block is called with `flag=3`.

Jacobian Matrix Evaluation

The Jacobian matrix is needed by the numerical solver. It can be either calculated numerically by DASSL or be given by the user. In the later case, if the dynamical system is a DAE, the block is called with `flag=10` to read its Jacobian matrix. The computation function needs also a scalar, that the solver provides. This scalar is a global variable and can be accessed via an external function call, i.e.,

```
double Get_Jacobian_parameter(void);
```

The other usages of this scaler is used to improving the integration convergence. That will be discussed later.

3.3.3 Example 1: Finite State Machine Block

A finite state machine (FSM) or finite automaton is a model of behavior composed of states, transitions, and actions.

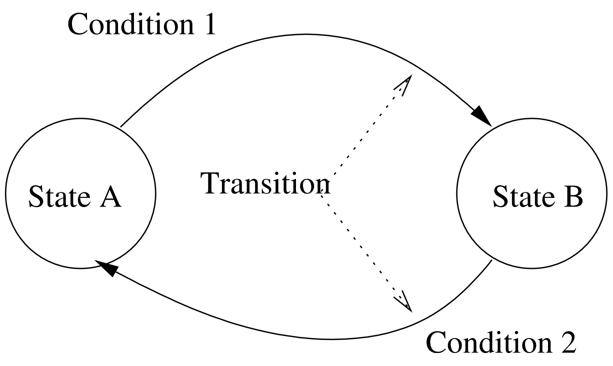


Figure 3.3.2: State flow diagram of sticky mass.

A state stores information about the past, i.e., it reflects the input changes from the systems start to the present moment. A transition indicates a state change and is described by a condition that would need to be fulfilled to enable the transition. An action is a description of an activity that is to be performed at a given moment. There are several action types:

- **Entry action:** execute the action when entering the state
- **Exit action:** execute the action when exiting the state
- **Input action:** execute the action depending on present state and input conditions
- **Transition action:** execute the action when performing a certain transition

A FSM can be represented using a state diagram (or state transition diagram) as in Fig. 3.3.2. Besides this, several state transition table types are used. The most common representation is shown below: the combination of current state (B) and condition (Y) shows the next state (C). The complete action information can be added only using footnotes. An FSM definition including the full action information is possible using state tables.

Current State/Condition	State A	State B	State C
Condition X	...	State A	State B
Condition Y	State C	...	...
Condition Z	...	State C	State A

Finite state machines are very widely used in modeling of application behavior, design of hardware digital systems, software engineering, and the study of computation and languages. As an example, we present the implementation of a sticky point masses system, taken from [?]. This sticky point mass is a simple hybrid automaton system. There are two point masses on a frictionless table with two springs attaching them to fixed walls. Given initial positions other than the equilibrium points, the point masses oscillate. The distance between the two walls are close enough that the two point masses may collide. The point masses are sticky, i.e., when they collide, they stick together and become one point mass with two springs attached to it. We also assume that the stickiness decays exponentially after the collision, such that eventually the pulling force between the two springs is big enough to pull the point masses apart. This separation gives the two point masses a new set of initial positions, and they oscillate freely until they collide again. The system is a finite state machine with two modes, separated and together. In the separated state, there are two differential equations modeling two independently oscillating point masses. There is also an event detection mechanisms, implemented by subtracting one position from another and comparing the result to zero. If the positions are equal, within a certain accuracy, then the two point masses collide, and a collision event is generated. This event will trigger a transition from the separated state to the together state. And the actions on the transition set the velocity of the stuck point mass based on Law of Conservation of Momentum.

In the together state, there is one differential equation modeling the stuck point masses, and another first order differential equation modeling the exponentially decaying stickiness. There is another expression computing the pulling force between the two springs. As for the moment the number of differential equations in Scicos should remain the same, the fourth differential equation is just added to satisfy this criteria. The guard condition from the together state to the separated state compares the pulling force to the stickiness. If the pulling force is bigger than the stickiness, then the transition is taken. The velocities of the two separated point masses equal to their velocities before the separation [**?**].

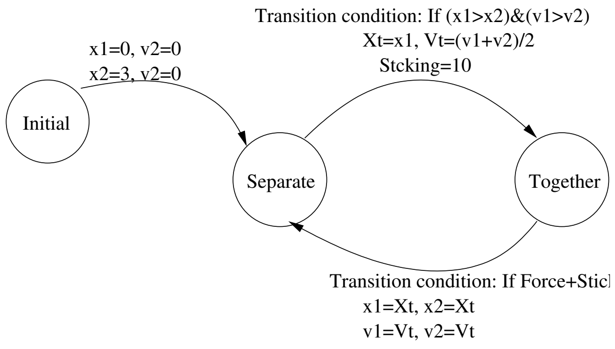


Figure 3.3.3: State flow diagram of sticky mass.

In Separate mode the system differential equation set is

$$\begin{aligned} \dot{x}_0 &= x_1 \\ \dot{x}_1 &= 1 - x_0 \\ \dot{x}_2 &= x_3 \\ \dot{x}_3 &= 4 - 2x_2, \end{aligned}$$

and in stuck mode is

$$\begin{aligned} \dot{x}_0 &= x_1 \\ \dot{x}_1 &= (5 - 3x_0)/2 \\ \dot{x}_2 &= -x_2 \\ \dot{x}_3 &= 0. \end{aligned}$$

The system goes from separated mode into stuck mode if the condition

$$(x_0 > x_2) \text{ and } (x_1 > x_3)$$

is fulfilled. Then the system goes to stuck mode and the system is initialized, i.e.,

$$\begin{aligned} x_0 &= x_0 \\ x_1 &= \frac{x_1 + x_3}{2} \\ x_2 &= 10. \end{aligned}$$

The system goes back into separated mode if

$$x_2 + x_0 - 3 < 0,$$

and then

$$\begin{aligned} x_0 &= x_0 \\ x_1 &= x_1 \\ x_2 &= x_2 \\ x_3 &= x_3. \end{aligned}$$

To implement this system we use a discrete-time state, indicating the system configuration (mode), four continuous-time states, and three zero crossing functions. Here is the interfacing functions of a Scicos block to simulate this hybrid automaton system.

```
AUTOM.sci:

function [x,y,typ]=AUTOM(job,arg1,arg2)
//Absolute value block GUI.
// Copyright INRIA
x=[];y=[];typ=[];
select job
case 'plot' then
  standard_draw(arg1)
case 'getinputs' then
  [x,y,typ]=standard_inputs(arg1)
case 'getoutputs' then
  [x,y,typ]=standard_outputs(arg1)
case 'getorigin' then
  [x,y]=standard_origin(arg1)
case 'set' then
  x=arg1;
  case 'define' then
  //
  nstate=4
  nin=0
  nout=2
  nmode=0
  nzero=3
  ndsate=1
  model=scicos_model()
  model.sim=list('autom',10004)
  model.in=ones(nin,1)
  model.nzcross=nzero
  model.nmode=nmode
  model.state=ones(nstate*2,1)
  model.dstate=ndsate
  model.out=nout
  model.blocktype='c'
  model.dep_ut=[%f %t]
  gr_i='xstringb(orig(1),orig(2),''Automaton'',sz(1),sz(2),''fill'')'
  x=standard_define([2 2],model,[],gr_i)
end
endfunction
```

And the computation function is

```
autom.c:
#include <math.h>
#include <scicos/scicos_block.h>
void autom(scicos_block *block, int flag)
{
  double  rpar[5];
  double *x = block->x;
  double *z = block->z;
  double *xd = block->xd;
  double *y=block->outptr[0];
  double *u=block->inptr[0];
  double *g=block->g;
  double *res=block->res;
  int    *mode = block->mode;
  int    *jroot=block->jroot;
  int     nevprt=block->nevprt;
  int     property[5];
  int     phase;
  double  t;
  t=get_scicos_time();
  //-------------------------------------------------------
  if (flag == 0){
    if(z[0]==1){
      res[0]=xd[0]-x[1];
      res[1]=xd[1]-1+x[0];
      res[2]=xd[2]-x[3];
      res[3]=xd[3]-4+2*x[2];
    }else if (z[0]==2){
      res[0]=xd[0]-x[1];
      res[1]=xd[1]-(5-3*x[0])/2;
      res[2]=xd[2]+x[2];
      res[3]=x[3];
    }
    //-------------------------------------------------------
  }else if (flag == 1) {
      y[0]=x[0];
    if(z[0]==1){
      y[1]=x[2];
    }else if (z[0]==2){
      y[1]=x[0];
    }
    //-------------------------------------------------------
  }else if (flag == 9) {
    if (z[0]==1){
      g[0]=(x[0]-x[2]);
      g[1]=(x[1]-x[3]);
      g[2]=0;
    }else if (z[0]==2){
      g[0]=0;
      g[1]=0;
      g[2]=x[2]+(-3+x[0]);
    }
    //-----------------------------------------------------
  }else if (flag == 7) {
    if(z[0]==1){
      property[0]=1;
      property[1]=1;
      property[2]=1;
      property[3]=1;
    }else if (z[0]==2){
      property[0]=1;
      property[1]=1;
      property[2]=1;
      property[3]=-1;
    }
    set_pointer_xproperty(property);
  }else if (flag ==2 & nevprt<0) {
    if (jroot[0]==1){
      x[0]=x[0];
      x[1]=(x[1]+x[3])/2;
      x[2]=10;
      z[0]=2;
    }else if(jroot[2]==-1){
      x[2]=x[0];
      x[3]=x[1];
      z[0]=1;
    }
  }else if (flag == 4) {
    x[0]=0;
    x[1]=0;
    x[2]=3;
    x[3]=0;
    x[4]=0;
    z[0]=1;
  }
}
```

The simulation result of the system has been scketched in Fig. 3.3.4.

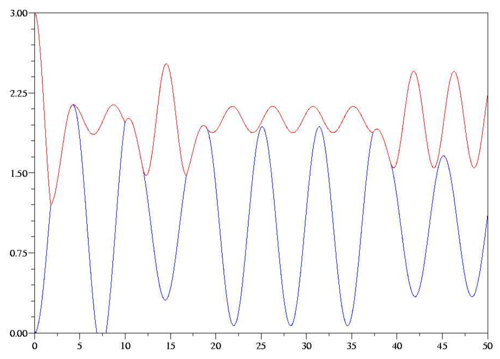


Figure 3.3.4: Simulation result of sticky mass hybrid automaton system.

3.3.4 Example 2: AMESim/Scicos interface block

AMESim[1].is a simulation software for the engineering systems. In AMESim, the user can build models of engineering systems by connecting blocks in a graphical environment. The blocks represent individual components of the system (see Fig. 3.3.5).

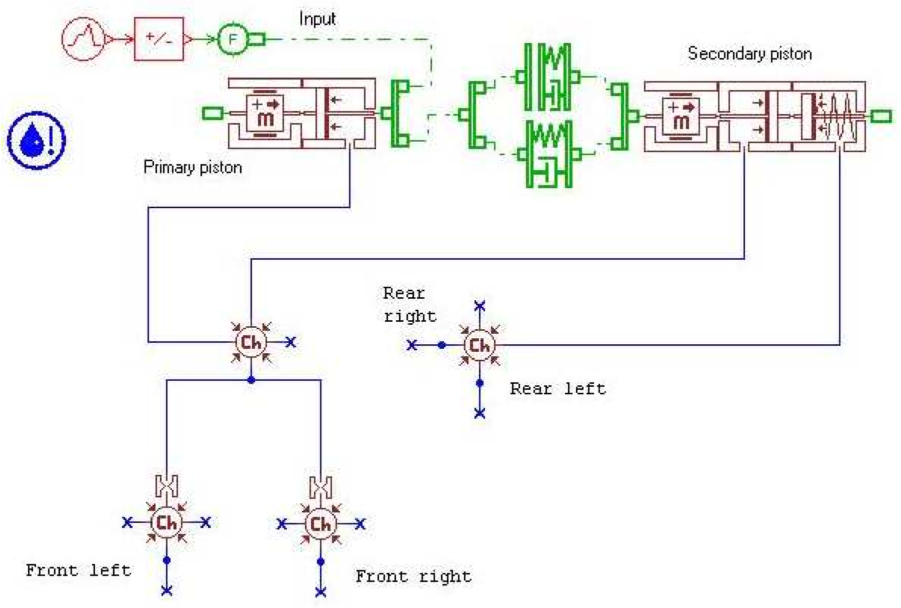


Figure 3.3.5: An engineering system using standard hydraulic and control components in AMESim.

The standard library of AMESim provides control and mechanical components and sub-models allowing the user to perform dynamic simulation of engineering systems. In addition, there are optional libraries such as the Hydraulic Component Design, the hydraulic resistance, pneumatic, thermal, thermal-hydraulic, cooling system, power train and filling libraries. Each

[1]This work was a part of METISSE project performed in INRIA

component contains a set of equations defining the dynamic behavior of the block. When the model is complete, a simulation of the system can be proceeded. During the simulation AMESim refers to the model of the system. The model is built up from the equations of each component within the system and the equations that connect the components.

In the past, the simulation software used for engineering design tended to be totally independent and incapable of communicating with one another. This is no longer acceptable since neither package can provide all the facilities needed. Modern practice is to provide interfaces between software to enable them to work together so that the user can obtain the best features of both. In this section we will explain how a model which is developed in AMESim can be used in Scicos. This gives the user access to powerful controller design features, optimization facilities, and power spectral analysis, etc. of Scilab and Scicos softwares.

The AMESim/Scicos interface enables the user to construct a model of a subsystem in AMESim and to convert it to a Scicos block. The Scicos block can then be imported into Scicos and used within a Scicos block diagram system, just like any other Scicos block. The interface is designed so that the user can continue to use many of the AMESim facilities while the is running in Scicos. In particular the user can change the parameters of the model within AMESim in the normal way, examine the results within by creating plots just as if they were produced in a regular AMESim run. Normally, the user will have AMESim and Scicos running simultaneously so that the user can use the full facilities of both packages. As an example, consider the model depicted in Fig. 3.3.6.

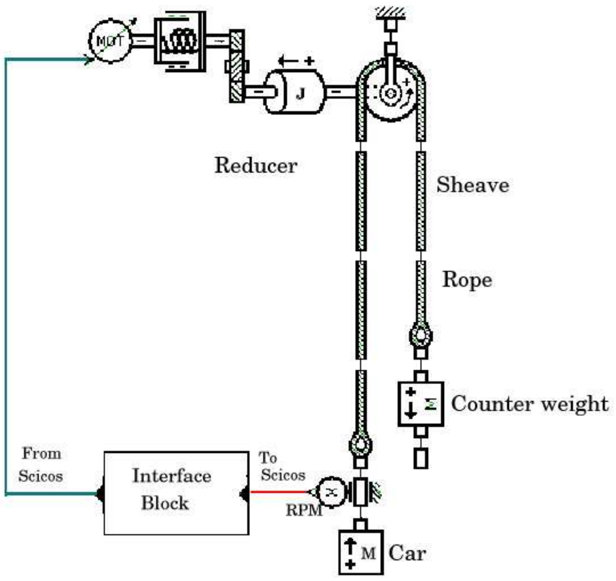


Figure 3.3.6: An elevator system modeled using AMESim.

This is an elevator system that has been built in AMESim with an interface blocks to establish two links to/from Scicos. This nonlinear system is then imported into Scicos and can be run entirely in Scicos (see Fig. 3.3.7). The block `AMEs Scicos` in Fig. 3.3.7 is the block that contains the elevator's model.

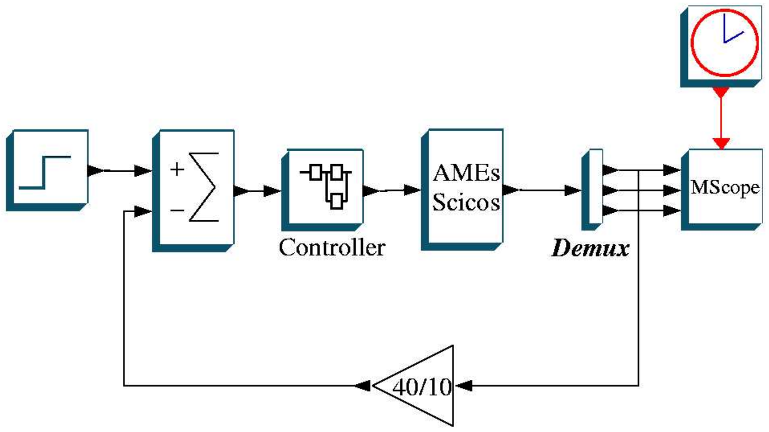


Figure 3.3.7: A Scicos model containing an AMESim block imported from AMESim.

Implementation

When an AMESim model, containing an interface block, is compiled in AMESim, several files are generated. For an AMESim model that is called `NAME.ame`, these file will be generated: `libNAME_.dll`, `NAME_.data`, `NAME_.c`, `NAME_.la`, `NAME_.make`, `NAME_.param`, `NAME_.sim`, `NAME_.var`, `NAME_.ssf`, `NAME_.state`, and `NAME_.lock`. `NAME_.data` contains the initial value of states of the model, `libNAME_.dll` contains all model information, and `NAME.c` contains the C code that is used to build the interface imported AMESim block in Scicos.

The first step to import the AMESim model into Scicos is building a Scicos block with appropriate parameters such as number of inputs or states. These information are stored in `libNAME_.dll` (in Windows) or `libNAME_.so` (in Linux). To get these information, this library files should be dynamically linked with Scilab (using `c_link` and `zilib_for_link` functions in Scilab). Then the function `NAME_info`, defined in the linked library file) is evoked to obtain the block parameters. Here is the interface function of the AMEs/Scicos block.

```
function [x,y,typ]=BLOCKAME(job,arg1,arg2)
 // Copyright INRIA
 x=[];y=[];typ=[];
 select job
 case 'plot' then
   standard_draw(arg1)
 case 'getinputs' then
   [x,y,typ]=standard_inputs(arg1)
 case 'getoutputs' then
   [x,y,typ]=standard_outputs(arg1)
 case 'getorigin' then
   [x,y]=standard_origin(arg1)
 case 'set' then
 //========================================================
   x=arg1;
   graphics=arg1.graphics;
   exprs=graphics.exprs
```

```
  model=arg1.model;
  while %t do
    Path1=exprs(1)
    [ok,Path1,Fname1,Ponof1,Pint1]=getvalue('Set AMESim block parameters',..
        ['Path name';'File name';'print yes=1/no=0';'print interval'],..
      list('str',1,'str',1,'vec',1,'vec',1),exprs)
    if ~ok then break;end //user cancel modification
    if part(Path1,length(Path1)) ~= '\' then
      Path1=Path1+'\';
    end
    Ponof1=int(Ponof1)
    Pint1=int(Pint1)
    Fname1=stripblanks(Fname1)
    Path1=stripblanks(Path1)
    mess=[]
    if Ponof1<=0 then
      mess=[mess;'Print on/off should be 0/1';' ']
     ok=%f
    end
    if ~ok then
     message(['Some specified values are inconsistent:';' ';mess])
     break;
    end
    chdir (Path1);
    flag  = "c";libs  = [];
    files = [Path1+Fname1+'.o';];
    libs=[];
    [a,b]=c_link(Fname1); while a;  ulink(b);[a,b]=c_link(Fname1);end
    zilib_for_link(Fname1,files,libs,flag);
    xlink=link('lib'+Fname1+'.dll',[Fname1;Fname1+'info'],'c');
    nx=0; nin=0; nout=0; nvar=0;
    [nx,nin,nout,nvar]=call(Fname1+'info',"out",[1,1],1,..
                        'i',[1,1],2,'i',[1,1],3,'i',[1,1],4,'i');
    exprs(1)=Path1;
    exprs(2)=Fname1;
    exprs(3)=sci2exp(Ponof1);
    exprs(4)=sci2exp(Pint1);
    model.sim(1)=Fname1
    ipar=[Ponof1;Pint1]
    model.in=nin
    model.out=nout;
    model.ipar=ipar
    model.state=zeros(2*nx,1)
    model.dep_ut=[%f %t]
    graphics.exprs=exprs;
    x.graphics=graphics;x.model=model
    break;
  end
//===========================================================
case 'define' then
  Fname='slink_';
  Path='C:\';
  nx=1; nin=1; nout=1; nvar=0;
```

```
   Ponof=1;Pint=0;
   model=scicos_model()
   model.sim=list(Fname,10004)
   model.in=[nin]
   model.out=[nout];
   model.rpar=[Ponof;Pint];
   model.ipar=1
   model.nzcross=1
   model.blocktype='c'
   model.dep_ut=[%f %t]
   model.state=zeros(2*nx,1)
   exprs(1)=Path;
   exprs(2)=Fname;
   exprs(3)=sci2exp(Ponof);
   exprs(4)=sci2exp(Pint);
   gr_i=['txt=[''AMEs'';''Scicos''];';'xstringb(orig(1),orig(2),txt,sz(1),sz(2),..
         ''fill'');']
   x=standard_define([2 3],model, exprs,gr_i)
 end
endfunction
```

After building the interface function, the computation function of the block should be constructed. AMESim provides a template file (`*.etemp`) to build an interface with other softwares. We built the `scicos.etemp` file. This template file is used to generate the `NAME_.c` file that contains the computation function of the interface block. In the computation function we can call several functions to get access to the nonlinear equations of the model, e.g., the function
`funcval(&t, x, u, res, y, &flagx);`
receives the current value of states and inputs of the model, and returns the derivatives and new outputs of the model.

In order to generate the residuals of the model in `flag=0` of the Scicos block, the `funcval` function is called and in return the derivatives are stored in `res` vector. Then the residuals of the model is computed with `res[i]=res[i]-xd[i]`.

The `funcval` function updates also the outputs of the model. Therefore this function can be used to update the outputs of the Scicos block in `flag=1`.

Generating the zero-crossing functions of the AMESim block is a little tricky. The AMESim block can only report that a zero crossing has happened or not. This report is given each time that the function `funcval` is called via `flagx` variable. To create a pseudo zero-crossing surface in Scicos block, the output variable `flagx` is monitored in `flag=2` and `phase-2` (i.e., when the solver calls the block). If there is any change, we change the sign of `zerocrossingflag`. `zerocrossingflag` can be used in `flag=9` to localize the zero-crossings. Here is the complete computation function of the Scicos interface block.

```
void  NAME(scicos_block *block,int flag)
{
  double *rpar = block->rpar;
  int    *ipar = block->ipar;
  double *z = block->z;
  double *x = block->x;
  double nx = block->nx;
double *xd = block->xd;
double *res = block->res;
double *y = block->outptr[0];
int   *ysize = block->outsz;
int   nout = block->nout;
int   nin = block->nin;
double *u = block->inptr[0];
int   *usize=block->insz;
double *g = block->g;
int nevprt = block->nevprt;
int i, flagx;
double t, AMEoutput;
int    LastWasReset;

t =  get_scicos_time();
  //------------------------------------------
if (flag==0){
  //------------------------------------------
  if(first_call){
     DisplayMessage("first_call in through flag.0 - this is BAD \n");
  }else{
    SetTimeAtThisStep(t);
    flagx = 1;// variable step solver
    funcval(&t, x, u, res, y, &flagx);
    for(i=0;i<nx;i++)  res[i]-=xd[i];
  }
  //-------------------------------------------
}else if (flag==1){
  //-------------------------------------------
    LastWasReset = ipar[0];
  if(first_call){
    flagx = 0;
    funcval(&t, x, u, res, y, &flagx);
    SetTimeAtLastStep(t);
  }else{
    if (get_phase_simulation()==1) {
           if(LastWasReset==1){
               flagx = 0; //last was restart -allow Amesim model to reset states
               }else{
               flagx = 2;  // Allow discontinuities to be reported
           }
           ClearStatesChanged();
           funcval(&t, x, u, res, y, &flagx);
           if( (flagx > 2) || AreStatesChanged() ) do_cold_restart();
           if (flagx > 2) LastWasReset = 1;
    }else{
           flagx = 2; // We need the discontinuities here for the zerocrossing
           funcval(&t, x, u, res, y, &flagx);
           if(flagx > 2){
                zerocrossingflag = 0.5;
           }else{
                zerocrossingflag = -0.5;
           }
```

```
      }
    }
    //--------------------------------------------
  }else if (flag==2 && nevprt==1){
    //--------------------------------------------
    FinalTime = t; // Will this cause problems ?
    SetIsPrinting();
    flagx = 1;   // This is a "print call" - flag =1 seems resonable
    funcval(&t, x, u, res, y, &flagx);
    ClearIsPrinting();
    if(fd_results != -1){
      if(PrintInt > 0.0) {
             if(t < NextPrint) return; // Do not save into the result file yet
             NextPrint = t + PrintInt;
      }
      // Write time & variables to slink_.results.
      OutputResults(t);
    }
    //--------------------------------------------
  } else if (flag==9){
    g[0]=zerocrossingflag;
  } else if (flag==5){

    if(fd_results != -1){
      OutputResults(FinalTime);
      close(fd_results);
    }
    fd_results = -1;
    AmeCallAtEnd(0);

  } else if (flag==4){
    first_call = 1;
    points = position = 0;
    fd_results = -1;//output_result_file pointer
    ConstructFileNames("NAME");
    //----------------
    LastWasReset = ipar[0];
    Input(x); //reading from Data file
    Initialize(x);   // calling blocks via Xin_ routine to initialize

    AMEoutput =rpar[0]; //print yes/no
    AMEoutput=0;//no output_result_file
    PrintInt  =rpar[1];   //print interval
    NextPrint = 0.0;

    if(AMEoutput == 0.0) {
      outoff = 1;
    }else{
      outoff = 0;
    }

    if(fd_results == -1 && AMEoutput != 0.0){
      AmeReadFile(&t, x);
    }

    LastWasReset= 0;
  }
}
```

Chapter 4

Numerical Solvers

Scicos is a software environment for modeling and simulation of dynamical systems. The underlying formalism in Scicos allows for modeling very general hybrid dynamical systems, i.e., systems including continuous-time, discrete-time and event based behaviors. For the continuous-time part, Scicos uses standard ODE/DAE solvers which are controlled by the Scicos simulator which also handles the discrete-time and event driven parts of the model. This control includes, in particular, proper management of the re-initializations and zero-crossings.

Scicos includes a graphical editor for constructing models by interconnecting blocks, representing predefined or user defined functions, a compiler, a simulator, and some code generation facilities. A number of blocks are available in various Scicos toolboxes. Scicos can be used to model and simulate a wide range of dynamical systems. Modeling of physical system often ends up to an ODE or a DAE. To integrate a continuous-time system, a numerical solver should be used. Scicos uses two numerical solvers, one for ODE's and one for DAEs. But using a solver in Scicos is not just a matter of interfacing the code. ODE and DAE solvers have many control parameters which should be used to optimize the simulation in different situations. Indeed in the hybrid context, controlling the solver, which should be done automatically and should remain transparent to the user, is a complex matter due to the interactions between the continuous-time dynamics and the rest of the system. Clearly the discrete-time part affects the continuous-time part through simple connections. A continuous-time system driven by a digital controller is a typical example. At sampling times, the value of the control jumps creating discontinuities in the ODE/DAE. In such situations for example, the simulator must make sure that the ODE/DAE solver does a cold restart otherwise, since variable step ODE/DAE solvers require continuity, it may fail or give erroneous results.

But the interaction goes also the other way around: the continuous-time dynamics can generate an event and affect the discrete-time part of the system. For example if the level of water in a tank is modeled as the state of a differential equation, then in a Scicos model we have the possibility of generating an event when the level falls below a given value. In this case, the ODE/DAE solver must know when to stop and generate a signal. This is done using the *zero-crossing* facility available in most ODE/DAE solvers. Using this facility however, in the hybrid context like in Scicos, is not straightforward and some modifications had to be made in these codes.

Most of the problems in using standard solvers in a hybrid environment are common to both ODE and DAE solvers. However, there is an additional difficulty with the DAE case:

the problem of re-initialization and finding consistent initial conditions.

This chapter presents some of the problems and solutions implemented in Scicos, and discusses in particular modifications made in the ODE/DAE solvers.

4.1 ODE/ DAE Solvers

Scicos uses two numerical solvers: LSODAR [?] and DASKR [?, ?]. LSODAR is an ODE solver which is used when the Scicos diagram does not contain *implicit* blocks[1] which means that the continuous-time dynamics of the model can be represented as follows

$$\dot{x} = F(t, x),$$

where t is the time, x the state, and $\dot{x}$ the derivative of x with respect to time. This ODE system is solved by calling the ODE solver repeatedly. At each call the solver computes the solution over a given period of time. The function F may change, due to discrete-time dynamics (events), from one call to the other requiring cold restarts.

The dynamical model of certain physical systems cannot be described in ODE form, they are described by differential-algebraic equations (DAE). The dynamics of the system can be represented either in fully implicit form, i.e.,

$$\begin{aligned} 0 &= f_1(\dot{x}, x, t, u) \\ y &= f_2(\dot{x}, x, t, u), \end{aligned}$$

where u, x and y represent respectively the system's inputs, the state vector and the outputs. We assume that DAE is of index 1. To solve such a system, Scicos calls DASKR repeatedly, very much the same way as it does in the explicit case with LSODAR. From the user point of view, the main difference between the ODE and DAE here is that in the DAE case, to start a simulation, initial values for both x and $\dot{x}$ must be provided whereas in the ODE case only x needs to be furnished. This causes difficult and interesting problems, in particular, in the hybrid context.

LSODAR and DASKR are very similar in many respects and in particular in terms of control options. They are also very alike from an implementation point of view. This has been an important factor in selecting these two solvers for Scicos. Both LSODAR and DASKR solvers are variable-step solvers which means that they vary their step-sizes during the simulation. By reducing the step-size to increase accuracy when the state is changing rapidly and increasing the step-size to avoid taking unnecessary steps when the state is changing slowly, they can improve speed without compromising accuracy. That of course creates some difficulties in terms of their use in Scicos because the solver can take a step forward in time and then later step back which means that the evaluation times do not constitute an increasing sequence. But the advantages are overwhelming and worth the extra effort to make it work in harmony with the discrete-time part of the system. The variable step behavior is also unavoidable because of the zero-crossing detection feature which is needed in Scicos. Indeed, in order to accurately localize a zero, the solver needs to make repeated calls around it.

Another feature of these solvers is the ability to compute numerically the Jacobian matrix. Of course, they can also use a user-provided routine for evaluating the matrix. But in

[1]Implicit blocks that will be explained in detail in chapter 6, is a block that its physical behavior is described by a DAE

Scicos, the lack of symbolic representation of all block's dynamics[2] makes it impossible to evaluate symbolically the Jacobian matrix. So the solvers have to compute it numerically. For the sake of efficiency, the Jacobian is not evaluated at every time step. This useful and important feature poses some problems in the re-initialization phase of DAE system which will be discussed in subsequent sections.

4.2 DAE [Re]-Initialization Problem in Solvers

4.3 Event Detection

Mathematical models of many physical systems are described naturally by a system of differential equations. Simulation of these models is a little complicated in the presence of discontinuities, which occur in the form of discrete-time changes of system equations. Solution of ODE/DAEs containing discontinuities can be formulated as a combined discrete-time/continuous-time system or hybrid system. The ability to detect the time when a discontinuity occurs, or more precisely, the time when a function crosses some given value (by default considered zero) is of capital importance in the hybrid environment. When a zero-crossing occurs, the solver stops the simulation, pinpoints the exact crossing time and returns.

Zero-crossings are not just used to generate events in order to communicate with the discrete-time part of the model. In most cases zero-crossings are used to properly control the solver to simulate non-smooth continuous-time dynamics. Consider a continuous-time model in which the *absolute value* function is used, e.g.,

$$\begin{aligned}\dot{x} &= |u| \\ u &= \sin(t) \\ y &= x.\end{aligned}$$

This system can produce a non-smooth (derivative not continuous) signal even if the input is smooth. If the output affects a state derivative, it is important to halt the solver and do a cold restart at the point where the non-smoothness may appear. Of course, in this case it is the input value zero which corresponds to the point of non-smoothness. Although powerful, a few features were lacking in the zero-crossing mechanism of LSODAR and, in particular, DASRT (old version of DASKR) to make them operational in the Scicos environment. Two new features have now been implemented and some are already included in the official release of DASKR. These features/modifications will be described in the following sections.

4.3.1 Zero-Crossing as a Function of State Derivatives

In the integration of ODEs, it is reasonable to assume that the zero-crossing function cannot depend on state derivatives because any state derivative can be represented as a function of other states. But in the numerical simulation of DAEs, in a sense the state of the system contains both x and $\dot{x}$ so that it would be natural to allow the dependence of the zero-crossing function on $\dot{x}$. There are also physical situations where this would occur, for example, in simulations of a mechanical system with an acceleration based controller and an acceleration

[2]As we will see in chapter 6, only implicit blocks can provide their own Jacobian matrix, and this partial Jacobian can be used to compute the overall Jacobian

based event. This was not possible in DASRT/DASSL but this feature has been added to DASKR. Using DASKR we can solve systems of the form

$$0 = F(\dot{x}, x, t)$$
$$G_{zc}(\dot{x}, x, t) \qquad \text{Zero-cross function.}$$

However, there are some numerical problems which need to be addressed. Consider the above equation when the algebraic and differential equations are separated:

$$\dot{x}_d = f_1(x_d, x_a, t)$$
$$0 = f_2(x_d, x_a, t)$$
$$G_{zc}(\dot{x}_d, \dot{x}_a, x_d, x_a, t),$$

where x_a is an algebraic state and x_d is a differential state. For index-1 systems $\dot{x}_d$ can be rewritten as a function of state variables, and we will have

$$\dot{x}_d = f_1(x_d, x_a, t)$$
$$0 = f_2(x_d, x_a, t)$$
$$G'_{zc}(\dot{x}_a, x_d, x_a, t).$$

Note that introducing a new variable for eliminating $\dot{x}_a$ will increase the DAE index by one, i.e.,

$$\dot{x}_d = f_1(x_d, x_a, t)$$
$$\dot{x}_a = v$$
$$0 = f_2(x_d, x_a, t)$$
$$G'_{zc}(v, x_d, x_a, t),$$

is a DAE index-2 system, and so an unsolvable system for certain numerical solvers.

For this system, before the beginning of integration and just after a cold-restart, the derivative of x_a is not known. It is estimated only after the start of integration. Most of the time, the estimates will not be accurate enough and the solver would be more likely to obtain multiple roots, see Fig. 4.3.1. If after the root finding, the solver is [hot]-restarted, since the solver uses the history to estimate the solution, there will be no problem.

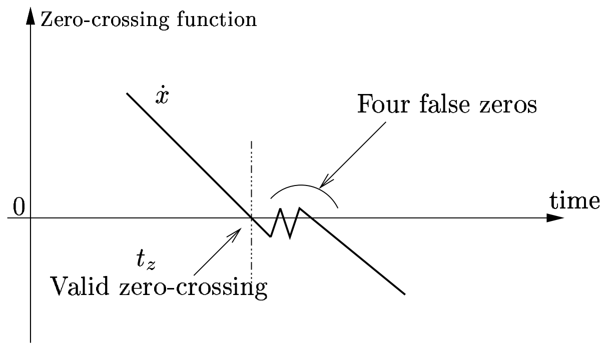


Figure 4.3.1: Zero sticking.

In general, DASKR cannot be trusted to do a good job of finding roots at (or very near) the initial time/after a cold-restart, especially if the root function(s) depend on a derivative of an algebraic state. This is because DASKR has very little history information about the solution when it is searching for such a root. In this case, even if it has an accurate x_a and $\dot{x}_a$ at $t = 0$, it has no way to compute $\dot{x}_a$ accurately for $t > 0$, and the value it uses is the same as the input initial value. That is, when it does its initial root finding, before taking any steps, it uses $\dot{x}_a = \dot{x}_a(0)$ identically, where $\dot{x}_a(0)$ is the input initial guess for $\dot{x}_a$. Consider the example

$$\begin{aligned} x &= \cos(x) \\ y &= \begin{cases} x & \text{if } (\dot{x} > 0) \\ -x & \text{if } (\dot{x} \le 0) \end{cases} \\ G_{zc} &= \dot{x} \ \textit{zero-crossing function.} \end{aligned}$$

Scicos is designed to simulate hybrid systems. That means that zero-crossing surfaces will be a feature of many of the problems that it is asked to solve. Scicos tries to not do a cold restart whenever possible. However, depending on the particular problem, different parts of the prior solution are most useful, and in some cases, a cold restart is necessary. The investigation of how best to incorporate cold, warm, and hot restarts, and how to make these capabilities available to the experienced user without complicating things for the novice users is an ongoing investigation.

4.3.2 Zero-Crossing Direction

In some applications of zero-crossings, it is not enough to have the time that a surface crosses zero; it is also important to know in which direction the crossing has occurred.

Consider the bouncing ball system. In the bouncing ball we need the zero-crossing direction as well. When the ball hits ground, the solver should generate a zero-crossing event to inform the simulator. This event cannot be detected unless the ball's altitude become negative, see Fig. 4.3.2. So in the next restart, the position would slightly negative. Starting from a negative altitude and a positive velocity, would result in another zero crossing. This zero crossing should be ignored, a simple solution would be looking at the zero-crossing direction and eliminating the `- to +` crossing direction.

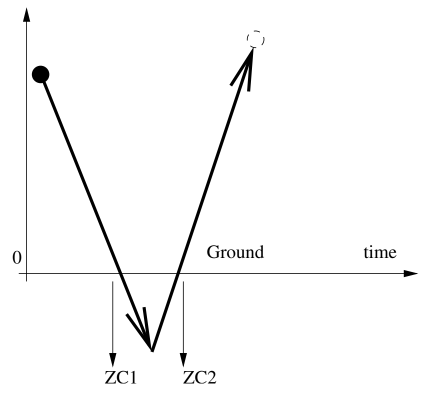


Figure 4.3.2: Bouncing ball.

Here is another example that shows the importance of crossing direction in a simulation. Orbitode is a standard test problem for non-stiff solvers that traces the path of a spaceship traveling around the moon and returning to the earth [?]. In fact, it is a restricted three-body problem with the following system of equations:

$$\begin{aligned}
\dot{x}_1 &= x_3 \\
\dot{x}_2 &= x_4 \\
\dot{x}_3 &= 2\,x_4 + x_1 - (1-\alpha)\,\frac{x_1+\alpha}{r_1} - \alpha\,\frac{x_1 - 1 + \alpha}{r_2} \\
\dot{x}_4 &= -2\,x_3 + x_2 - (1-\alpha)\,\frac{x_2}{r_1} - \alpha\,\frac{x_2}{r_2} \\
r_1 &= ((x_1+k)^2 + x_2^2)^{\frac{3}{2}} \\
r_2 &= ((x_1 - 1 + k)^2 + x_2^2)^{\frac{3}{2}} \\
\alpha &= 0.0121286.
\end{aligned}$$

The x_1 and x_2 variables are the x and y coordinates of the spaceship. The trajectory for some initial conditions that produce a periodic turning has been depicted in Fig. 4.3.3.

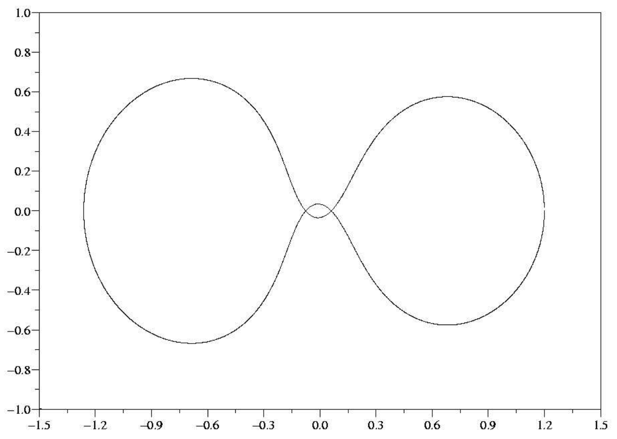


Figure 4.3.3: Spaceship trajectory.

Suppose that we need to know precisely when the spaceship has the maximum distance from the origin. For that, the *Distance* function should be constructed. When this function is at a maximum, the spaceship is in the desired position. By differentiating the *Distance* function we can obtain a zero-crossing function G_{zc} to be used as an event generator.

$$\begin{aligned}
\text{Distance} &= \sqrt{(x_1 - x_{10})^2 + (x_2 - x_{20})^2} \\
G_{zc} &= (x_1 - x_{10})x_3 + (x_2 - x_{20})x_4.
\end{aligned}$$

The problem with this zero-crossing function is the fact that the zero-crossing function passes the zero point both at the origin and at the maximum distance point (see fig. 4.3.4.

This example shows the need to specify the crossing direction is of great importance for certain applications. In this example it is used to distinguish the origin and the maximum distance points.

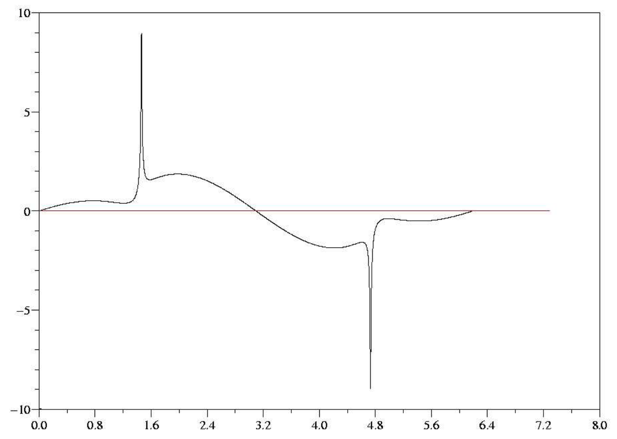


Figure 4.3.4: Zero-crossing function of derivative of spaceship's distance from origin.

Unfortunately LSODAR and DASKR did not provide this information. In the previous versions of Scicos where this feature was not available in the solvers, the direction of the zero-crossing was computed by storing previous values of the zero-crossing functions. But this was very cumbersome and in some situations produced erroneous results. This problem has been solved by slightly modifying the codes without jeopardizing backward compatibility. In previous version of DASKR,whenever a root occurs, Integrations stops at the crossing point and reports by sending a `IDID=5`. To understand which root has crossed the zero, DASKR fills out the `Jroot` array on return with:

- `Jroot(i) = 1`; if Root(i) has crossed the zero
- `Jroot(i) = 0`; if not.

In the current version of DASKR, the information about crossing direction is coded in the sign of `Jroot`, i.e.,

- `Jroot(i) = 1`; if Root(i) has a root and changes from - to +.
- `Jroot(i) = 0`; No Crossing.
- `Jroot(i) = -1`; if Root(i) has a root and changes from + to -.

Implementation

In this section, we will explain the implementation of zero-crossing in DASKR. LSODAR uses nearly the same routines and procedures. DASKR uses two routines to detect the zero-crossings; *Drchek* and *Droots*. The role of the root-finder module (*Drchek* routine) is to determine if any of the zero-crossing functions changes the sign during the integration time. When the solver starts the integration, at the beginning of each integration the overall integration time is divided into several small time-intervals with variable or constant sizes, called integration steps. The BDF Integration method is used to advance the time on each integration step. *Drchek* reads all the zero-crossing functions and saves their values at the beginning of each integration step and compares them at the end of the integration steps. If there are any sign changes, *Drchek* calls the *Droots* routine to find the exact time of the earliest zero-crossing. If there are any crossings, the *Droots* routine will find and return the exact time (location) of the crossing. If more than one function has changed sign during the last integration step, *Droots* will find the left most crossing.

The numerical algorithm used to find a zero of a function is very simple: DASKR uses a Regula-Falsi (false position) algorithm. It has the advantage of converging for a large range of problems, and does not require the derivative of the function. All that it needs is the interpolating function that return the value of zero-crossing function at point between the end-points of the last integration-step. The following simplified listing shows a sketch of the Regula-Falsi algorithm.

```
  t1  = t(n)
  t2  = t(n+1)

1:f1=f(t1)

  f2=f(t2)

  tx  = min[t1- f1*(t2-t1)/(f2-f1)] for all zero crossing functions

  If (abs(t1-tx)<tol  or abs(t2-t1)>10*tol ) goto 2

  fx=f(tx)

  If f1*fx <0 then t2=tx
              else t1=tx
  goto 1

2: tz=t2
   if f1> f2 direction = -1
   if f1< f2 direction = +1
   if f1== 0 direction = -1
   if f2== 0 direction = +1
   end
```

In this listing the `t(n)` and `t(n+1)` are two end-points of the current integration step. To handle the multiple function case, we do one step of Regula-Falsi on the all functions in order to find (`tx`). With this (`tx`) value, a new interval `[t1,tx]` (or `[tx,t2]` depending on the sign of `f1*fx`) is considered to run another step. This process is repeated until the interval become small compared to the tolerances.

At the end of the procedure, when the search distance becomes small enough, the zero-crossing direction can be obtained based on the sign of `t1` and `t2`, see Fig. 4.3.5.

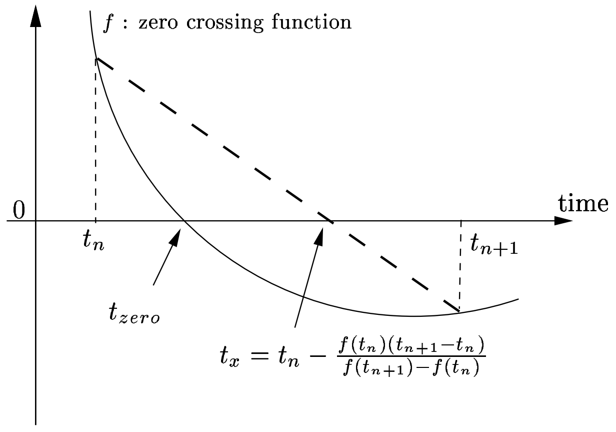

f : zero crossing function
0
time
t_n
t_{n+1}
t_{zero}
$t_x = t_n - \frac{f(t_n)(t_{n+1}-t_n)}{f(t_{n+1})-f(t_n)}$

Figure 4.3.5: Search for the earliest root in integration step interval.

`sign(t1) > 0`	and	`sign(t2) < 0`	`Jroot = +1, istate = 5`
`sign(t1) > 0`	and	`f(t2) = 0`	`Jroot = -1, istate = 5`
`sign(t1) < 0`	and	`sign(t2) > 0`	`Jroot = +1, istate = 5`
`f(t1) = 0`	and	`sign(t2) < 0`	`Jroot = -1, istate = 5`

4.3.3 Zero-Crossing Sticking/Masking

In some situations, in hybrid systems like at the beginning of the integration or just after a zero crossing, the zero-crossing surface tends to remain attached to zero. More precisely, as time advances the surface value does not change. This phenomenon is common in hybrid systems where the system modes may change just after passing a threshold (which is, of course, a zero-crossing surface). In a Scicos simulation, it can very well happen that the input to a system like

$$\dot{x} = |u|,$$

which contains an inherent zero-crossing and mode, remains at zero over a period of time. This means, from the solver point of view, that one of the zero-crossing functions is stuck on the value zero. This situation is not properly dealt with in the solvers. For example, in DASKR, this produces the following error message:

```
DASKR-- R IS ILL-DEFINED.  ZERO VALUES WERE FOUND AT TWO VERY CLOSE T VALUES, AT T = R1
```

This a DASKR message that is raised when a zero-crossing function remains at zero. In this case, the simulation is halted and cannot be resumed unless the model is changed. But conceptually such a case does not pose a problem. To illustrate this point, consider the following example.

$$G(\dot{x},x) : \begin{cases} 0 & \text{if } x \leq 1 \\ (x-1)^3 * (x-2) * (x-3)^3 & \text{if } 1 < x < 3 \\ 0 & \text{if } 3 \leq x. \end{cases} \tag{4.3.1}$$

where $G_{zc}(x)$ is the zero-crossing function, as depicted in Fig. 4.3.6. This zero-crossing function is smooth around $x = 1$ and $x = 3$ and should not cause any problems for the solver. However the solvers fail to simulate this system; the reason being the stuck zero.

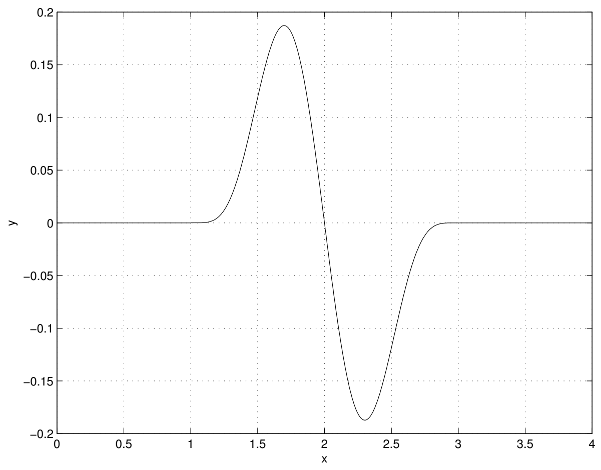


Figure 4.3.6: The zero-crossing function of (4.3.1).

As an anther interesting example, consider the backlash model. Although it is a common object in nonlinear-control, a more precise modeling and simulation may require an implicit solver. The BACKLASH block is a system for which a change in input causes an equal change in output. However, when the input changes direction, because of an inherent dead-band in backlash, the output remains unchanged until a certain moment where the input touches the other edge of the backlash, see Fig. 4.3.7. This figure shows the backlash's model, with the default dead-band width of DB= M-L.

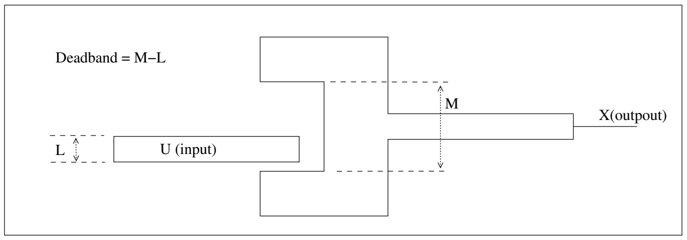


Figure 4.3.7: Backlash block.

Backlash can be modeled as,

$$F(x, \dot{x}, t) = \begin{cases} u(t) + DB/2 - x = 0 & \text{if } Mode = 1 \\ u(t) - DB/2 - x = 0 & \text{if } Mode = 2 \\ \dot{x} = 0 & \text{if } Mode = 3 \end{cases}$$

with three different modes, as depicted in Fig. 4.3.8

- MODE = 1: Engaged in up position, the input is increasing (positive derivative) and the output is equal to the input minus half of the dead-band width.
- MODE = 2: Engaged in down position, the input is decreasing (negative derivative) and the output is equal to the input plus half of the dead-band width.
- MODE = 3: Disengaged, the output does not follows the input, and remains constant

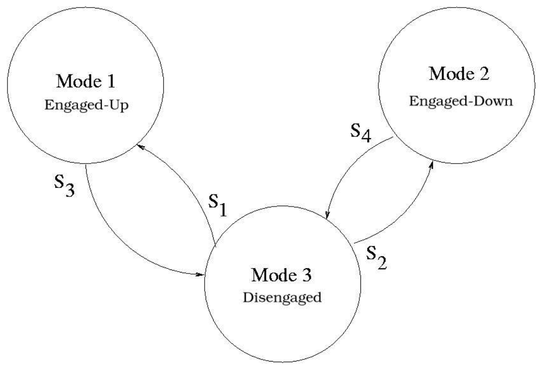


Figure 4.3.8: Backlash mode transition.

To be able to change the mods properly the following zero-crossings can be used:

$$G(x,\dot{x},t) = \begin{cases} g_1(x) = u(t) + DB/2 - x \\ g_2(x) = u(t) - DB/2 - x \\ g_3(x) = \dot{x}. \end{cases}$$

But there is a problem with these zero crossings because g_1 and g_2 are non-zero only when backlash is disengaged. In other times they are zero, so standard LSODAR/DASKR cannot integrate this model. The following pseudo-code shows how stuck zeros are detected and treated in standard LSODAR/DASKR:

```
RT(Y,Y',T0,R0) - Evaluate the zero-crossing functions at T0 and placed them in R0

if all(R0(i) != 0) then Return;

dt=Hmin
Y(i)=Y(i)+dT*Y'(i)
T0=T0+dT

RT(Y,Y',T0,R0) - Evaluate the zero-crossing functions at T0 and place them in R0

if any(R0(i) == 0) then Raise an error message and halt the simulation;
                    else Return
```

At the beginning of each integration step, zero-crossing stuck check is done. If there are any zero-crossing functions with zero value, the solver advances the time for $t_2 = t_0 + h_{min}$ and new state values are calculated by an interpolation function, and then zero-crossing functions are reevaluated. If they are still zero, an error message will be raised and the simulation will be halted. This method could be erroneous and halt the simulation without a real stuck, for example consider this example

$$F(x, \dot{x}, t) = \begin{cases} x - sin(t) = 0 \\ g_z(x) = x \\ x(0) = 0, \; \dot{x}(0) = 0 \quad \text{initial conditions.} \end{cases}$$

In this example, x is an algebraic state, so at $t = 0$ the value of the derivative of $\dot{x}$ is unknown and normally is set to zero. With this initial value, $g(x)$ is zero and even after advancing the time, it remains on zero, so a false stuck alarm is given and simulation will be halted.

Another flaw of standard LSODAR/DASKR is imprecision in detecting of the exact time when a zero-crossing reaches zero and stays stuck, i.e., when a zero-crossing surface is non-zero but it becomes zero without actually crossing the zero, see Fig. 4.3.9. In the official version of LSODAR/DASKR, the zero is announced at t_{n+1} which is not precise enough if the step-size is long.

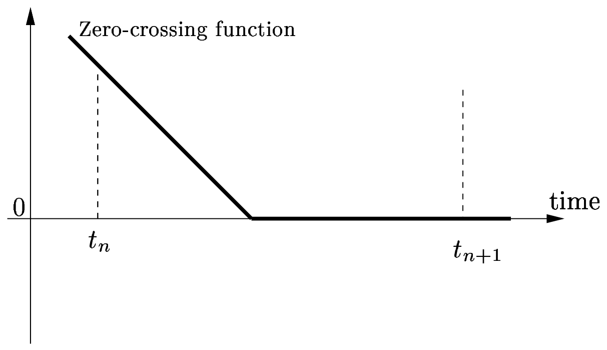


Figure 4.3.9: Imprecision in zero detection.

To overcome the mentioned problems, an automatic `Masking/Unmasking` mechanism of the zero-crossings has been introduced in the solvers LSODAR and DASKR. With this mechanism, when at the start of a simulation a surface has value zero, the zero-crossing function is masked, in other word, it is no longer treated as a zero-crossing function. Its value, however, is monitored continuously to detect if it becomes non-zero at some point. When this happens, an iterative procedure like the one used to pinpoint zero-crossings is used to find the exact time at which the surface is no longer zero (see Fig. 4.3.10). The solver stops and returns as it finds a zero-crossing but with another flag.

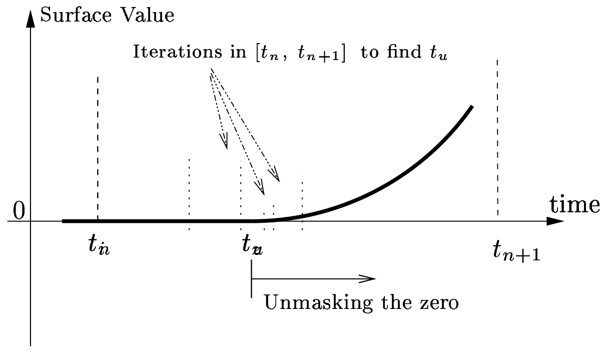


Figure 4.3.10: Unmasking.

The implemented mechanism for masking/unmasking sorts out the problem of imprecision in detection of zero-crossing, depicted in Fig. 4.3.11, as well. It finds the exact time of the crossing by use of an iterative procedure. In this case the simulation is stopped and a zero at t_z is announced and after processing the event of zero-crossing, the integration can be resumed and if zero-crossing function is still on zero, it will be masked, see Fig. 4.3.11.

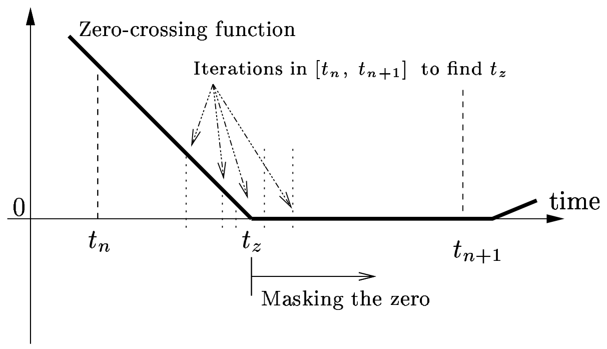


Figure 4.3.11: Masking.

Masking mechanism sorts out another problem with DASKR and LSODAR. If the integration starts exactly on zero a point of zero-crossing function, for example,

$$\begin{aligned} \dot{x} - 1 &= 0, \qquad x(0) = 1 \\ G_{zc} &= x - 1, \end{aligned}$$

this zero-crossing value is ignored. But with the modification made in these solvers, a message indicates that a zero crossing has detaches from zero value. Then the simulator can handle the zero. This is quit useful at the beginning of the simulation for an event such as `if(time>0)then`.

The introduction of the Masking/Unmasking mechanism respects backward compatibility by all means since it handles cases where the actual solvers stop right away with an error message. These modifications are not yet included in the official releases of LSODAR and DASKR but are available in the Scicos source code.

Implementation

During the simulation, DASKR monitors the sign of zero-crossing functions at the beginning and at the end of each integration step in order to detect the possible sign changes in the zero-crossing surfaces. After detecting a sign change, the solver pinpoints the exact location of the crossing by use of a Regula-Falsi algorithm. In this phase, the zero-crossing function is interpolated between two integration-step end-points. The following listing shows how the masking and unmasking procedure works:

```
  t1  = t(n)
  t2  = t(n+1)

1:f1=f(t1)

  f2=f(t2)

  If ( (f1 ==0) or (f2==0) ) then
     tx=(t1+t2)/2
  else
     tx  = min[t1- f1*(t2-t1)/(f2-f1)];  for all zero crossing functions

  If (abs(t1-tx)<tol  or abs(t2-t1)>10*tol ) goto 2

  fx=f(tx)

  If f1*fx <0 then t2=tx
            else t1=tx
  goto 1

2: tz=t2
   if (f1 > f2) direction = -1
   if (f1 < f2) direction = +1
   if (f1 == 0) and (f2 != 0) unmasking
   if (f2 != 0) and (f1 == 0) masking
   end
```

To mask the stuck zeros, at the beginning of the each integration step, all zero-crossing functions are evaluated and saved in `R0` to be compared with the zero-crossing functions values at the end of step. At the same time the index of zero-crossing surfaces with zero value are saved in an array called `Mask`. This array is used in the `Droots` routine to discard stuck zero-crossings from root-finding process.

At the end of each integration step, where the sign of zero-crossing surfaces are compared with the sign of `R0` (the sign of zero-crossing surfaces at the beginning of the step), the value of masked surfaces are checked. If they are not zero, the root-finding process is applied to find the exact time when it has detached from zero. Unlike the Regula-Falsi procedure where the signs are important, here the values are important.

Like what stated in the previous section, `Jroot` and `Mask` arrays play a key role in zero-crossing and zero masking process. `Jroot` is used to communicate between the solver and the main program. The size of `Jroot` and `Mask` are `Ng`. `Jroot` is altered by DASKR/LSODAR to announce to the main program the occurrence of zero crossing, and to report masking and unmasking to the simulator. *Jroot* is filled with these conditions:

- **Zero-crossing:** There is at least one zero-crossing

`f(t1) > 0`	and	`f(t2) < 0`	`Jroot = -1, istate = 5`
`f(t1) > 0`	and	`f(t2) = 0`	`Jroot = -1, istate = 5`
`f(t1) < 0`	and	`f(t2) > 0`	`Jroot = +1, istate = 5`

- **Unmasking:** No zero-crossing but there is at least one unmasking

`f(t1)==0`	`f(t2) > 0`	`Jroot = +2, istate = 6`
`f(t1)==0`	`f(t2) < 0`	`Jroot = -2, istate = 6`

Beside the advantage of the masking and unmasking procedure introduced in Scicos version of LSODAR/DASKR in preventing the unnecessary simulation stop, the solver no longer needs to advance the time at the start of each integration step. This may be seem trivial but in some models, it prevents some false zero-crossing detections. The flowchart of the new DASKR root finder routine and the new code of `Drchek` and `Droots` is depicted in Fig. 4.3.12.

4.4 Jacobian Matrix

One of the most important elements in the numerical integration of an ODE/DAE is the Jacobian matrix. In fact, the convergence rate of the Newton's method is closely related to the precision of the Jacobian matrix. The Jacobian matrix can be provided either analytically or numerically to the solver. Most of the time it is a difficult to provide it analytically, so the solver can compute it via numerical differentiating. But that may cause problem in stiff ODE/DAE. In the following sections some problems that we encountered in integration of DAEs will be explained.

4.4.1 Scaling Problem in Jacobian

DASKR assumes that the error in the function is on the order of floating point roundoff. If that assumption is not valid for your problem, the difference increment must be adjusted to reflect that. Check that you have scaled the difference increment to reflect the size of x. If the components of x differ dramatically in size, consider a change of independent variables to rescale them. Consider this example,

In our case the Jacobian is not available and must be computed numerically in the following manner:

$$J_{ij} = \frac{\triangle \tilde{F}_i(Y_j)}{\triangle Y_j} = \frac{\tilde{F}_i(Y_j + \triangle Y_j) - \tilde{F}_i(Y_j)}{\triangle Y_j}.$$

The $\triangle Y_j$ value depends on relative and absolute tolerances (`Rtol, Atol`) which are solver parameters. These parameters can be adjusted element-wise or globally, but in general purpose simulation environments such as Scicos, it is not reasonable to expect the user to furnish these values. They are selected globally and this can create numerical difficulties if the state variables do not have comparable sizes. Consider for example the following DAE system:

$$DAE = \begin{cases} 0 = \dot{y}_0 - 1 \\ 0 = y_1 - 10^{+06} \\ 0 = y_2 - 10^{-06}. \end{cases}$$

If in this system `Rtol = Atol =` 10^{-6}, then the Jacobian computation yields a singular matrix. The reason is that $\triangle Y_1$ is so small that using double precision arithmetic we have

$$f_2(Y_1 + \triangle Y_1) = f_2(Y_1).$$

Of course, this problem does not concern just the computation of the initial condition, but also the simulation itself. To avoid this problem, we envisage the following possibilities:

- Analytical computation of the Jacobian,
- Intelligent element-wise selection of Tolerances,
- Normalizing state vector,
- Jacobian free methods, like minimum root square.

The first possibility can be implemented if only we have a symbolic representation of the ODE/DAE. The second and third methods can be applied if we have an estimate of the variables, so it cannot be applied in Scicos. The only option is the fourth one that was used in Scicos to handle the cases where the Jacobian matrix is singular.

4.4.2 Singular Jacobian at Initialization

Consider this index-1 DAE

$$0 = \begin{cases} \dot{x}_1 - x_1 \\ x_1 - x_2^3 + 1, \end{cases}$$

with the initial condition

$$x = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

Then the Jacobian matrix is

$$J = \begin{pmatrix} \alpha - 1 & 0 \\ 1 & 3x_2^2 \end{pmatrix},$$

where α is a scaler given by solver. As a result, with this initial condition, the Jacobian will be singular and there is no possibility to continue with regular methods. Note that this problem only exists during initialization and is independent of numerical or analytical Jacobian calculation and depends only on initial guess points. To sort this problem out, a possibility is solving the initialization equation with Jacobian free methods, such as a minimum least square method, and gradient method. Suppose that we want to compute the consistent initial condition for

$$0 = F(\dot{x}_d, x_d, x_a), \tag{4.4.1}$$

where x_d is given and x_a and $\dot{x}_d$ should be computed. So if we construct the vector of unknowns we will have

$$y = \begin{pmatrix} x_a \\ \dot{x}_d \end{pmatrix},$$

and the objective is minimizing $F^t(y)F(y)$. The cost function is

$$C(y) = F^t(y)F(y). \tag{4.4.2}$$

The problem is an unconstrained minimization that can be written as a convex quadratic minimization problem, for which different solution methods are available. The gradient of C is

$$\nabla C(y) = J(y)F(y),$$

where $J(y)$ is the Jacobian matrix that is computed as follows

$$J(y) = \frac{\partial F}{\partial y}.$$

The minimization algorithm has been implemented in CONMIN, available in the ACM TOMS1[3] package. It uses a restarted memoryless quasi-Newton conjugate gradient method [**?**, **?**]. We used this routine for the present study. As input, it requires the value of the cost function, C, and its gradients, i.e., ∇C.

It is worthwhile to note that, although we implemented this method with numerical Jacobian ($J(y)$), using the analytical Jacobian results in a smaller minimum value. The reason is the fact that the numerical Jacobian does not necessarily indicate the maximum descent direction, specially when the step-size is comparable to the perturbation σ in the numerical Jacobian computation.

4.4.3 Jacobian Matrix Update

In the general Newton's method, the Jacobian matrix has to be recomputed and re-factored with each iteration, since the matrix is a nonlinear function of variables. However, great gains in efficiency can be obtained by only computing and factoring the Jacobian matrix once, and using this frozen Jacobian matrix for several subsequent iterations. Since this is an approximation to the true system Jacobian, at each iteration the same convergence could not be reached. Actually, this is not very important since several iterations can be realized using the frozen Jacobian in the time required for computing and factorizing a new one. It is common practice in numerical solver codes not to call for a Jacobian matrix update at a step unless we have an indication that using the matrix of the previous integration step will slow down the convergence. Allowing a Jacobian matrix update more frequently does not necessarily imply that the code becomes more expensive. Since we do this just in the first phase of integration, where the solver attempts to find the initial conditions to start the integration.

For most problems, however, we do not encounter this problem and the performance of DASKR is satisfactory. In particular, when the initial guess concerning the initial condition is not far from the actual initial condition. When this is not the case, and this does occur in Scicos applications, the code can be slightly improved. Currently in DASKR the computation

[3]www.netlib.org/toms/500

of Jacobians is not performed at every step. This, of course, is reasonable to do during simulation for efficiency and justified by the fact that during the simulation, we have a very good initial guess for every nonlinear problem we need to solve. But in computing the initial condition, the initial guess, which is nothing but the previous values of the state, can be far off due to the discrete-time behavior of the system; an event can make the state jump, for example. In this case a more frequent Jacobian evaluation increases the chances of converges. In fact, we have taken a conservative approach by forcing a Jacobian re-evaluation at every step. The overhead is not considerable because this concerns only Jacobian computations for the purpose of computing consistent initial conditions. Such computations are rare events (cold restarts) compared to the simulation effort which is permanent.

4.5 Minimum Step Size

In DASKR, `Hmin`, the minimum step size, is computed as a function of t_n, the current integration time and t_{out}, the solution point:

```
Hmin=4.0D0*uround*MAX(abs(tn),abs(tout))
```

As a result, as integration time progresses, `Hmin` grows. Consider the the integration of the DAE ($0 = f(\dot{x}, x)$), over the two distinct intervals: $T_1 = [0, 1]$ and $T_2 = [1000, 1001]$ using the same initial condition at the starting time. The numerical solver will use two different minimum step sizes on these two intervals even though the DAE is time invariant and the exact solutions are identical. While this is designed to keep the value of h significant, it may cause problems at certain points where the step size should be reduced to meet the error criteria.

To sort this problem out, remember that `Hmin` is intended to be a value slightly above the roundoff level in t_n, the current integration time. As such it is appropriate that it varies with t_n. In DASKR, `Hmin` is used in two ways:

- At the start, $|t_{out} - t|$ is required to be at least `Hmin`, to guarantee that the user has provided the direction of the integration reliably.
- If the integration has difficulty passing the convergence or the error test with step-size h of size $|h|=$ `Hmin`, it stops with an error message saying that h is at `Hmin` and the error test has failed.

In contrast to the DASSL family, the ODEPACK solvers [?] have `Hmin` $= 0$ as the default value, but the solver issues a warning when $t + h = t$. With a software package like Scicos, where a wide variety of problems are solved, and the users may not have much expertise in numerical analysis, it is important to try and reduce the number of integration failures as much as possible. The good experience reported for ODEPACK suggests that this alternative often leads to successful integration. Thus it seems that if `Hmin` is set to 0, this failure mode will be eliminated, provided $|tout - t|$ is nonzero. In Scicos we have changed the DASKR source code and instead of the DASKR test on the `Hmin`:

```
If (abs(h) .ge.  hmin) then
```

we use the test:

```
If (t+h .ne.  t) then
```

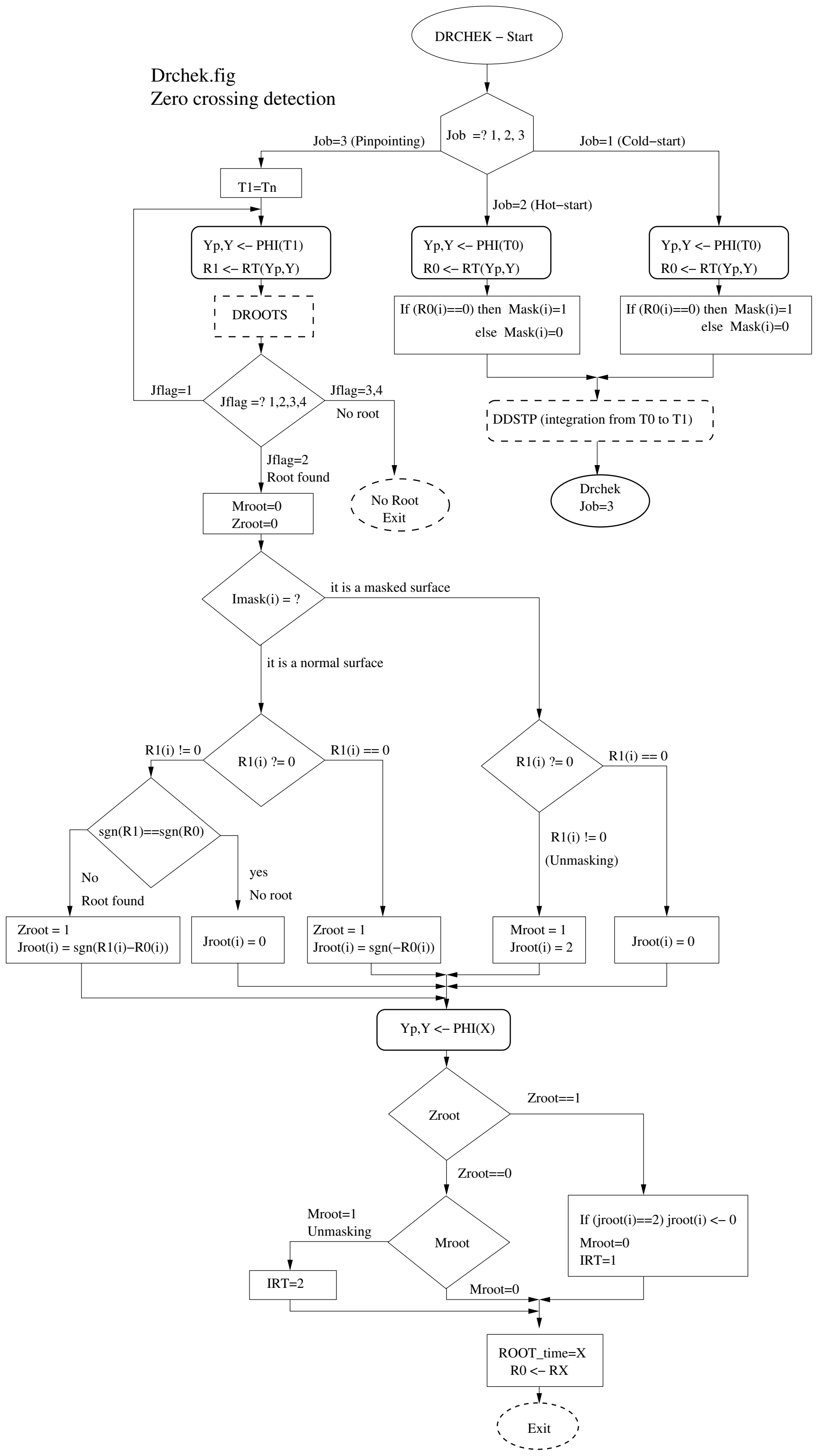


Figure 4.3.12: New DASKR/LSODAR root finder flowchart.

Chapter 5

Scicos Architecture

The Scicos simulator is a hybrid simulator capable of executing discrete-time, continuous-time, and hybrid models. It is composed of a graphical editor, an intermediate compiler (pre-compiler), a compiler, a simulator, and a code generator facility. The Scicos general layout is depicted in Fig. 5.0.1.

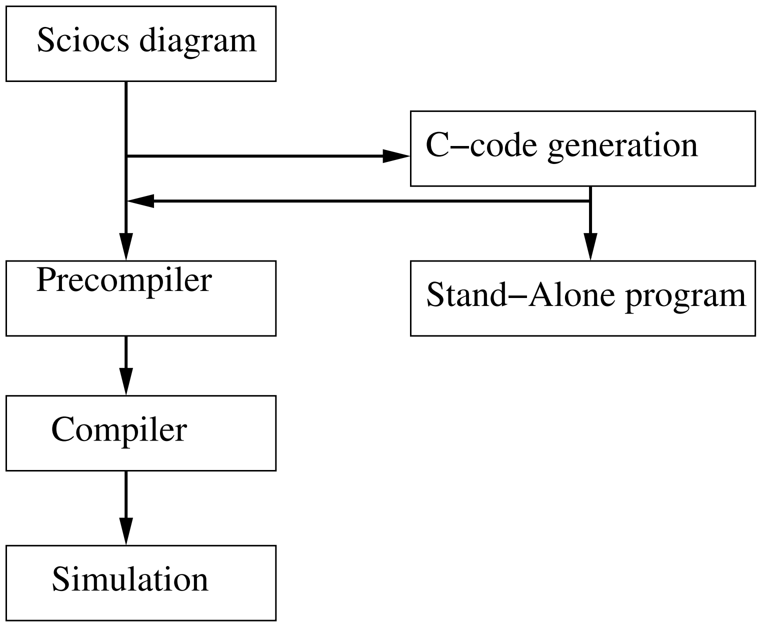


Figure 5.0.1: Scicos layout.

Scicos has a graphical editor to build the hybrid dynamical systems via block diagram formalism. To build a model, the user can get the predefined block instances in the block library and interconnect them by regular links (signal) as well as activation links (events).

The Scicos pre-compiler (*compiler front-end*) translates a Scicos diagram into an intermediate data representation, that will be used by the main compiler. In this stage the following tasks are done

- a flat block diagram is provided through replacing the Superblocks with their contents in the main diagram,

- regular and activation connection matrices between blocks are built,
- a consistency verification such as incompatibility between port parameters of connected blocks and unconnected inputs is performed,
- several data such as block lists, block types, block links, and block inputs/outputs are collected.

This stage is done by the `c_pass1.sci` function in Scicos, and provides a verified model in addition to a primitive collected data that will be delivered to the main compiler.

The (*compiler back-end*) works with the provided intermediate data to produce final usable data for the simulator. This stage converts the *Scicos diagram* to a *Scicos model*. The Scicos model is model that is conformed with the Scicos semantic and all event inheritances have been done. This stage, performed by the `cpass2.sci` function, also determines the execution orders of blocks. The execution orders are the order in which blocks should be activated to update their outputs, zero-crossings, and state derivatives, etc.

After extracting the Scicos model and collecting all execution orders, the simulator can call the blocks to obtain the desired information to perform integration. The simulator uses ODE and DAE numerical solvers to integrate the system and produce the simulation results.

In addition to modeling and simulation, Scicos generates an equivalent C-code for a continuous-time and discrete-time sub-model of the Scicos diagram. The generated C-code can be used in real time applications, independent of Scicos. In this chapter we will focus on the Scicos compiler and simulator and how a simulation is performed.

5.1 Scicos Compiler

A compiler is a computer program that translates a computer program written in one computer language (the source language) into an equivalent program written in another computer language (called the output, object, or target language). The Scicos compiler translates the model built in graphical editor in block diagram form into a data usable by the simulator. Compiling a whole diagram is done in two stages, i.e., pre-compiling, which flattens the block diagram model, and the main compiler that performs several tasks such as:

- Event inheritance mechanisms and algebraic loop detection
- Synchronism and inheritance management and execution order extraction
- Optimizing the execution orders
- Critical event classifications

These tasks have considerable importance in simulation because they determine how the blocks should be called during simulation in different situations. To start, we will explain the inheritance and critical event mechanisms that relate the Scicos compiler to the Scicos simulator.

5.1.1 Inheritance

Strictly speaking Scicos provides an event driven environment. This means that the activation of each block is explicitly determined by an activation signal. However, through the inheritance mechanism, Scicos provides to some extent a data flow behavior as well. Consider the Scicos diagram in Fig. 5.1.1.

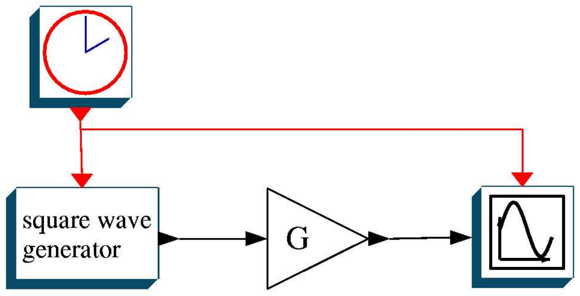


Figure 5.1.1: Scicos model with a discrete-time event generator.

Here we have a block which does not have activation input ports, it is activated by inheritance. A block with no activation input port (and not always active), inherits its activation through its regular inputs. The activation associated with a regular signal is the activation times of the block which has generated the signal. In this example we have only one source of activation, i.e., the `Event Clock`. The `Gain` block inherits its activation from its regular input signal generated by the Square Wave Generator block. This latter is activated by the activation signal from the `Event Clock`, so due to the inheritance mechanism, the `Gain` block is also activated by this activation signal. In fact we would have exactly the same model if the `Gain` block had an activation input port and the output of the `Event Clock` was connected to it. Consider now the example in Fig. 5.1.2.

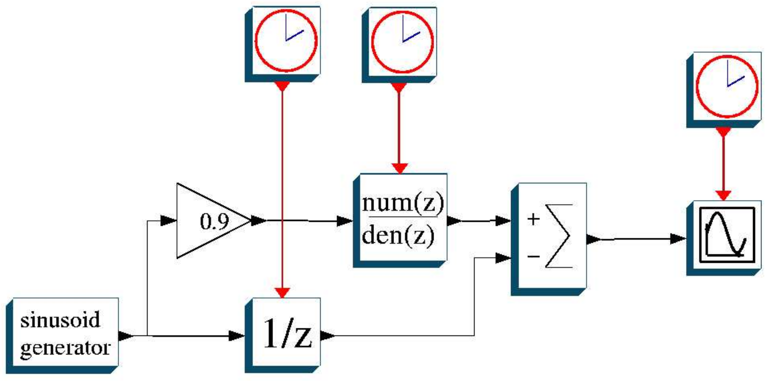


Figure 5.1.2: Inheritance mechanism.

In this example the `Summation` works by inheritance. In such simple cases, inheritance means that an activation input port is added to the block which inherits. Each new activation input is connected to the same activation source that is activating the source block (block its regular output is connected to the regular input of inheriting block). The `Summation` block inherits from two input signals, and the inheritance mechanism creates two input activation ports for the block and connects them according to the sources of the regular inputs. The way Scicos handles the inheritance in this case can be seen in the diagram of Fig. 5.1.3, which is completely equivalent to the original diagram but where all the activations are explicitly drawn.

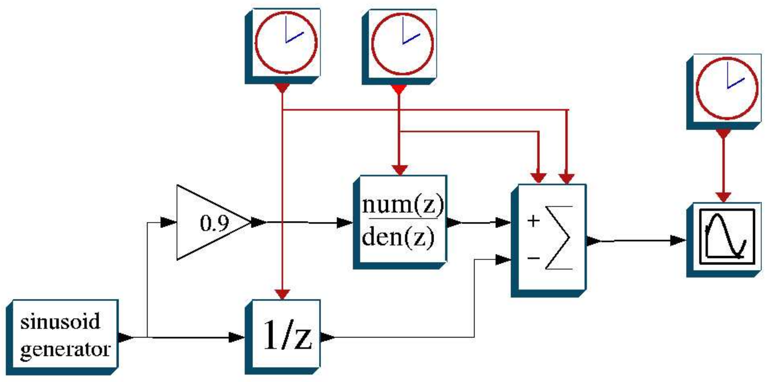


Figure 5.1.3: Inheritance mechanism, explicit activation of diagram of Fig. 5.1.2.

The inheritance mechanism is fairly simple to understand when a single activation exists in the model. With the presence of conditional activations or in asynchronous cases, the inheritance still works. For example, the Scicos block diagram system of Fig. 5.1.4, after inheritance, will be converted into diagram of Fig. 5.1.5.

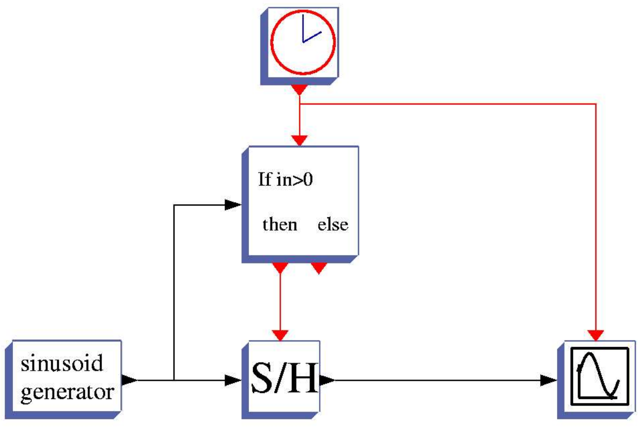


Figure 5.1.4: Block diagram system with conditioning.

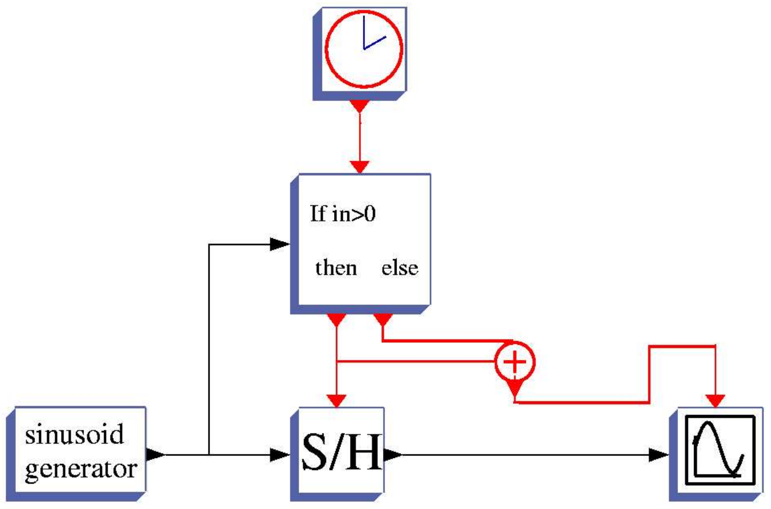


Figure 5.1.5: Inheritance mechanism with conditioning.

5.1.2 Continuous-time and Always-active Blocks

A block can be declared *always active* by declaring the parameter `dep_t=True` in its interfacing function. This means, in particular, that the block is a continuous-time block, i.e., its outputs and its states can vary continuously with time. The inheritence mechanism is applied for the continuous-time block as well. Normally, the always active property should be designated by an activation signal received on an activation input port if we are to be consistent with Scicos formalism. However, to avoid useless complexity at the level of the editor, this property is simply specified as a block property. There are a number of blocks in Scicos palettes which are always active. These blocks can be used with other blocks to construct hybrid diagrams.

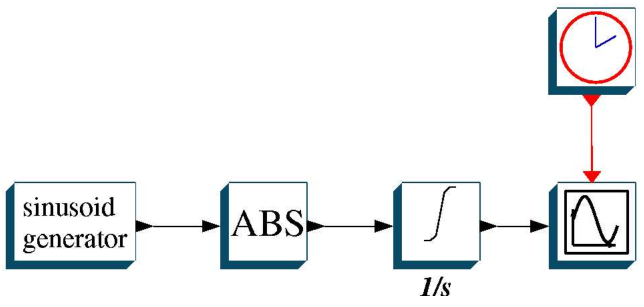


Figure 5.1.6: Always active blocks.

In the example shown in Fig. 5.1.6, the `Sinusoid Generator` and the `Integrator` block (1/s) have the always active property. The `Absolute value` block inherits this property from `Sinusoid` block. The conditional activating also works with continuous-time signals.

Consider the Scicos diagram in Fig. 5.1.7.

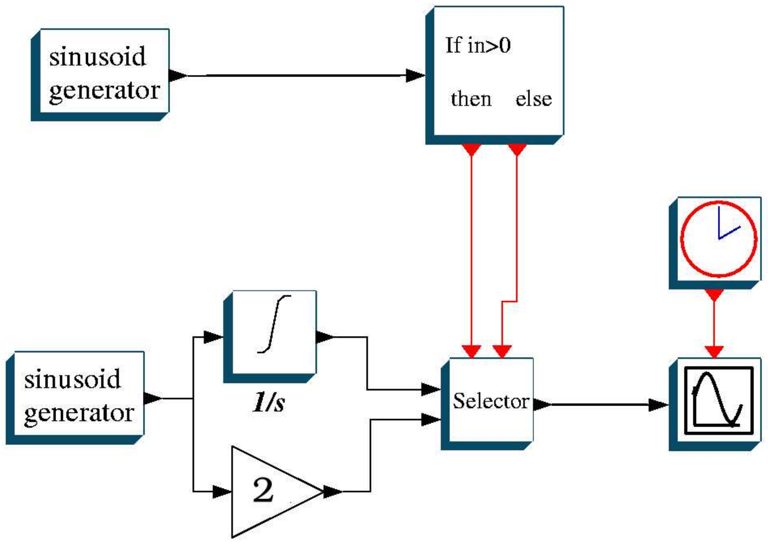


Figure 5.1.7: Continuous-time conditioning.

In this example, the scope displays either the integral of a sinusoid wave or its amplified signal. This is done by selecting ($\int \sin(\omega t)$) or ($2sin(\omega t)$) depending on the sign of another sine wave signal ($sin(\alpha t)$).

5.1.3 Scicos Formalism for Continuous-time Signal

The formalism used in Scicos is based on the formalism of synchronous languages, in particular, *Signal* and its extension to continuous-time systems [**?**]. We do not give a full presentation of the formalism here (for more see [**?**, **?**, **?**]), we simply review some aspects which specifically have to do with the continuous-time behavior in order to lay the ground work for presenting the specific issues related to continuous-time simulation.

Even though there exists an inheritance mechanism in Scicos formalism which provides a data-flow type of behavior, Scicos is fundamentally event-triggered. This has made the continuous-time extension non-trivial. The basic idea consisted in treating the continuous-time events just like an ordinary (discrete-time) event [**?**].

Events are signals which activate system components in discrete-time event-driven environments. We extend the notion of event to obtain what we call an *activation signal* which consists of a union of isolated points in time and time intervals [**?**, **?**]. Each Scicos signal is then associated with an activation signal specifying time instances at which the signal can evolve, see Fig.5.1.8.

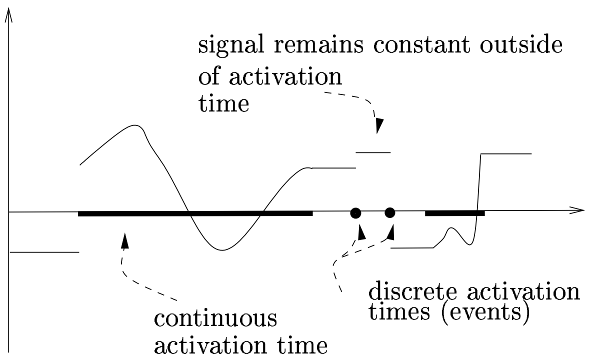

signal remains constant outside of activation time
continuous activation time
discrete activation times (events)

Figure 5.1.8: A typical Scicos signal. Thick line segments represent the activation times.

The fundamental assumption is that over an activation interval, in the absence of events, the signal is *smooth.* The Scicos compiler propagates the activation signals through the model in order to obtain activation information about all the signals present in the system. This information is valuable for the simulator which has to properly parameterize and call the numerical solver.

It would be unrealistic to imagine that a formalism can be developed independent of the properties of the numerical solver. That is why it is important to study these properties and identify precisely what properties are important and must be taken into account in the development of the formalism and the implementation of the modeler and simulator.

5.1.4 Compiler Outputs

After establishing all inherited activation links, the compiler prepares the necessary information for the simulation of the model. The simulator needs to have several types of information about the model at requested instants of simulation. State derivatives of ODEs (or residuals in DAE models), new discrete-time state to update, and zero-crossing function values should be provided for the numerical integrator. For each of these types of information that are distributed in the model, several blocks should be called to gain valid data. The question that arises is which blocks and in which order they have to be called in order to provide the requested information. The Scicos compiler's task is generating these **execution orders**.

Execution order is the order in which the blocks should be called to obtain the requested data. For example, consider the Scicos diagram depicted in Fig. 5.1.9.

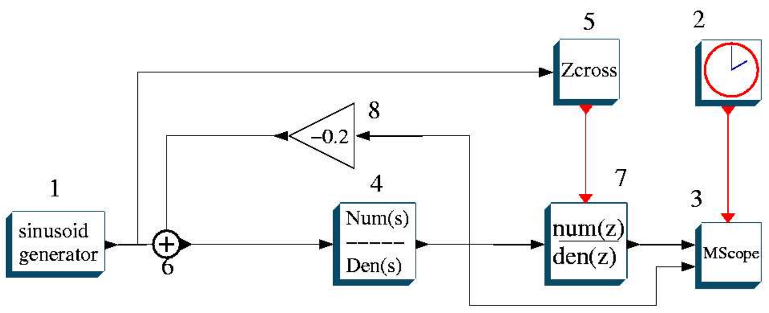


Figure 5.1.9: Continuous-time conditioning.

In this Scicos block diagram, to compute the state derivatives, the output of block {(1, 4), (8), (6)} should be updated and the continuous-time states of block {(4)} read. Note that the reading order between each set is important, e.g., calling with order{(1, 4), (6), (8)} gives a false result. To compute the zero crossing function value, the output of block {(1)} should be updated then zero-crossing functions of block {(5)} be read.

The discrete-time states of a system are in the blocks that contain discrete-time states. Any block containing discrete-time states (here it is {(7)}) should have either a discrete-time input-event port, or an internal zero-crossing to update its discrete-time states. Whenever an event is activated, the activated blocks (`clkptr`, `ordptr`, `ordclk` vectors) should update their outputs, then the discrete-time equations in activated blocks can be updated.

To each event, we associate a list specifying the blocks which are to be activated when the event fires and the order in which they should be activated.

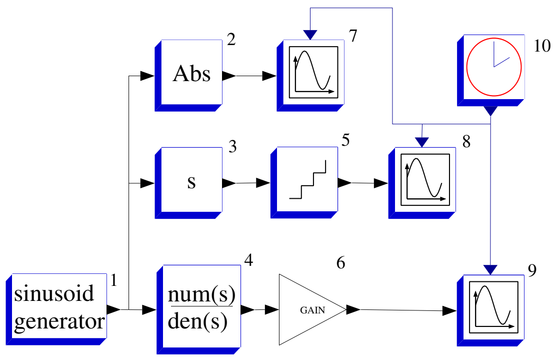


Figure 5.1.10: Execution order.

The same holds for continuous-time activations (as it was said before, Scicos treats the continuous-time activation similarly to the way it handles discrete-time events). A continuous-time activation is a call to the system to update its continuous-time components. For that, there is an activation list and an order in which the activations must be realized. We call it `Cord` list.

For example, in Fig. 5.1.10, to update the system at the requested times, the blocks $\{(1), (3, 4), (2, 5, 6)\}$ should be called to update their outputs. During the continuous-time simulation (when the solver is at work), the 3, 4 blocks are called to deliver their updated residuals ($f(\dot{x}, x, t, u)$). Note that these blocks are the blocks in `Cord` which contain continuous-time states.

The `Cord` list can be used either for updating the continuous-time parts of the system at specific times or during the integration by the solver. So unlike discrete events, continuous-time events are evoked in four distinct situations:

1. In the simulator, to update the continuous-time parts at a requested time (`flag=0, phase=1`).
2. By the differential equation solver, to update the continuous-time parts so that system residuals can be correctly evaluated (`flag=0, phase=2`).
3. In the simulator, to update the `modes` variables (`flag=9, phase=1`).
4. By the differential equation solver, to update the values of the zero-crossing surfaces (`flag=9, phase=2`).

It turns out that in the second, third, and forth cases, we do not necessarily need to activate all the blocks in the `Cord` list. In the second case, we only need to update the outputs of blocks which affect directly or indirectly the input of a state bearing block. Same for the third case except that blocks containing zero-crossing surfaces must be considered. For example in Fig. 5.1.10, the 2, 5, 6 blocks which are in `Cord` don't have any effect on the residuals. Thus, they can be excluded from `Cord` and be stored in a new list. The new list, `Oord` list, is comprised of all the blocks for which their outputs may affect the input to a block containing a continuous-time state. The blocks affecting the zero-crossing surfaces (in this case, *Abs* and the *Quantizer* blocks) are 1,2,3,5. We call this list `Zord`.

The use of `Oord` and `Zord` optimizes the number of calls to each block in the integration phase. This optimization is very important for simulation efficiency because the solver requires many function evaluations (depending on the stiffness/nonlinearity of the system and the required precision). In a typical application, for every evaluation due to a discrete-time event, the solver requires hundreds or even thousands of continuous-time function evaluations.

5.1.5 Event Classification

Besides obvious solver parameterizations such as setting various error tolerances, maximum step size, etc..., the simulator must automatically start and stop the simulation when necessary. When an event occurs, the continuous-time simulation must stop so that the event-triggered components of the system can be activated. Once this is done, the continuous-time simulation can proceed. So event times are the times of stop and restart of the continuous-time simulation. Although they are similar in this way, the events originating from different sources perform different tasks. Hence they have different importances and properties. In subsequent sections, two important properties of events will be introduced.

Event Predictability

Events can be put into two categories: *predictable* and *non-predictable*. Consider the following system:

$$\dot{x} = \begin{cases} x\sqrt{1-t} & \text{if } t \leq 1 \\ 0 & \text{if } t > 1. \end{cases}$$

To model this system in Scicos, we need to generate an event at time $t = 1$ in order to change the dynamics of the system. See the Scicos diagram illustrated in Fig. 5.1.11.

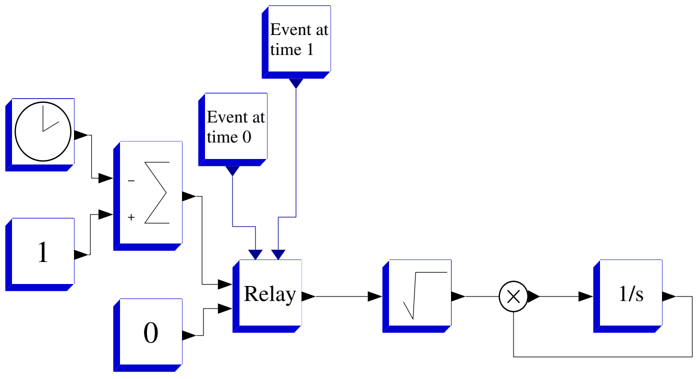


Figure 5.1.11: Predictable event.

But this event is also predictable. We know ahead of time its occurrence time ($t = 1$). In general, most events in a Scicos diagram are generated by `Event Clocks` and their times are predictable.

Non-predictable events are zero-crossing events. These events are generated when a signal crosses zero and their activation times are not known in advance. A predictable event can be considered and modeled as a non-predictable event. For example in the above example, the event at time 1 can be obtained by using a zero-crossing test on the function $1 - t$. But that would be inefficient for two reasons: first, the zero-crossing tests are additional work for the solver. Second, to detect a zero-crossing, the solver has to go beyond the crossing and perform iterations to pin-point exactly the location of the crossing. In the above example, this results in an error since for $t > 1$, $\sqrt{1-t}$ is not defined. See the Scicos diagrams illustrated in Fig. 5.1.12 and Fig. 5.1.13. In the Scicos diagram illustrated in Fig. 5.1.12, the zero-crossing block defines a zero-crossing surface. The solver has to go beyond the surface in order to find the exact time of crossing. That is why it attempts to evaluate the value of $\sqrt{1-t}$ for t larger than 1. The same system is implemented in a slightly different way as illustrated in Fig. 5.1.13. In Fig. 5.1.13, the `If-then-Else` block redirects its activation signal to one of its output activation ports depending on the value of its regular input. If this latter is positive, the activation goes through the `If` port, if not, it goes through `Else`. The `If-then-Else` and the `Event-Select` blocks are the only blocks which redirect activation signals without creating delays. These blocks have built-in zero-crossing surfaces that generate an event when the activated output port changes, because such a change may produce discontinuity and thus the solver must be informed. The presence of this zero-crossing forces the solver to step beyond the $t = 1$, and as a result, the simulation fails again.

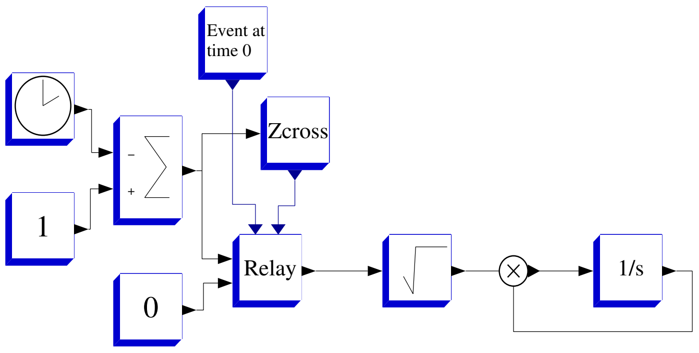


Figure 5.1.12: The simulation fails in this case.

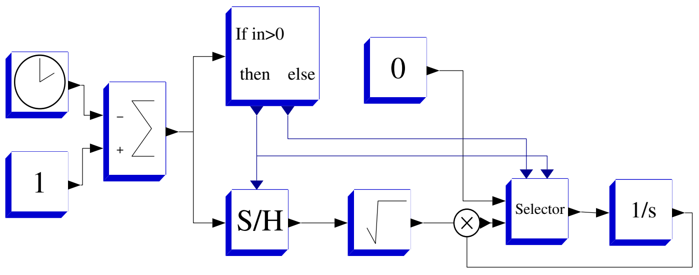


Figure 5.1.13: The simulation fails also in this case.

Event Criticality

Although all events force the simulator to stop and restart the integration, they are, however, not equal in importance. Consider the Scicos diagrams in Fig. 5.1.14 and Fig. 5.1.15. In

Fig. 5.1.14, the event generated by the `Event Clock` drives the `Scope`. The `integrator` block (`1/s`) receives on its regular input port the sine function which is generated by the *ever-active*, `sinusoid generator` block (ever-active means the block is always active). The output of the integrator is sampled by the `Scope` at its activation times (times at which it receives an activation on its activation input port, (i.e., the times of the events generated by the `Event Clock`). In this case, the solver must be stopped at each of these times, however, since the corresponding events do not produce any non-smoothness on the input of the integrator, there is no reason to re-initialize the solver (do a *cold restart*). We say this event is **not critical**.

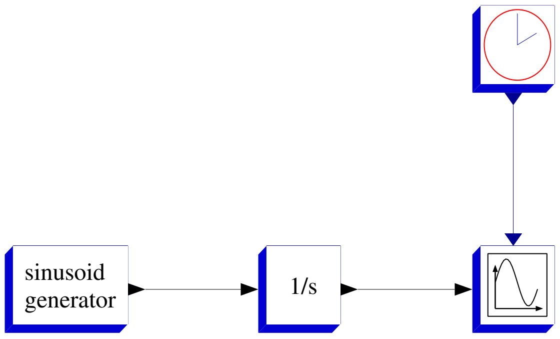


Figure 5.1.14: Non-critical event.

In diagram of Fig. 5.1.15, the event activates the `random generator` block which outputs a random variable. This output remains constant until the next event reactivates the block. The integrator then receives a piecewise constant signal to integrate. Thus the event in this case creates a discontinuity at the input of the integrator. Clearly in this case the solver must be re-initialized. This event is **critical**.

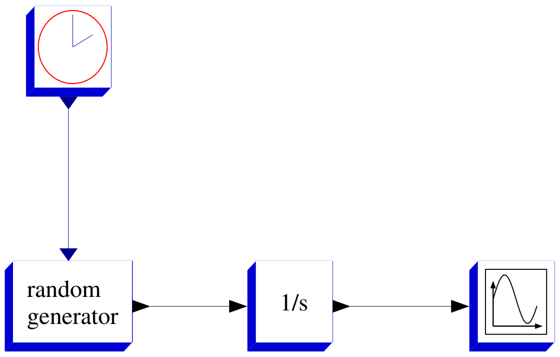


Figure 5.1.15: Critical event.

Furthermore, an event can be critical in another way: an event can cause a jump in the internal state of a block. One such example is the integrator with reset on event. Clearly if a jump is produced in the state, the solver must be restarted cold.

In Fig. 5.1.13, if the event is treated as predictable, the simulator, using the fact that the upcoming event at time 1 is critical, sets the critical time to 1 preventing the solver to step beyond $t = 1$.

Critical Event Classification

In the previous sections the importance of Critical events has been shown. Here the definition, and the classification algorithm of Event in terms of criticality will be discussed.

Definition: *A discrete-time event is critical if upon activation, either an input of a block, containing zero-crossing surfaces or continuous-time states, is updated, or the continuous-time states of a block jump.*

Before explaining the critical event classification algorithm, the following items should be noted.

• The critical property concerns only discrete-time events. Therefore to avoid considering a continuous-time event as critical, all purely continuous-time activation links must be excluded from the classification procedure.

• Block having direct input-output relationships (direct feed-through) and ever-active blocks can pass on the discontinuities arriving at their inputs to the following blocks. These blocks are called DTB (Discontinuity transmitting block).

• Synchro blocks that do not have an input event port and inherit the continuous-time activation, are referred to as Continuous-time Synchro (CS) blocks.

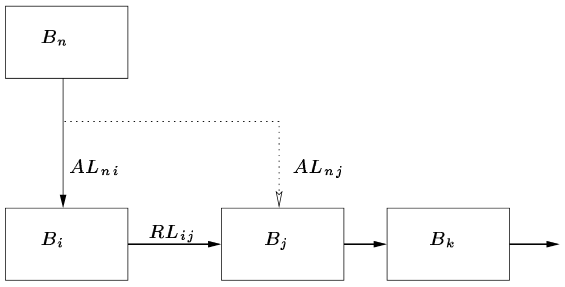


Figure 5.1.16: Event inheritance.

Algorithm:

1. Propagate the events through DTB's to determine the blocks that are potentially subject to discontinuity at their inputs. For example in Fig. 5.1.16, if block B_n activates block B_i (here i.e., AL_{ni}), and if a regular link between blocks B_i and B_j exists (i.e., RL_{ij}), and if the block B_j is a DTB, then an explicit activation link between block B_n and block B_j (i.e., AL_{nj}) is established[1]. Repeat this process until no more activation link can be established.

2. Find all CS blocks (Synchro blocks not activated explicitly) and store them in CS-list.

[1]All the changes made in the diagram are discarded at the end when the critical events are identified so that other phases of compilation can be carried out normally

3. Identify all the blocks activated by CS blocks and add them to the list of DTB's. For example, in Fig. 5.1.17, the B_j block is activated by CS_n. so it should be added it to the list of DTB's. The reason is that the block B_j receives a continuous-time activation through CS_n therefore it can pass the eventual discontinuities at its input port into the B_k block.

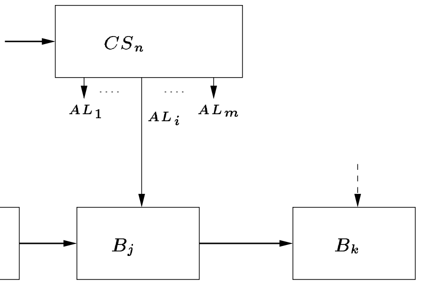


Figure 5.1.17: Continuous-time Synchro Block.

4. Remove all activation links originating from CSs (e.g., in Fig. 5.1.16 all AL_i links) and then remove CSs from the diagram.

5. Go to step 1 and repeat until in step 2 CS-list becomes empty.

6. For each event source, search through all the blocks it activates. If any of them contains continuous-time states or zero-cross surfaces, then flag the event as critical.

At later stages of compilation, Synchro blocks (i.e., `If-then-Else` and `Event-Select`) are duplicated if they have multiple sources of activation. So each Synchro, at the end, is activated just by one activation source. This process is done after critical event classification and the newly generated events inherit the property of the original events.

5.2 Scicos Simulator

The simulator of Scicos is supposed to generate the behavior of the Scicos model, that is very often a hybrid system. A hybrid system simulation should be able to perform, discrete-time event scheduling, continuous-time integration, and discrete-time and continuous-time part interactions (state events and event handling). Since a hybrid system can be a purely discrete-time or a purely continuous-time system, the simulator should perform its normal operations even for a purely discrete-time or a purely continuous-time system, as well. In this thesis the discrete-time simulation is not treated in detail (interested readers are referred to [?, ?], but the discrete-time event scheduling in the simulator is outlined in order to be able to understand the general timing of hybrid model simulations. In the following sections, first we study the structure of a purely discontinuous-time simulator, then a simulator that uses a numerical solver to integrate a system, and at the end, we add them up together to obtain a hybrid simulator.

5.2.1 Purely Discrete-time Simulator (Event Scheduler)

The discrete-time part of a hybrid model is defined by the discrete-time variables, the discrete-time equations, and the simulation time. A discrete-time signal is changed only with discontinuous jumps over time and its states remain unchanged until the next event arrives. At each event time, discrete-time equations are activated to update the state of the system. The simulator's role is the scheduling and activating the events to advance the simulation time. The overview of the Scicos event scheduler simulator has been given in Fig. 5.2.1.

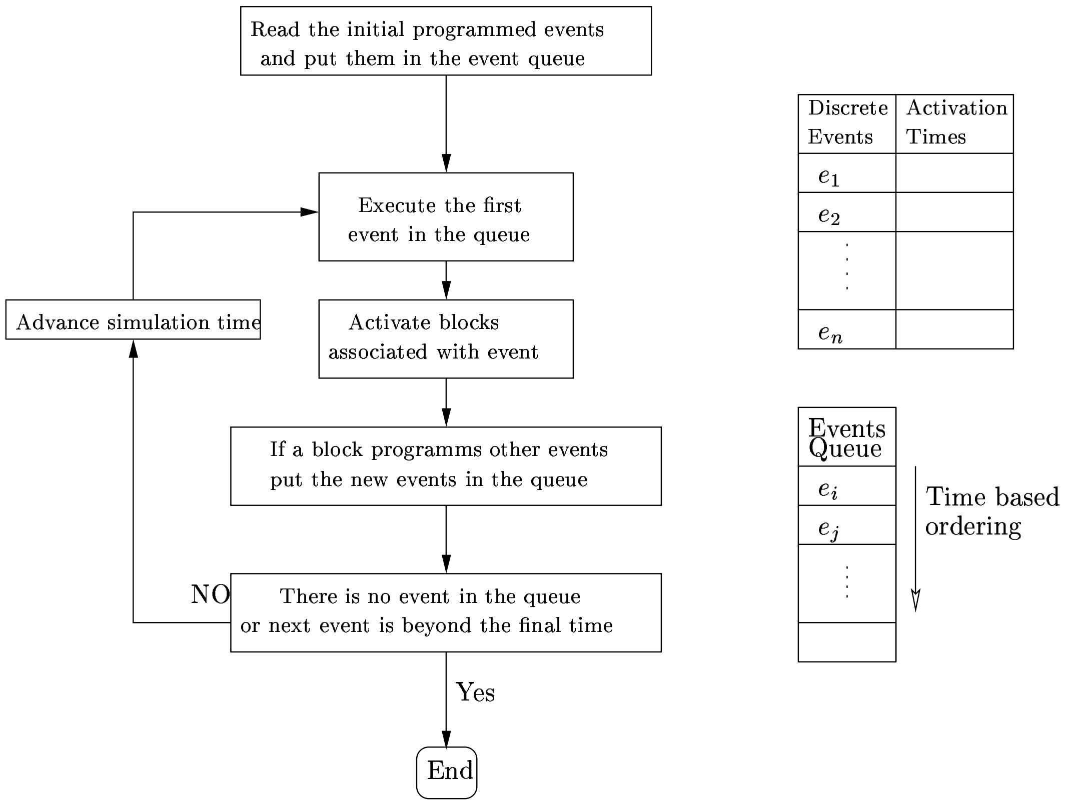

Read the initial programmed events and put them in the event queue
Execute the first event in the queue
Activate blocks associated with event
If a block programms other events put the new events in the queue
There is no event in the queue or next event is beyond the final time
NO
Advance simulation time
Yes
End
Discrete Events
Activation Times
e_1
e_2
e_n
Events Queue
e_i
e_j
Time based ordering

Figure 5.2.1: Discrete-time simulator/Event scheduler.

In Scicos, a simulation is started by reading the pre-programed events in the interfacing function of the blocks. The execution time of programed events are inserted in the `tevts` vector. To manage the event scheduling, Scicos uses an event queue (`evtspt` vector). The events are arranged in the vector according to their execution time. A pointer (`pointi`) indicates the next event to be executed.

Each event has an associated list that indicates the blocks to be activated. When an event is fired (activated), the blocks in the associated list are called in an order defined by `ordptr` and `ordclk` vectors. The blocks in the list are called two times; First, they are called with `flag=3` to program the future events. If there are any, the new event is programed in the event queue. Then, the blocks are called with `flag=2` to update their discrete-time states. This process is done in the `ddoit` function in `scicos.c`. The flowchart of `ddoit` is in Fig. 5.2.14.

5.2.2 Pure Continuous-time Simulator

Simulation of a continuous-time system is nothing but calling a numerical solver with appropriate parameters and inputs. Scicos uses two numerical solvers for ODEs and DAEs. ODEs can be considered to be an special case of DAEs. But DAE and ODE solvers are quite different in several aspects such as convergence, initial conditions specification and speed. So although ODEs can be integrated by DAE solvers, we preferred to use another solver to integrate ODEs efficiently.

In Scicos, LSODAR is used as a ODE solver and DASKR is used as a DAE solver. Since the BDF methods used in DASSL family solvers are designed to solve index-1 full implicit DAEs, and index-2 semi-explicit DAEs, we have confined ourself to model and simulate the index-1 fully implicit problems. So at this time the Scicos simulator needs to receive an index one DAE. If the user formulates a problem which results in a DAE of higher index than one, the resulting Jacobian matrix will be singular and the DASKR integrator will most likely return a (**-8**) error message i.e.,

```
  IDID = -8
 -- The matrix of partial derivatives appears to be singular (direct method).
```

To get an idea of how DASKR works, this is the calling sequence of the DASKR solver in Scicos:

```
C2F(ddaskr)(C2F(simblkdaskr), neq, told, x, xd, &t, info,  &rtol, &Atol,
            &istate, &rhot[1],&nrwp, &ihot[1], &niwp, rpardummy, ipardummy,
            C2F(Jacobian), rpardummy, C2F(grblkdaskr), &ng, jroot);
```

where

- **simblkdaskr** is the name of a subroutine that provides the residual values of the DAE system,
- **neq** is the number of equations in the system,
- **told** is the current value of the simulation time,
- **x** contains the solution components at told,
- **xd** contains the derivatives of the solution components at told,
- **t** is the next event time or final time,
- **info** is an integer array used to communicate details of how the solution is to be carried out,
- **rtol, Atol** represent absolute and relative error tolerances,
- **istate** indicates what the code did,
- **rhot** is a real work array of length `nrwp` which provides the code with needed storage space,
- **ihot** is a integer work array of length `niwp` which provides the code with needed storage space,

- **rpardummy, ipardummy** are real and integer parameter arrays which you can use for communication between your calling program and the Residual, root functions, and Jacobian subroutines,
- **Jacobian** is the name of a subroutine which can optionally be provided for calculating Jacobian data involved in solving linear systems,
- **grblkdaskr** is the name of the subroutine for defining root functions. The number of root functions is `ng`,
- **jroot** is an integer array of length `ng` for output of root information.

In a purely continuous-time system, the **told** and **t** are the start and final simulation times, but in a hybrid system, they define the time interval between two consecutive events.

The `simblkdaskr` subroutine is called to compute the residuals of the DAE set at solution points. The diagram in Fig. 5.2.15 shows a simplified flowchart of reading continuous-time states in the `simblkdaskr` routine of the Scicos simulator. To guarantee that inputs of all blocks are updated when the residuals are read, the blocks should be called with `flag=0` in an order which is stored in `oord` vector. So, the `simblkdaskr` routine calls the block in the `oord` list with `flag=1` to update the block outputs, then it calls them again with `flag=0` (in the same order) to read their residuals.

The `grblkdaskr` subroutine is called to compute the zero-crossing function values to detect the roots. The diagram in Fig. 5.2.16 shows a simplified flowchart of reading zero-crossing surfaces in the Scicos simulator. To compute the zero-crossing functions, the block containing the zero-crossing surfaces should be read. To read them, the `grblkdaskr` function calls the block in an order which is defined in the `zord` vector. First, they are called with `flag=1` to update their output, then they are called with `flag=9` to read their zero-crossing values. Whenever DASKR finds a root, it stops the integration and returns with `Idid=5`, and the crossed surfaces are indicated in the `jroot` vector. This option is used in hybrid simulator to generate state events.

When the numerical solver stops integration, it returns with flag (`istate`) indicating the integration situation. The flag can have several values. Some of them that happen more frequently are:

- **1:** A step was successfully taken in the interval-output mode. The code has not yet reached `t` and current time is `told`.
- **2:** The integration to T_{stop} was successfully completed (`T=`T_{stop}) by stepping exactly to T_{stop}.
- **3:** The integration to `t` was successfully completed (`told=t`) by stepping past `t`. `x` and `xd` are obtained by interpolation.
- **4:** The initial condition calculation, with `INFO(11) > 0`, was successful, and `INFO(14)=1`. No integration steps were taken, and the solution is not considered to have been started.
- **5:** The integration was successfully completed by finding one or more roots at `told`. The valid roots are indicated by setting `1` or `-1` in the `jroot` vector.
- **6:** The integration was successfully completed by finding a root, detached from zero. The unmasked roots are indicated by setting `2` or `-2` in the `jroot` vector.

- **-1:** A large amount of work has been expended (about 500 steps).
- **-2:** The error tolerances are too stringent.
- **-7:** The nonlinear system solver in the time integration could not converge.
- **-8:** The matrix of partial derivatives appears to be singular (direct method).

A major problem in simulating a DAE system, is to provide a consistent initial condition. This happens only one time at the beginning of simulation in a purely continuous-time system. But in discontinuous DAEs, this will be far from a trivial problem. To have a better control on DAE initialization, DAE consistent initial condition finding and DAE integration are separated. First a solver is called to find a consistent set of initial condition, then the same solver or another is called to perform the integration. This has been shown in digram of Fig. 5.2.2. This subject will be studied further in the coming sections.

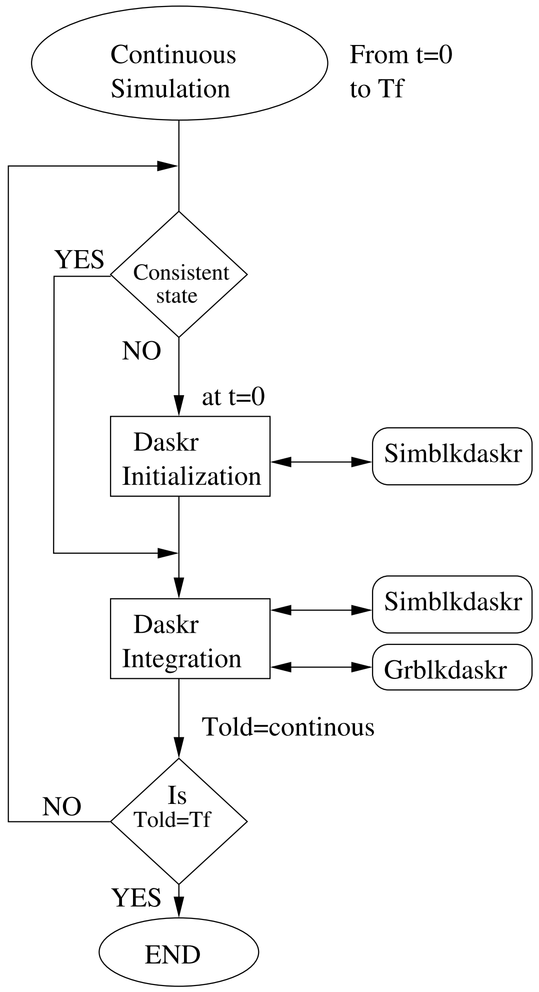


Figure 5.2.2: Continuous-time simulator.

5.2.3 Hybrid Simulator

Simulation of a hybrid system is composed of two phases; the *discrete-time phase* and the *continuous-time phase.* In the discrete-time phase, event scheduling, event activating and state updating (discrete-time and continuous-time) can be done, but the simulation time does not advance. In the continuous-time phase, the numerical solver integrates the differential equation, the simulation time advances, and an event detecting process is performed. Integrating of the continuous-time system is done from one event time to the next event time. The continuous-time phase and discrete-time phase are interlaced.

In Scicos a discrete-time phase starts when an event occurs, and lasts until all blocks in `ddoit` or `zdoit` (depending on event type, time-event or state-event) are called and achieved in a consistent continuous-time state, in the sense that no further discrete-time actions can take place before a continuous-time phase has elapsed. This is important when dynamical system contains `If-then-Else` or `Switch` blocks. Between each two adjacent discrete-time phases a continuous-time phase takes place. A continuous-time phase starts after a discrete-time event and ends when another event (either predictable or unpredictable) arrives.

In previous sections, we learned how a discontinuous-time event and continuous-time simulator work. A hybrid simulator has the structure of discontinuous-time event simulator and a continuous-time integration is run in between two consecutive discrete-time events. The rest of the simulator is detecting and activating state events and some optimization strategies. The flowchart of the Scicos simulator has been given in Fig. 5.2.17, 5.2.18, 5.2.19, and 5.2.20. In the following sections we will focus on zero-crossing events and the initialization problem.

5.2.4 Zero-Crossing Management

The continuous-time part of a hybrid system can influence the discrete-time parts only via discrete-time events. In integration phase, if the solver detects a zero-crossing, it stops the integration and returns with flag (`istate=5`). The crossing time is stored in `told`, and crossed roots are indicated in `jroot`. Using these information, the associated zero-crossing event will be programed in the event queue. It will be the next event to be executed before resuming integration. The solver stops also when a masked zero detaches from zero. In this case the corresponding flag is (`istate=6`).

Numerical Solver Restart Control

There are many events in a hybrid system, such as the events of a clock that provides a sample time for the scope or sub-sampling clocks. Most of them do not have any effect on the system. For example, a clock event updating the scope does not cause any discontinuity, so it is not necessary to restart the solver at each clock event. As cold-restarting the solver is costly and time consuming, solver should be cold-restarted as few times as possible. The classification of the events allows us to avoid unnecessary cold restarts. Without it, at every event time, the solver should be cold restarted to make sure that the numerical solver uses consistent information.

The lack of consistency is avoided in two ways. If the critical event is fired immediately, a cold restart is performed. Because in such a case, a discontinuity may occur in a signal (or in its derivative) fed to a block containing continuous-time states or zero crossing surfaces. This discontinuity may cause the solver to fail during restart. This is particularly a problem in Scicos because Scicos uses a BDF (backward differentiating Formula) method to integrate the continuous-time systems. In order to implement this method, the `critev` vector stores the critical events. Whenever an event is fired in `ddoit`, it's criticality is checked. If it is a critical one, the next call to the numerical solver will be a cold-restart.

Another situation where inconsistency can occur is when a critical event is programed for a future time. If the solver had been allowed to look beyond this time, the new critical event may affect values which are already used by the solver, i.e., values beyond this time. This change of model can result in a failure of the solver which expects to receive consistent information. To see more precisely what happens in this case, note that after each event at time $t(i)$, when the integration resumes, the start-time $t(i)$, the end-time $t(i+1)$, and the time beyond which the solver is not allowed to explore $t_{\text{stop}}(i)$ are passed to the solver to integrate the system from $t(i)$ to $t(i+1)$. Note that the solver may advance the time beyond $t(i+1)$ and compute the solution at $t(i+1)$ by interpolation. Let $t_{\max}$ be the largest value of t for which the solver has requested an evaluation of the function; clearly $t(i+1) \leq t_{\max} \leq t_{\text{stop}}(i)$. After the processing of the event(s) at time $t(i+1)$, the integration resumes from $t(i+1)$. If the events at $t(i+1)$ program a critical future event at a time t_c earlier than $t_{\max}$, then the function evaluations done in the previous integration period are no longer valid. One way to make sure that such a situation does not occur is to force a cold-restart if $t_c < t_{\text{stop}}(i)$. Note that in this case we set $t_{\text{stop}}(i+1) = t_c$ if not $t_{\text{stop}}(i+1) = t_{\text{stop}}(i)$.

In addition to automatic management of solver cold restarting, the user can directly forces the solver to perform a cold-restart by evoking the external function:

```
void do_cold_restart()
```

This function forces the simulator to do a cold-restart. From the simulator's point of view, calling this function in a block specifies that a discontinuity occurs (or has occurred) and the state needs to be re-calculated. When such a statement is executed, a new initial condition is re-calculated at the beginning of the next integration.

5.2.5 Multi-Model System and Discontinuous DAE

Modeling a physical system very often leads to a DAE with discontinuities or a multi-model DAE. A multi-mode formulation is a way of describing non-smooth multi-model systems in terms of a finite number of smooth systems. The idea is to divide the state space of the system into different regions and associate a mode to each region. It is assumed that the system is described in terms of a single smooth model within each region. A simple example of a multi-mode DAE is:

$$\begin{array}{lll} \text{If } (x>0) & \text{then} & \\ & & f_1(x,\dot{x})=0 \\ & \text{else} & \\ & & f_2(x,\dot{x})=0. \end{array}$$

Unless clarity requires that we include it, we have omitted the time parameter t from the equations. In this example, the DAE has 2 modes: the first one is on when $x > 0$ and the second is on when the first one is not true. In general, a multi-model system is defined via some conditional statements (If-then-Else, Switch). In other word, a hybrid model may be composed of several models such that each model is valid in a certain region. As a physical example, consider a water tank with an open outlet.

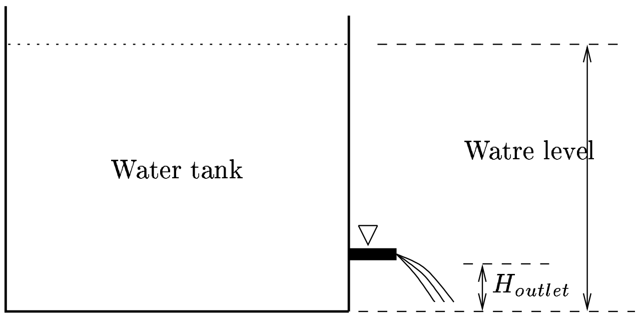


Figure 5.2.3: A water tank as a multi-model system.

In this system, x, the water level in the tank Fig. 5.2.3, can be expressed as a function of Q, outgoing water flow:

$$\dot{x} = k_1\, Q,$$

where k_1 is constant. Q can also be expressed as a function of x:

$$Q = \begin{cases} k_2\sqrt{x - H_{outlet}} & \text{if} \quad x \geq H_{outlet} \\ 0 & \text{if} \quad x < H_{outlet}. \end{cases}$$

Hence the behavior of this simple model is described by a multi-model system composed of two ODEs. This regional separation with smoothness assumption in each region is very important because non-smooth systems cannot be fed directly to the solver. Numerical solvers assume that the state variables and their first derivatives are continuous. If this assumption is not true, special precautions have to be taken at the discontinuity points. Because of the discontinuity, the discontinuous ODE/DAE cannot directly be fed to the numerical solver for integration. Since the linear multi-step method used by LSODAR/DASKR uses state variables and theirs derivatives over several integration steps to estimate the solution behavior, using two functions with previous Jacobian estimation deteriorates the convergence and most of the time will causes a failure in integration.

To avoid this problem, the numerical solver should use only one ODE/DAE up to the discontinuity point and then use the next ODE/DAE. In most cases the discontinuity is unpredictable, and it should be detected. To detect and localize the discontinuity time, solvers use zero-crossing functions that cross zero over the discontinuity point. For example, for the tank example we can use:

$$G_{zc}(x) = x - H_{outlet}.$$

With this discontinuity function, the solver can find the exact discontinuity time. However, to localize this point, the solver should step over this discontinuity point. Using the second set of ODEs beyond the discontinuity point, normally results in repeated step failure and a reduce in integration step-size to obtain a coherent derivative and to meet accuracy requirements. For some problems the solver gives up with an error message when the step gets too small, and in some other cases the solver may continue with an erroneous result. For example, consider the simple system modeled in Fig. 5.2.4. In Fig. 5.2.5-left we have the simulation result without considering the discontinuity and zero-crossing. On the other hand in Fig. 5.2.5-right, we have the correct simulation result. To sort this problem out a `mode` variable is associated with each zero-crossing function of a multi-model system in Scicos.

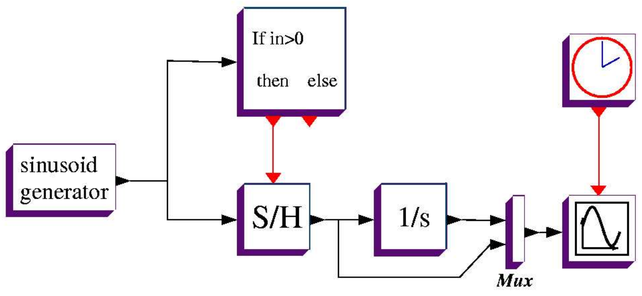


Figure 5.2.4: A model with discontinuity.

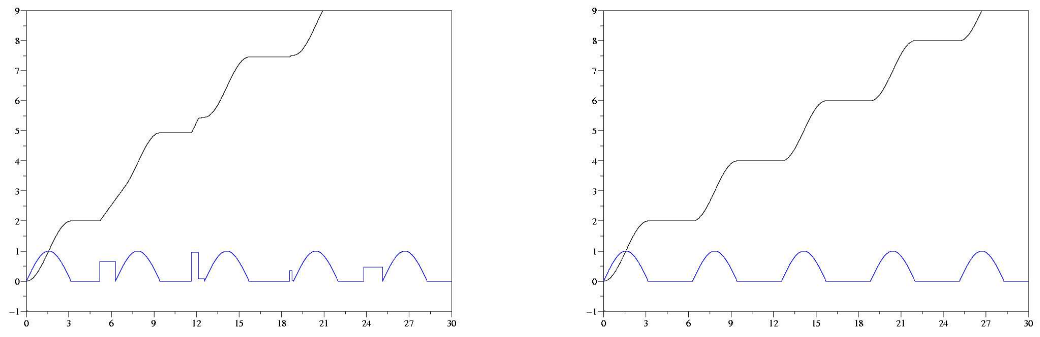


Figure 5.2.5: The erroneous (left) and correct (right) simulation results of model of Fig. 5.2.4

`Mode` Variable and Zero Crossing

We use the `mode` variable to assign and fix an equation set for every time interval between each two discontinuities. As an example, consider this ODE,

$$\dot{x} = \begin{cases} f_1(x) & \text{if} \quad g(x) \geq 0 \\ f_2(x) & \text{if} \quad g(x) < 0. \end{cases}$$

A general solution has been illustrated in Fig. 5.2.6, where x_a indicates the discontinuity point where $g(x_a) = 0$.

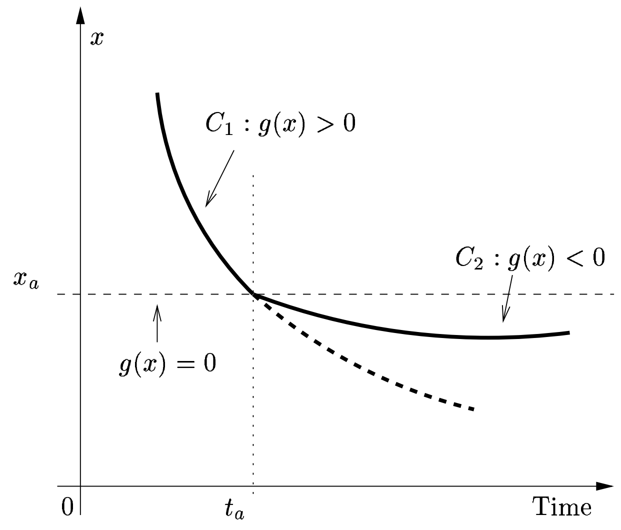


Figure 5.2.6: Water flow as a function of time.

Before reaching x_a, the solver is in `mode 1` , i.e., it uses $\dot{x} = f_1$, without considering the f_2 equation even when the solver needs to compute f_1 for $x > x_a$ (the thick dashed line). The solver employs the f_1 equation to integrate until it detects a zero-crossing. Then the solver localizes the crossing point and stops just over the discontinuity point (i.e., at $x = x_a^+$ and $t = t_a^+$). After detecting the zero crossing and stopping the integration, the `mode` variable should be updated in order to feed another equation set (i.e., f_2) to solve. For any discontinuity in the ODE/DAE we assign a `mode` variable to indicate which equation should be used before and after the discontinuity.

In general, when the numerical solver is called, the system of equations should not be changed. The solver should see a smooth set of ODE/DAE during integration. Instead, the zero-crossing functions that are the conditions to change the system of equations are examined by the solver to check whether they have changed or not. In case of any change, the `mode` variables should be updated to feed another set of equations to solver.

In Scicos, to integrate any ODE/DAE model with discontinuity, we assign systematically a `mode` variable to each discontinuity point, characterized by an `If-then-Else` block. For example, to simulate this system

$$0 = \begin{cases} f_1(\dot{x}, x) & \text{if} \quad g(x) \geq 0 \\ f_2(\dot{x}, x) & \text{if} \quad g(x) < 0, \end{cases}$$

we rewrite it in this form:

$$0 = \begin{cases} f_1(\dot{x}, x) & \text{if} \quad mode == 1 \\ f_2(\dot{x}, x) & \text{if} \quad mode == 2, \end{cases}$$

$$\begin{array}{lll} \text{if} \quad (g(x) \geq 0) & \text{then} & mode = 1 \\ & \text{else} & mode = 2. \end{array}$$

It should be mentioned that any `If-then-Else`, not resulting in a discontinuity, can be used without the `mode` variable. That is why there is a parameter in `If-then-Else` blocks that lets the user define whether the block is used with or without zero-crossing. For instance, in this system the `mode` variable is not necessary.

$$\dot{x} = \begin{cases} x^2 + x + 1 & \text{if} \quad x \geq 1 \\ 2x^2 - x + 2 & \text{if} \quad x < 1. \end{cases}$$

The use of the `mode` variables, eliminates the difficulties over discontinuities but there are still some problems concerning `mode` fixing and `mode` initialization that will be addressed in the following sections. But first, the `phase` variable which plays an important role in `mode` handling should be introduced.

`Phase` Variable

Using the `mode` variable to handle the discontinuities makes the block's mechanism a little complicated. For instance, consider the `saturation` block which is a common modeling object in control systems. The input/output characteristics of this block is plotted in Fig. 5.2.7.

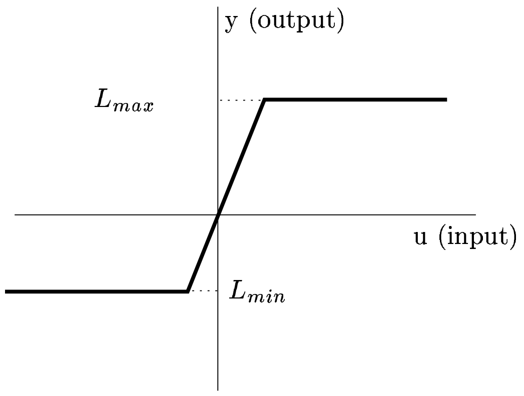


Figure 5.2.7: Input/output characteristic curve of saturation block.

This block doesn't have any state to disturb the solver with its possible discontinuity, but its output is characterized by piecewise continuous equations:

$$Output = \begin{cases} L_{min} & \text{if} \quad u \geq L_{min} \\ u & \text{if} \quad L_{min} < u < L_{max} \\ L_{max} & \text{if} \quad u \leq L_{max}. \end{cases}$$

The output of this block can be connected to the input of another block such as a block with internal continuous-time state which may result in a discontinuity during integration, see Fig. 5.2.8. To avoid this possible discontinuity, the `saturation` block or in general a block with discontinuous behavior should be used with a `mode` variable. In fact, its output function should be defined with a `mode` variable to provide a smooth input for the integrator block in Fig. 5.2.8.

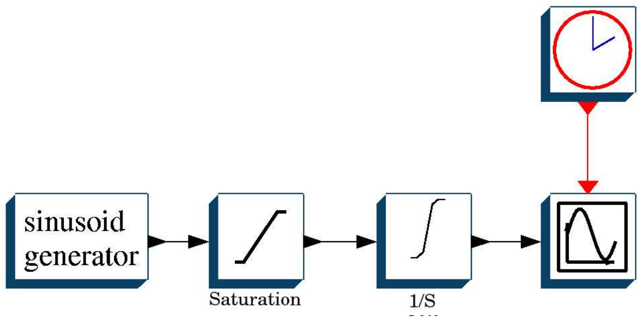


Figure 5.2.8: Saturation block feeding an integrator.

The saturation block can also be used without introducing a risk of causing discontinuity at the input of a block with internal continuous-time state, see Fig. 5.2.9.

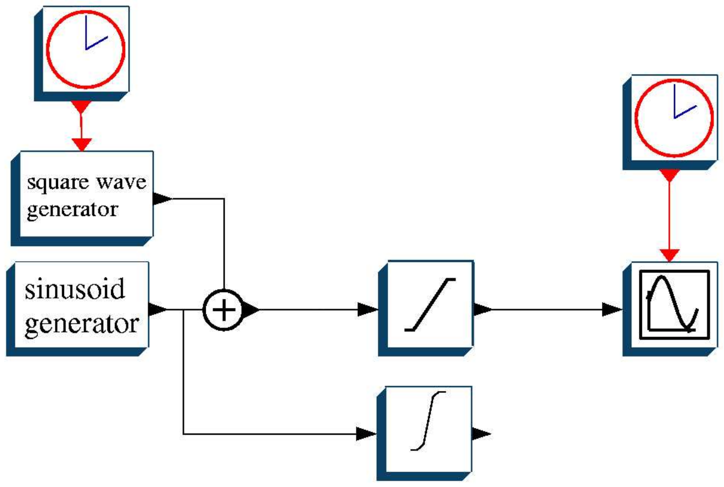


Figure 5.2.9: Saturation block with discontinuous input.

Here at any clock event time, there is a jump in the input of the `saturation` block. In this case the output should follow input changes without considering the `mode` variable. It seems that using `mode` variable in output updating should depend on who has requested the update. In general a Scicos block can be called by two sources; By the Numerical solver, and By the simulator.

To handle the problem of output updating in presence of `mode`, it would be reasonable to know who has requested the output update. We use the `phase` variable to indicate it. The `phase` variable has been introduced to distinguish these two cases.

Definition: `Phase` *is a global variable that indicates the current situation of the simulation. If the numerical solver is running, then the* `phase` *value is* **2**, *else* **1**.

With the `phase` variable, the output of the `Saturation` block can be written as follows :

$$Output = \begin{cases} \text{if } (\texttt{phase}{=}{=}1) \begin{cases} L_{min} & \text{if} \quad u \geq L_{min} \\ u & \text{if} \quad L_{min} < u < L_{max} \\ L_{max} & \text{if} \quad u \leq L_{max} \end{cases} \\ \text{if } (\texttt{phase}{=}{=}2) \begin{cases} L_{min} & \text{if } \texttt{mode}{=}{=}1 \\ u & \text{if } \texttt{mode}{=}{=}2 \\ L_{max} & \text{if } \texttt{mode}{=}{=}3, \end{cases} \end{cases}$$

that meets with all of the requirements.

`Mode` Fixing

The `mode` variables have been introduced to be used in conjunction with `If-then-Else` blocks to provide a smooth system for the numerical solver. The `mode` variables should be initialized and fixed before the numerical solver reads the ODE/DAE set. To initialize and fix them, the zero-crossing values are read. Suppose that $g_i(x)$ is the corresponding zero-crossing function of `mode[i]`, then `modes[0]` is initialized to `1` if $g_i(x) > 0$ and to `2` if $g_i(x) < 0$, i.e.,

$$\begin{array}{lll} \text{if} \quad G_i(x) \geq 0 & \text{then} & mode_i = 1 \\ & \text{else} & mode_i = 2. \end{array}$$

To explain how this has been implemented in Scicos, recall that the simulation function of a Scicos block is called with a special `flag` indicating the task to be performed. For instance,

- `flag 0:` Compute and Return the ODE function or DAE residuals.
- `flag 1:` Block output updating.
- `flag 9:` Compute and return zero crossing functions.

To integrate a model in Scicos, the numerical solver invokes the blocks containing the ODE/DAE, with `flag=0`. If the ODE/DAE contains any discontinuity, corresponding zero-crossing functions should also be defined in the block in `Flag=9`. The solver can read the zero-crossing functions by invoking the block with `flag=9` to detect and localize the discontinuity points.

The task of the `mode` setting can be performed by introducing a new `flag`, but it will not be a good choice. To do it in an efficient way it should be noted that the zero-crossing functions are used also by the numerical solver to locate the discontinuities and as we need to read the zero-crossing functions before `mode` fixing, it seems reasonable to set them just after reading the zero-crossing function inside the `flag=9`. But the zero-crossing functions are called by two sources:

1) Numerical solver, to detect and localize the discontinuity points
2) Simulator, to set and fix the block's `mode` variables before running the numerical solver.

During integration, the numerical solver calls the block with `flag=9` to monitor the zero-crossing functions, at the same time `modes` should not be changed. To avoid this undesirable mode setting, we use the phase variable. In fact, it is used to exclude access of the solver to mode settings.

Hence, in flag=9, modes can be set only if phase is 1. This pseudo-code illustrates what is inside a simulation function of a typical Scicos block.

```
Block(...)
  if (flag == 0) {
    ..
    res[i]=Fi(xd,x,t,mode[j])
    ..
  }

  if (flag == 1) {
    ..
    if (phase==1){
       output[k]=H(xd,x,t)
    }else{
       output[k]=H(xd,x,t,mode[j])
    }
  }

  if (flag == 9) {
    ..
    g[j]=Gj(xd,x,t)

    if (phase==1){
        if (g[j] >= 0) mode[j]=1;
                else mode[j]=2;
     }

  }
```

5.2.6 Multi-Mode DAE Initialization

Computation of consistent initial conditions becomes particularly difficult when the DAE is not smooth and, in particular, when it is a multi-mode system (defined with several modes). The search for the initial condition in this case is not just the classical problem of finding zeros of smooth functions but it is interlaced with searching for the correct mode set. Indeed, each mode set implies a smooth function and the solution of this function may imply another mode set.

The classification in terms of differential and algebraic is also a key element in the study of multi-mode DAEs, and in particular, the problem of finding consistent initial conditions since the classification specifies which variables are solved for. Consider, for example, the following multi-mode DAE

$$\begin{cases} \dot{x} = \sin t \\ \text{if } \; x < 1 \text{ then } \; y = 2 \;\text{ else } \; y = 3, \end{cases}$$

where in Scicos, we rewrite it as:

$$\text{ODE :} \qquad \begin{cases} \dot{x} = \sin(t) \\ \text{if } \texttt{mode[1]==1} \quad \text{then} \quad y = 2 \quad \text{else} \quad y = 3, \end{cases}$$

$$\text{Zero-crossing :} \quad \text{if } x < 1 \quad \text{then} \quad \texttt{mode[1]} = 1 \quad \text{else} \quad \texttt{mode[1]} = 2.$$

In this example, clearly x is differential variable and its initial value is given (chosen freely). On the other hand, y is an algebraic variable and its value is directly obtained from that of x. This multi-mode DAE is well-posed and it is easy to find consistent initial conditions for it. Indeed, in this case, the condition that needs to be tested involve the variable x which is given. Now consider the following multi-mode DAE

$$\begin{cases} \dot{x} = x + y + z \\ \text{if } \ y < 1 \text{ then } \ z = 0 \ \text{ else } \ z = 3 \\ \text{if } \ z < 1 \text{ then } \ y = 0 \ \text{ else } \ y = 3, \end{cases}$$

where in Scicos, we use:

$$\text{ODE :} \quad \begin{cases} \dot{x} = x + y + z \\ \text{if } \texttt{mode[1]==1} \text{ then } \ z = 0 \ \text{ else } \ z = 3 \\ \text{if } \texttt{mode[2]==1} \text{ then } \ y = 0 \ \text{ else } \ y = 3, \end{cases}$$

$$\text{Zero-crossing :} \quad \begin{cases} \text{if } \ y < 1 \quad \text{then } \texttt{mode[1]=1} \quad \text{else } \texttt{mode[1]=2} \\ \text{if } \ z < 1 \quad \text{then } \texttt{mode[2]=1} \quad \text{else } \texttt{mode[2]=2}. \end{cases}$$

This problem is more complicated because the conditions are based on the values of y and z (both algebraic variables), and the solution cannot be obtained without considering multiple scenarios. In fact, this system doesn't even have a unique solution for the algebraic variables in terms of the differential variables. If x is given a value, then (y, z) can be either $(0, 0)$ or $(3, 3)$. The discontinuity function may be a function of differential and algebraic variables and their derivatives, for example in general we may have:

$$\begin{aligned} \text{If } \ G_i(\dot{x}_d, x_d, \dot{x}_a, x_a, t) > 0 \ \ & \text{then } \ \texttt{mode[i]=1} \\ & \text{else } \ \texttt{mode[i]=2}, \end{aligned}$$

where x_d and x_a are the differential and algebraic states respectively and $\dot{x}_d$ and $\dot{x}_a$ are their time derivatives. In Scicos it is assumed that differential variables are known and the other variables should be computed. As a result, based on the dependence of discontinuity functions on these variables and derivatives, `mode` initialization can be divided into three cases:

- when $G_i(x_d)$, i.e., zero-crossing function is a function of x_d
- when $G_i(x_d, \dot{x}_d, x_a)$, i.e., zero-crossing function is a function of x_d, x_a, $\dot{x}_d$,
- when $G_i(x_d, \dot{x}_d, x_a, \dot{x}_a)$, i.e., zero-crossing function is a function of x_d, x_a, $\dot{x}_d$, $\dot{x}_a$,

In the following sections we will explain these cases and the way Scicos treat them.

`Mode` Initialization for $G(x_d)$

When the discontinuity functions depend only on differential states, it means that their discontinuity function values will not change during initialization. This case can be compared with the initialization of ODEs. In ODEs, all initial states are known, and in fact, just evaluation of discontinuity functions (zero-crossings) is enough to initialize the `modes`. If the discontinuity function is in the following form

$$\begin{array}{lll}\text{If} \quad G_i(x_d) > 0 & \text{then} & \texttt{mode[i]=1} \\ & \text{else} & \texttt{mode[i]=2},\end{array}$$

this initialization procedure can be used to select the initial `mode` set.

- **1)** Assigning the differential states, i.e., x_d
- **2)** Evaluation of zero-crossing functions and fixing the `modes`
- **3)** Invoking DASKR to compute consistent initial conditions, i.e., x_a, $\dot{x}_d$.
- **4)** `modes` and states are consistent. End.

`Mode` Initialization for $G(x_d, x_a, \dot{x}_d)$

In this case `mode` initialization is not as easy as the preceding case. In fact, this situation is the most frequent case in dynamical hybrid initializations. When the discontinuity functions depend on all existing variables in the DAE, the situation would become complicated. Because the value of the algebraic and the derivative of differential states may not be fixed during initialization. Furthermore, the initial value of x_a and $\dot{x}_d$ are not known, hence a dynamical system which contains n `modes`, say characterized by n *If then else* statements, can start in any of the 2^n possible modes. For the systems with large n and especially when the conditions are functions of algebraic variables, it is practically impossible to find the starting `mode` via trial and error.

If the discontinuity function is in this form:

$$\begin{array}{lll}\text{If} \quad G_i(x_d, x_a, \dot{x}_d) > 0 & \text{then} & \texttt{mode[i]=1} \\ & \text{else} & \texttt{mode[i]=2}.\end{array}$$

The following pseudo-algorithm can be used to initialize the `modes`:

- **1)** Assigning the differential states (x_d) and take the guess values of x_a and $\dot{x}_d$
- **2)** Based on x_d values and current x_a and $\dot{x}_d$ guess values, fix the `modes`
- **3)** If number of iteration exceeds `nmod`, Error `-17`, exit.
- **4)** Save the current `modes`
- **5)** Invoke DASKR to find the consistent initial conditions, i.e., x_a, $\dot{x}_d$
- **6)** Reevaluate the zero-crossing functions and fix the `modes`
- **7)** Compare new `modes` with stored ones, If there is any change, go to step **3**
- **8)** The `modes` and states are consistent, End

This procedure has been implemented in Scicos, but there are examples for which this algorithm does not converge.

`Mode` Initialization for $G(x_d, x_a, \dot{x}_d, \dot{x}_a)$

There are problems that need the derivative of an algebraic state to initialize the system. But in DASKR the derivative of an algebraic state is estimated only when integration is running and, of course, starting the integration needs consistent `modes` and initial conditions.

Backlash is such an example that its `mode` depends on a derivative of an algebraic state. When the backlash is in the engaged-up position and does not move, the initial `mode` depends on the direction of input, if input goes up, backlash follows the input, i.e., `mode`=*Engaged-Up*, and if input goes down, backlash's output doesn't move, i.e., `mode`=*DisEngaged*.

When the `mode` initialization depends on the derivative of an algebraic variable, i.e., the discontinuity functions to fix the `modes` are in the form :

$$\begin{array}{lll} \text{If} \quad G_i(x_d, x_a, \dot{x}_d, \dot{x}_a) > 0 & \text{then} & \texttt{mode[i]=1} \\ & \text{else} & \texttt{mode[i]=2}, \end{array}$$

the derivative of the algebraic variables can be obtained in two different ways:

1) Differentiation of the algebraic equations

For the first case, suppose that we have the following DAE :

$$\begin{aligned} 0 &= f(\dot{x}_d, x_d, x_a) \\ 0 &= g(x_d, x_a). \end{aligned}$$

Differentiating the algebraic equations yields

$$\begin{aligned} 0 &= f(\dot{x}_d, x_d, x_a) \\ 0 &= g(x_d, x_a) \\ 0 &= \dot{x}_d \, g_{x_d}(x_d, x_a) + \dot{x}_a \, g_{x_a}(x_d, x_a), \end{aligned}$$

that is enough to compute x_a, $\dot{x}_d$, and $\dot{x}_a$. Now, the trial and error procedure can be used to initialize the `modes`. In this method, it is required to know the algebraic equations. This can be done automatically or can be defined by the user. Furthermore, we may need the derivative of input time dependent signals.

2) Integration to obtain an estimation

If differentiation is not applicable on the problem or we don't distinguish the differential and algebraic equations, another possibility is running the solver for a very short time to obtain an estimate of derivatives of variables. The following procedure has been implemented and tested in Scicos.

- **1)** Assigning the differential states x_d and take the guess values of x_a and $\dot{x}_d$
- **2)** Based on x_d and current x_a and $\dot{x}_d$, fix the `modes`
- **3)** Save x in x_s
- **4)** Save the current `modes` for comparison
- **5)** With the fixed `modes`, invoke DASKR to find the consistent initial conditions, to obtain x_a, $\dot{x}_d$

- **6)** Run DASKR (disabling zero-crossings) for a short time to obtain $\dot{x}_a$
- **7)** Restore x_s to x
- **8)** With the new state variables, reevaluate the zero-crossing functions and fix the `modes`
- **9)** If there is a change in `mode` values, go to step **4**
- **10)** The `modes` and states are consistent, End

Non-Convex Problems and (-17) Error Message

Although the trial and error method is often successful, this simple method may fail in some non-convex problems. As an example consider the DAE index-1 problem (5.2.1) whose $x_a - x_d$ curve has been plotted in Fig. 5.2.10. Here we have three conditional statements, so we could have up to 2^3 `modes` but since the conditions are not independent, we have only 3 `modes`,

$$0 = F(\dot{x}_d, x_d, x_a) = \begin{cases} \dot{x}_d - 1 \\ \begin{cases} x_d - x_a - 2 & \text{if} \quad x_a < -1 \\ x_d + x_a & \text{if} \quad -1 < x_a < 1 \\ x_d - x_a + 2 & \text{if} \quad 1 < x_a. \end{cases} \end{cases} \tag{5.2.1}$$

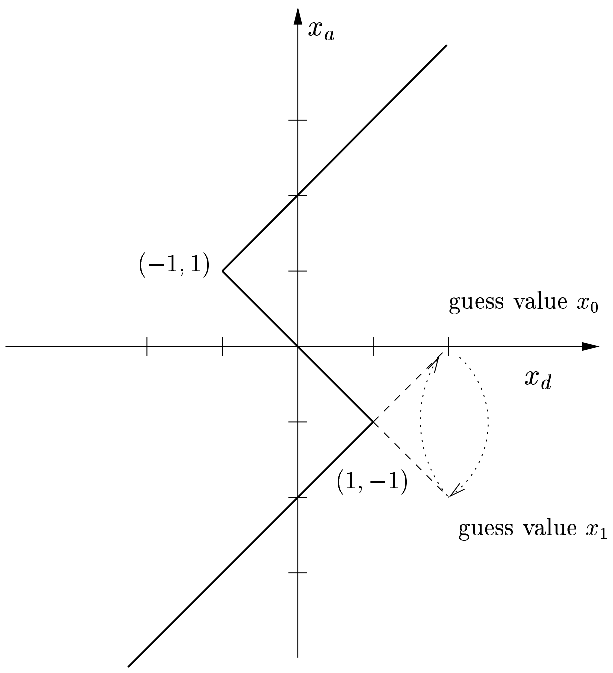


Figure 5.2.10: The algebraic part of the DAE (5.2.1).

In this case, the conditions are in terms of an algebraic variable (x_a) which is not known before complete initial condition calculation. Suppose we have a value for x_d and a guess value for x_a. If we start with ($x_d = 2$, $x_a = 0$), the solver (DASKR) cannot find the consistent initial condition, which is ($x_d = 2$, $x_a = 4$), using the above initialization routine, it falls in an infinite loop. If we begin with $x_d = 2$, and initial guess for $x_a = 0$. At the beginning, the second condition is correct, i.e., `mode =2`, which results in $x_a = -2$. This value is not consistent with current `mode` and another calculation needs to be done. In the next calculations, with $x_a = -2$ `mode` is **1** and $x_a = 0$ will be found. This value is also inconsistent with `mode=1` and another calculation needs to be done. In the subsequent tries the solver switches between the first and second mode without ever converging. Thus the solver falls into an infinite loop and switches between two adjacent conditions. To exit from this infinite loop, in the Scicos simulator, the number of consistent initial condition computation is counted and in the case where it becomes greater than the number of `modes`, A (**-17**) error message will be raised.

During integration, the zero-crossing surface can be used to detect the discontinuous point and switch between the equations. In initialization phase, however, an intelligent way to select the proper initial guess (or condition) does not always exist. The situation would be more complicated if the number of 'If then Else' conditions increases or the conditions become statements, as we have in our problems. In these cases, the convergence strongly depends on the initial guess. Note that in practical situations, often conditions depend only on differential variables which are supposed to be known thanks to the continuity assumption. In these situations, there is no problem. The problem is when the conditions depend on algebraic variables. In what follows, we show how to use the homotopy and piecewise-linear methods to compute consistent initial conditions in this case. In particular, we show how they can be used successfully on the above example.

5.2.7 Path Following Methods

Computing consistent initial conditions for DAEs involves solving a system of nonlinear equations. The usual methods used in DAE solvers for solving these equations are based on gradient methods and assume smoothness and a good initial guess. In the multi-mode framework, these methods often fail or exhibit convergence difficulties.

Recent application of continuation or homotopy techniques have shown them to give powerful methods for solving nonlinear equations when the initial guess is far from the solution. These methods are applied, in particular, to electric circuit models [**?**, **?**, **?**].

The homotopy method is used to solve a set of nonlinear algebraic equations $F(x) = 0$. To see how this applies to the problem of finding initial conditions for DAEs, let us assume that we have a semi-explicit index one DAE system which as we have seen in the previous section is characterized as follows

$$\begin{cases} \dot{x}_d = f(x_d, x_a) \\ 0 = g(x_d, x_a). \end{cases}$$

Here we have separated the state vector x into a vector x_d of size n_d, called the differential vector, and a vector x_a of size n_a, called the algebraic vector. In general, we can assume that the differential vector is known (it corresponds to physical states of the system which satisfy continuity properties). So what needs to be computed are the algebraic vector and the derivative of the differential vector. As noted earlier, we do not need a value for $\dot{x}_a$.

We are thus led to solving the following nonlinear system in $n = n_a + n_d$ equations and n

unknowns:

$$x = \begin{pmatrix} \dot{x}_d \\ x_a, \end{pmatrix} \tag{5.2.2}$$

and

$$F(x) \equiv \begin{pmatrix} f(x_d, \dot{x}_d, x_a) \\ g(x_d, x_a) \end{pmatrix} = 0. \tag{5.2.3}$$

The idea of the continuation method for solving $F(x) = 0$ is to introduce a continuation parameter to embed the problem into a larger system. This is done in such a way that this augmented system reduces to the original problem when the parameter is set to one; and when this parameter is set to zero, the augmented system is reduced to one that can easily be solved or has a known solution. This solution is used then as a starting point for solving the augmented system as the continuation parameter goes from zero to one.

The augmented system $H : R^{n+1} \to R^n$ is defined as follows

$$H(x, \lambda) = (1 - \lambda)F^0(x) + \lambda F(x) \quad \lambda \in [0, 1], \tag{5.2.4}$$

where $H(x, 0) = F^0(x)$ is chosen to be a simple function such that its root can be found easily. For example it can be $(x - x_0)$ where x_0 is the selected starting point. At $\lambda = 0$, the problem to solve is

$$H(x, 0) = F^0(x),$$

which is easy to solve, and at $\lambda = 1$,

$$H(x, 1) = F(x),$$

and the original problem is recovered. As λ moves from 0 to 1, numerical continuation methods trace the paths that originate at the solutions of the starting system toward the solutions of the target system.

To trace the solution from $\lambda = 0$ to $\lambda = 1$, there are several methods [?]. We will describe in brief the Predictor-Corrector and the piecewise linear methods.

Predictor-Corrector (PC) Method

The basic idea in Predictor-Corrector methods is to numerically trace the solution curve of $H(x, \lambda)$.

$$H(x(s), \lambda(s)) = 0 \qquad s: \text{ arc length parameter.} \tag{5.2.5}$$

The solution curve is a $Y(s)$ trajectory

$$Y(s) = \begin{pmatrix} \lambda(s) \\ x(s) \end{pmatrix}. \tag{5.2.6}$$

A predictor-corrector algorithms is performed in two steps:

1. Predictor: By use of current x and λ and their derivatives, their new values are predicted at $t + \Delta t$.

2. Corrector: The obtained estimates are corrected so that they remain on the $H(x,\lambda)$ curve.

So, it is enough to solve the following ODE

$$\frac{d}{ds}H(x(s),\lambda(s)) = 0, \tag{5.2.7}$$

with the following initial conditions:

$$\lambda(0) = 0,\ x(0) = x_0,\ \dot{\lambda} > 0,\ \| \frac{dY}{ds} \|_2 = 1.$$

If the Jacobian matrix (5.2.7) is full rank, then we have a unique solution [?, ?].

This formulation is a DAE

$$\begin{cases} \dot{\lambda}^2 + \dot{x}^2 = 1 \\ H(x,\lambda) = 0, \end{cases} \tag{5.2.8}$$

with the following initial conditions:

$$\lambda(0) = 0,\ x(0) = x_0,\ \dot{\lambda} > 0.$$

In the case where a DAE is smooth (no discontinuity) we can use any appropriate DAE solver to solve it. But in the multi-mode context, the nonlinear problem we have at hand is often not smooth. Fortunately the information about the points of non-smoothness is available. This allows us to still use a DAE solver but by making sure that the solver is restarted at the points of non-smoothness. This is done by introducing zero-crossing surfaces associated to these points as stopping tests for the solver. Note that while integrating each stretch, the `mode` must be fixed so that the solver does not see any non-smoothness.

It is desirable to use the integrators and the problem formulation already present in Scicos. A powerful DAE solver which can be used for implementing this homotopy method is DASKR [?] which is already used for simulation in Scicos. It has been used to implement the multi-mode homotopy technique described above with success on a number of problems. This technique may seem very time consuming and thus, not practical for simulation but it should be noted that Scicos keeps track of critical events (events which may cause non-smoothness) and issues a request for re-initialization only if such an event occurs. In general, events requiring a homotopy or other algorithm in place of the pseudo-algorithm are not very frequent.

As an example, Problem (5.2.1) is given to DASKR, with $x_d = 5$ and the algebraic guess value (starting point) $x_a = -10$. As it's been depicted in the fig.5.2.11, the λ value starts from zero when $x_a = -10$. As λ increases, x_a moves toward 0. At the discontinuity point ($x_a = -1$), DASKR stops the integration and homotopy is reinitialized with the new values of $x_a = -1$ and $\lambda = 0$. As it can be seen in the fig.5.2.11, the solution is $x_a = 7$ which has been reached at $\lambda = 1$. The homotopy method has enabled us to initialize this problem that the usual DASKR initialization failed on.

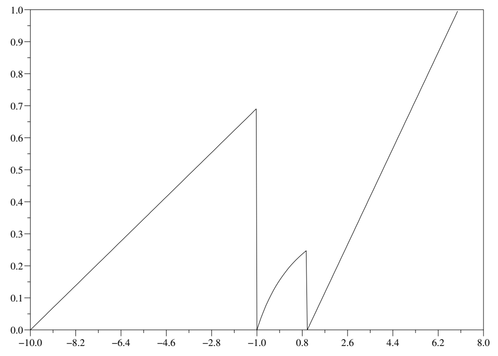


Figure 5.2.11: The λ variation for a homotopy solution of Problem (5.2.1).

As another example take This algebraic equation for which DASKR fails to obtain a solution with guess solution `[0,0,0]`.

$$\begin{cases} 0 &= x_3 - 1 \\ 0 &= x_1 + F(F(x+3) - F(-x_3)) \\ 0 &= x_3 - x_2, \end{cases}$$

where

$$F(x) = x\,\sin(x)\,\exp(-x).$$

To solve this equation with DASKR, we delivered the following code to be integrated until `L` crosses `1`.

```
delta(1)=(1-L)*(x1) + L*(x20-x3)
delta(2)=(1-L)*(x2) + L*(x1+F(F(x3)-F(-x3)))
delta(3)=(1-L)*(x2) + L*(x3-x2)
delta(4)=xd1*xd1+xd2*xd2+xd3*xd3+Ld*Ld-1.
```

Here the initial solution and directions are `[0,0,0]` and `[+1,+1,+1]` respectively. At `t=14.07` the solver reaches `L=1` and the final solution is `[-13.128, 1.0, 1.0, 1.0]`.

The Piecewise Linear (PL) Method

The piecewise linear continuation method is an alternative method to the classical PC methods. Whereas the PC method traces the exact solution curve of (5.2.4), in a PL method, a piecewise-linear curve approximates the solution curve. The piecewise linear methods require no smoothness of the underlying equations and hence may have a more general range of applicability than PC methods [?]. They are also well suited when the evaluation involves solving a set of inequalities.

The piecewise linear algorithm was originally introduced by Lemke and Howson [?] to solve the Linear Complementarity Problem (LCP). LCP arises in many fields such as quadratic programing and linear programing.

The problem is the following: Given a matrix $M \in \mathbb{R}^{k\times k}$ and $q \in \mathbb{R}^k$, find $u, y \in \mathbb{R}^k$ such that

$$\begin{cases} y = q + Mu \\ 0 \le y \perp u \ge 0. \end{cases}$$

This problem is denoted by LCP(q, M) and plays a major role in LCP theory [**?**]. We can interpret this system as a linear system with k switches. There are several methods for solving the LCP problem, see [**?**, **?**].

To get an idea of how a PL method approximates the solutions, these two definitions are cited from ([**?**]).

Definition: *Let $v_1, v_2 ... v_{N+1} \in \mathbb{R}^{N+1}$ be affinely independent points. Then the convex hull $[v_1, v_2, ..., v_{N+1}]$ is the N-simplex in $\mathbb{R}$ with $\{v_1, v_2, ..., v_{N+1}\}$ vertices. The convex hull $[w_1, w_2, ..., w_{r+1}]$ of any subset $[w_1, w_2, ..., w_{r+1}] \subset [v_1, v_2, ..., v_{N+1}]$ is an r-face of $[v_1, v_2, ..., v_{N+1}]$.*

Definition: *A triangulation τ of $\mathbb{R}^{N+1}$ is a subdivision of $\mathbb{R}^{N+1}$ into (N+1)-simplices such that*

1. *any two simplices in τ intersect in a common face, or not at all;*
2. *any bounded set in $\mathbb{R}^{N+1}$ intersects only finitely many simplices in τ*

After triangulation in $\mathbb{R}^{N+1}$ space, for $H(x, \lambda)$ a starting point and a proper simplex are chosen. To move from the starting point to the solution point, the algorithm must "traverse" through the neighboring simplex. There are several algorithms for finding the next simplex [**?**, **?**, **?**, **?**].

As an example of an implementation of the PL homotopy method of Eaves [**?**] and programed as an algorithm in Scicos, Problem (5.2.1) has been solved with this method. In Fig. 5.2.12, the produced simplices and the tracing path have been shown. Note that the algorithm stops around $x_a = 7$, which is the solution.

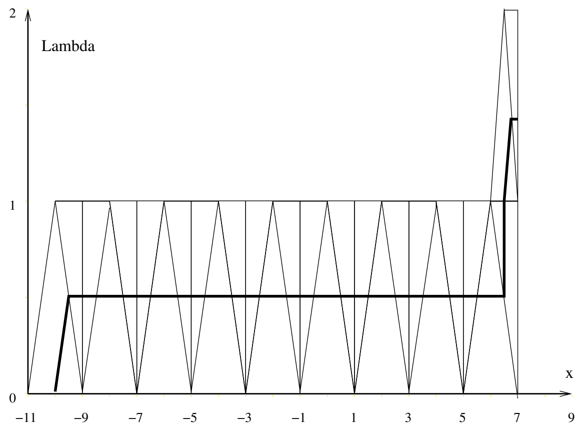


Figure 5.2.12: The simplices used to initialize Problem (5.2.1) with PL method.

Current Scicos Initialization Algorithm

In the previous section some of the issues involved with the implicit simulation of hybrid models, in particular, the problem of reinitialization and mode initialization were discussed. Three algorithms were presented. Some examples were given to show how the homotopy and

PL algorithms can successfully initialize where the simpler trial and error algorithm fails. But these methods are not good for every initialization problem.

The homotopy and PL method have also their own flaws and limitations. We used homotopy software introduced in [?] and HOMPACK90 [?]. In this software the initial direction is selected based on the initial derivative, which is not always a good choice. In one dimensional systems there are two direction, increasing or decreasing and just one of them yields the solution. The problem will be more complicated for multi-dimensional problems. For example, the last example explained in the *PC* method, will fails to reach $\lambda = 1$ if the initial direction `[-1,+1,+1]` is used, i.e.,

```
delta(1)=(1-L)*(-x1) + L*(x20-x3)
delta(2)=(1-L)*(x2) + L*(x1+F(F(x3)-F(-x3)))
delta(3)=(1-L)*(x2) + L*(x3-x2)
delta(4)=xd1*xd1+xd2*xd2+xd3*xd3+Ld*Ld-1.
```

For the homotopy PL method, we implemented PLALGO [?] in Scicos. In this software there are three possibilities for initial direction, (i.e., **I, -I**, and **Initial Jacobian**). But we observed that because of direction problem for ordinary problems, the trial and error method is more robust than PL and PC methods.

5.2.8 Scicos Simulator Flowchart

In this section, the Scicos simulator flowchart will be decomposed and given in several flowcharts.

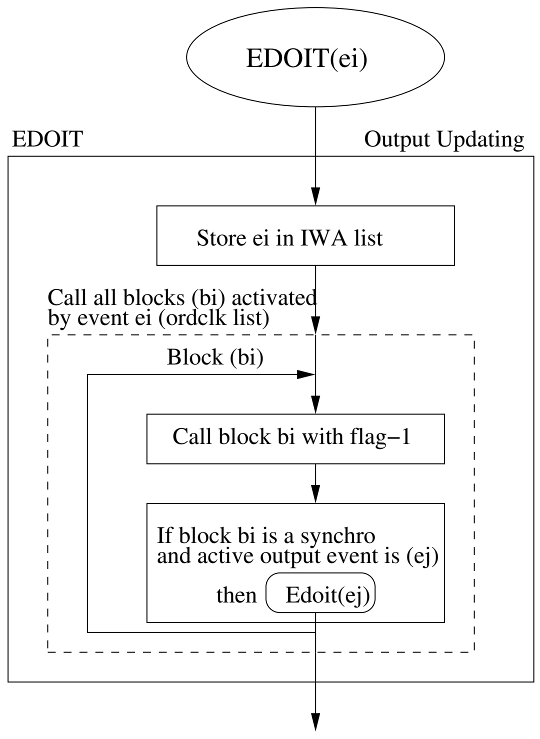

EDOIT(ei)
EDOIT
Output Updating
Store ei in IWA list
Call all blocks (bi) activated by event ei (ordclk list)
Block (bi)
Call block bi with flag−1
If block bi is a synchro and active output event is (ej)
then
Edoit(ej)

Figure 5.2.13: Scicos Simulator Flowchart: `edoit` function.

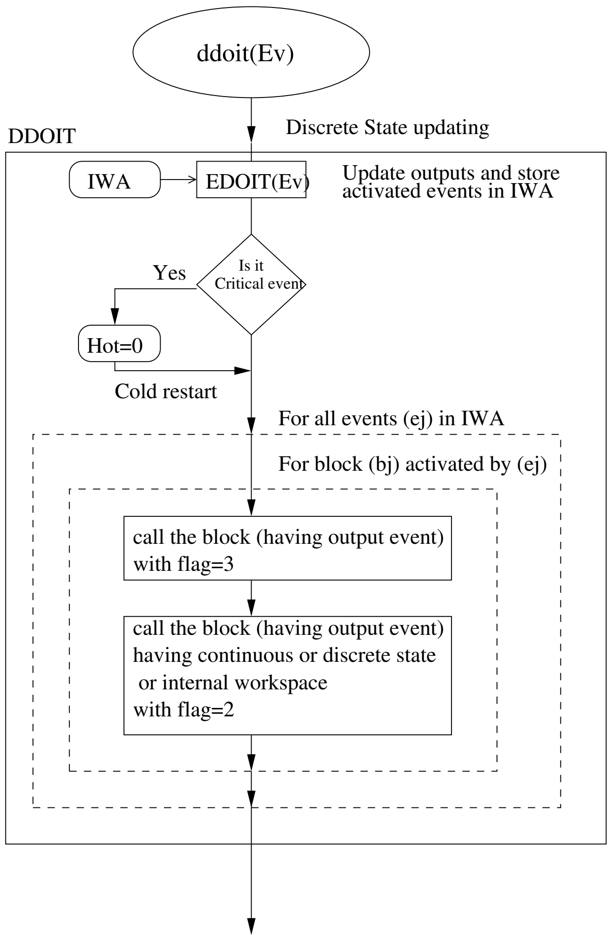

ddoit(Ev)
Discrete State updating
DDOIT
IWA
EDOIT(Ev)
Update outputs and store activated events in IWA
Is it Critical event
Yes
Hot=0
Cold restart
For all events (ej) in IWA
For block (bj) activated by (ej)
call the block (having output event) with flag=3
call the block (having output event) having continuous or discrete state or internal workspace with flag=2

Figure 5.2.14: Scicos Simulator Flowchart: `ddoit` function.

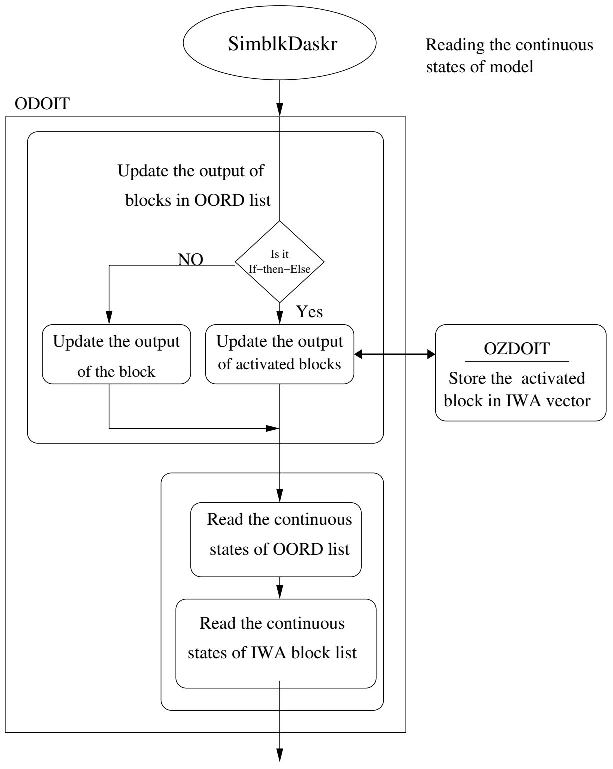

SimblkDaskr
Reading the continuous states of model
ODOIT
Update the output of blocks in OORD list
Is it If–then–Else
NO
Yes
Update the output of the block
Update the output of activated blocks
OZDOIT
Store the activated block in IWA vector
Read the continuous states of OORD list
Read the continuous states of IWA block list

Figure 5.2.15: Scicos Simulator Flowchart: `odoit` function.

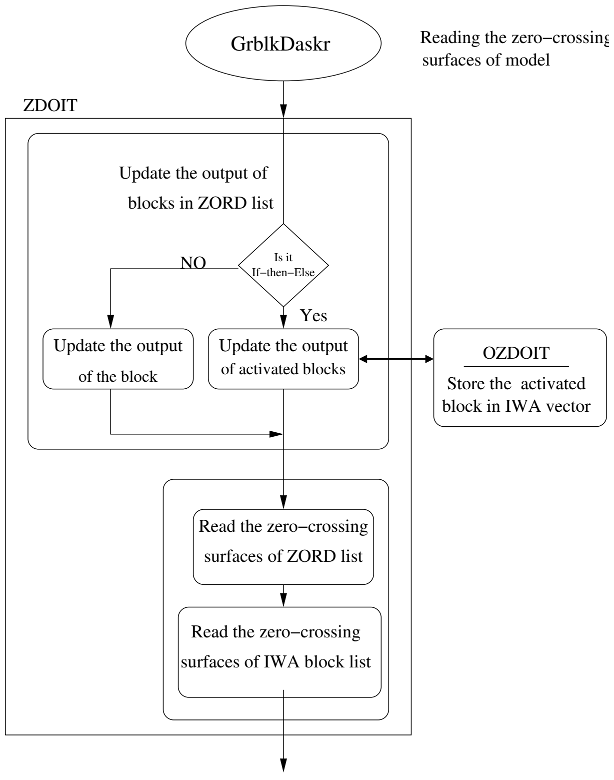


Figure 5.2.16: Scicos Simulator Flowchart: `zdoit` function.

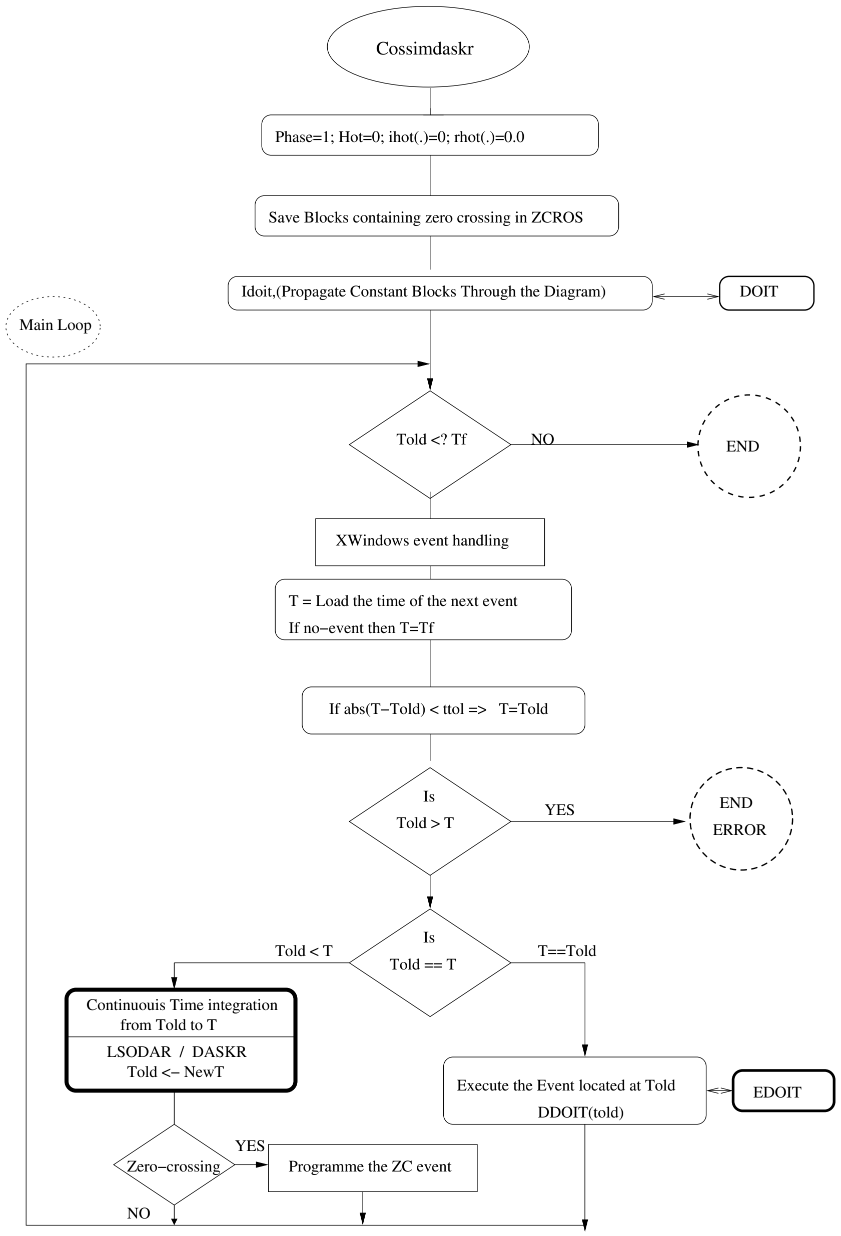

Cossimdaskr
Phase=1; Hot=0; ihot(.)=0; rhot(.)=0.0
Save Blocks containing zero crossing in ZCROS
Idoit,(Propagate Constant Blocks Through the Diagram)
DOIT
Main Loop
Told <? Tf
NO
END
XWindows event handling
T = Load the time of the next event
If no−event then T=Tf
If abs(T−Told) < ttol => T=Told
Is
Told > T
YES
END
ERROR
Is
Told == T
Told < T
T==Told
Continuouis Time integration
from Told to T
LSODAR / DASKR
Told <− NewT
Execute the Event located at Told
DDOIT(told)
EDOIT
Zero−crossing
YES
Programme the ZC event
NO

Figure 5.2.17: Scicos Simulator Flowchart: `cossimdaskr` function.

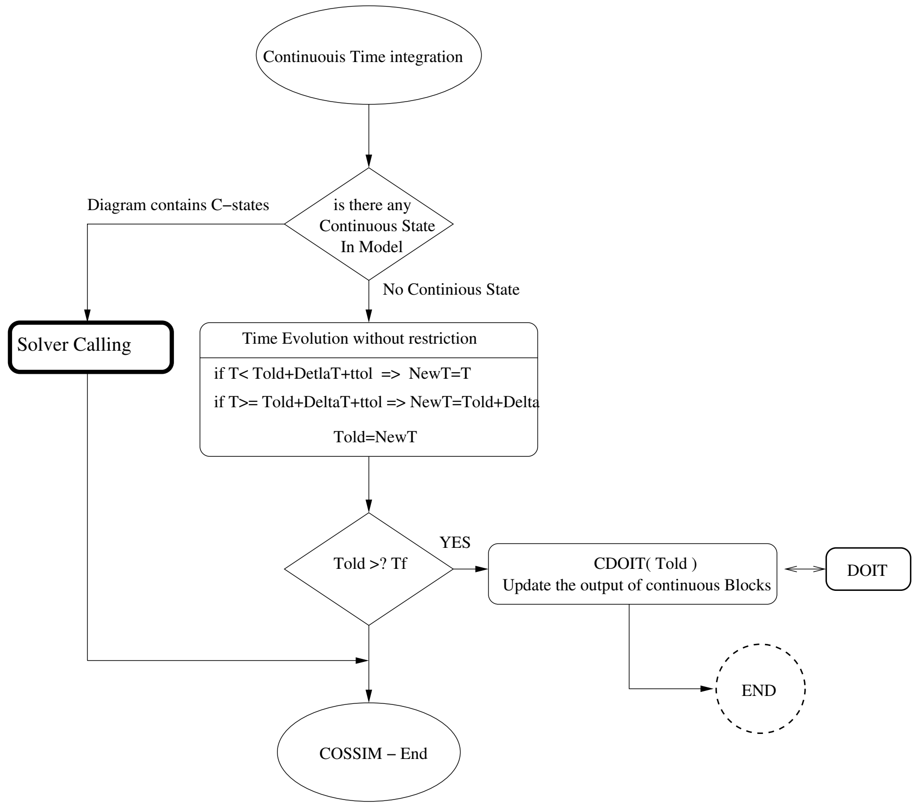

Continuouis Time integration
Diagram contains C−states
is there any
Continuous State
In Model
No Continious State
Solver Calling
Time Evolution without restriction
if T< Told+DetlaT+ttol => NewT=T
if T>= Told+DeltaT+ttol => NewT=Told+Delta
Told=NewT
Told >? Tf
YES
CDOIT(Told)
Update the output of continuous Blocks
DOIT
END
COSSIM − End

Figure 5.2.18: Scicos Simulator Flowchart: `cossimdaskr` function (continue).

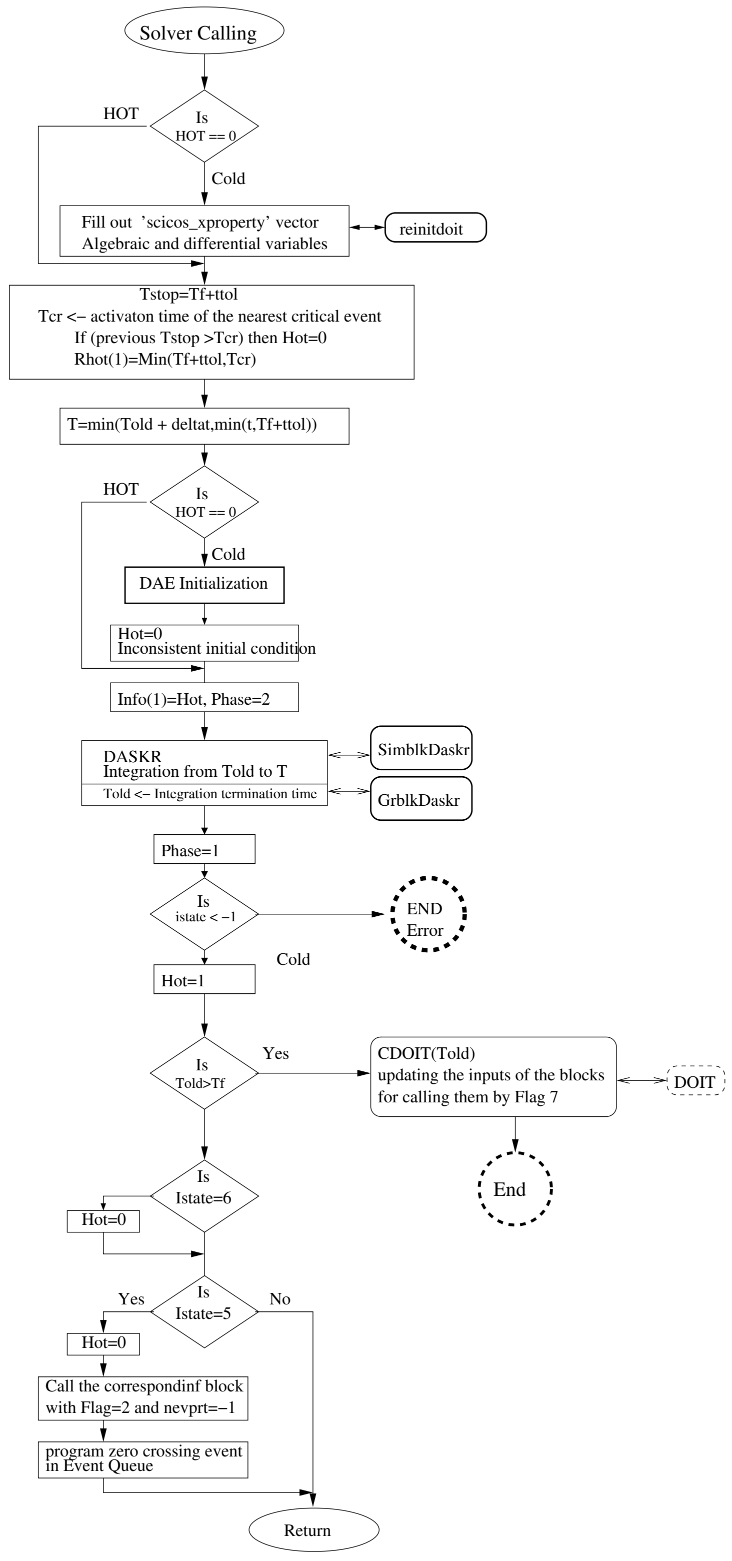

Solver Calling
HOT
Is HOT == 0
Cold
Fill out 'scicos_xproperty' vector
Algebraic and differential variables
reinitdoit
Tstop=Tf+ttol
Tcr <– activaton time of the nearest critical event
If (previous Tstop >Tcr) then Hot=0
Rhot(1)=Min(Tf+ttol,Tcr)
T=min(Told + deltat,min(t,Tf+ttol))
HOT
Is HOT == 0
Cold
DAE Initialization
Hot=0
Inconsistent initial condition
Info(1)=Hot, Phase=2
DASKR
Integration from Told to T
Told <– Integration termination time
SimblkDaskr
GrblkDaskr
Phase=1
Is istate < –1
END Error
Cold
Hot=1
Is Told>Tf
Yes
CDOIT(Told)
updating the inputs of the blocks
for calling them by Flag 7
DOIT
End
Is Istate=6
Hot=0
Yes
Is Istate=5
No
Hot=0
Call the correspondinf block
with Flag=2 and nevprt=–1
program zero crossing event
in Event Queue
Return

Figure 5.2.19: Scicos Simulator Flowchart: `cossimdaskr` function (continue).

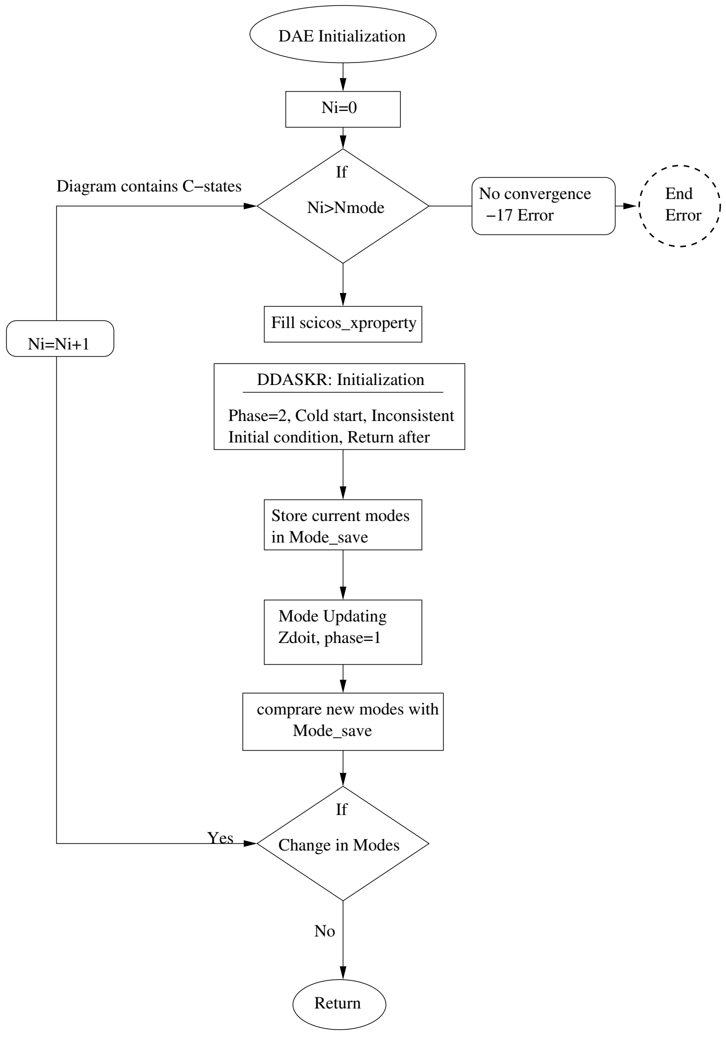


Figure 5.2.20: Scicos Simulator Flowchart: `cossimdaskr` function (continue).

Chapter 6

Scicos Extension

Modern technical systems like integrated circuits, mechatronic systems or distributed automation systems can be characterized as complex heterogeneous systems. Typically they show some of the following features:

- mixed-domain (mechanical, electrical, thermal, fluid, ... phenomena),
- Coupling between these domains, side effects, cross coupling,
- distributed and lumped (concentrated) elements,
- discrete-time and continuous-time signals and systems (in electronics: analog and digital),
- very large and stiff systems of differential equations to describe the continuous-time subsystems.

Depending on the level of abstraction, PDE and ODE/DAE are the mathematical models (system equations) of continuous-time subsystems. Many algorithms and computer programs are available for the numerical solution of the system equations. However, there is a demand for more and better assistance in finding the system equations. A powerful interdisciplinary modeling methodology is necessary to analyze real-world problems. Basically, the modeling process can be divided into two fundamental steps

- Modeling in the original physical domain: such as writing down the Kirchhoff laws in electrical circuit analysis.
- Transformation of the physical system model into a common mathematical form, i.e., Continuous-time and discrete-time equations (PDE, ODE/DAE).

At present, the second step is mostly well supported by simulators. So we will put our emphasis on the first step, the physically-oriented modeling of complex heterogeneous systems. The term physically oriented means: it is a goal of the modeling approach that as much as possible modeling steps be closely related to the design process of technical systems and to the intuitive procedure of the design engineer [?]. To achieve this objective, the capabilities of the hybrid dynamic system simulator Scicos has been extended to allow natural modeling of physical components. This chapter discusses these new features, the way they have been implemented in Scicos, and the new challenges that they bring in simulation.

6.1 Causal vs Acausal Modeling

Modular model construction can be classified in two major categories: causal and acausal. To make an analogy with computer programing languages, causal modeling corresponds to the use of assignment statements where the right-hand sides of the equations are evaluated and the result of the evaluation is assigned to the variables on the left-hand side of the equations. This corresponds closely to the evaluation of the outputs based on the inputs in a causal module. Acausal modeling on the other hand is closer to *equations* in the mathematical sense of the term. For an equation in a set of mathematical equations, it is not possible to classify (at least a-priori) its variables as inputs and outputs. In this case, for constructing the solution to the set of equations, the choice of the variable which should be computed in terms of the rest of the variables, in one particular equation, depends on the complete set of equations. In other words, a mathematical equation contains potentially a number of assignment statements; the choice of the statement which should be used for constructing the solution of the complete problem depends on all the equations. An acausal module is like a mathematical equation. It has ports of communication with other modules but these ports are not labeled a-priori as inputs and outputs.

Most physical components are more naturally modeled as acausal modules simply because physical laws are expressed in terms of mathematical equations. Consider a resistor in an electrical circuit. If the resistor is to be used as a module, it can only be an acausal module because depending on the way the resistor is used, the input can be a current and the output a voltage, or the input a voltage and the output a current (see Fig. 6.1.1).

$$\begin{aligned} v &= iR \\ i &= \frac{v}{R}. \end{aligned}$$

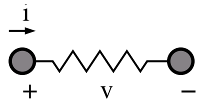


Figure 6.1.1: The electrical resistance symbol

In most modeling and simulation softwares, only causal modules (blocks) are considered. This situation is not only easier to implement but it provides a stronger notion of modularity which corresponds closely to the notion *subsystem* used by systems and control engineers. This is the reason why that way of modeling is referred to as **system level modeling**, as opposed to **component level modeling** where acausal modules (blocks) are used.

It is often possible to convert a component level model into a system level model by rewriting the equations and finding the appropriate causality structure in each module, but this task is time consuming, and error prone. Furthermore, the resulting (system level) model has very little resemblance with the original model making subsequent modifications difficult to realize. Indeed, a small change in the original model may require a complete redesign of the system level model.

To illustrate the difference between component level and system level modeling, consider the simple electrical circuit shown in Figure 6.1.2.

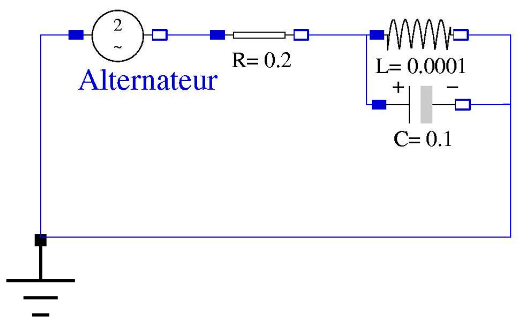


Figure 6.1.2: An electrical system to be modeled.

The circuit in Figure 6.1.2 clearly corresponds to a component level model. This is what an electrical engineer would like to work with. This circuit contains a voltage source, a resistor, an inductor, and a capacitor. To model and simulate this diagram, and display the voltage across the Capacitor in Scicos, we have to express the dynamics explicitly. We can use Kirchhoff's law which states that the loop voltage around a circuit must add up to zero, and using elementary models for each electrical component, we obtain:

$$\begin{aligned} 0 &= V_s + Ri + v_c \\ i_c &= C\frac{dv_c}{dt} \\ v_l &= v_C \\ v_l &= L\frac{di_l}{dt} \\ i &= i_c + i_l, \end{aligned} \tag{6.1.1}$$

where V_s is the alternator's voltage and i the current that passes through it. Where its corresponding equivalent system level model is depicted in Figure 6.1.3.

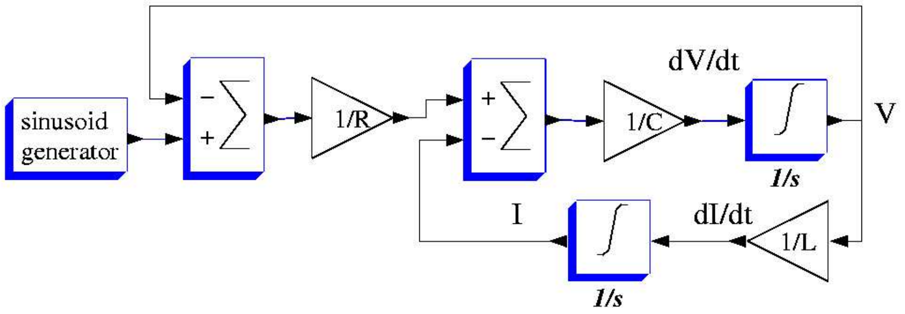


Figure 6.1.3: The system-level (causal) implementation of the model in Fig. 6.1.2.

We see clearly an important drawback in using such a model: there is no one to one correspondence between the original circuit elements and the blocks in the diagram. It is not difficult to see that, for example, replacing the resistor with a diode, would require starting over the construction of the system level model. This is an important point because even for this very simple example, the analysis to convert the intuitive physical model to a system level model is non-trivial and cumbersome.

It turns out, however, that in many situations, in particular in systems and control applications, the use of causal modules is very useful. As an example in Fig. 6.1.4, the F14 flight control system has been illustrated. This system is composed of several parts such as actuator, aircraft, control, feedback and some other dynamics. In such control systems it is much easier for the control engineer to have a system level model to use for the design of the controller and to optimize the control parameters.

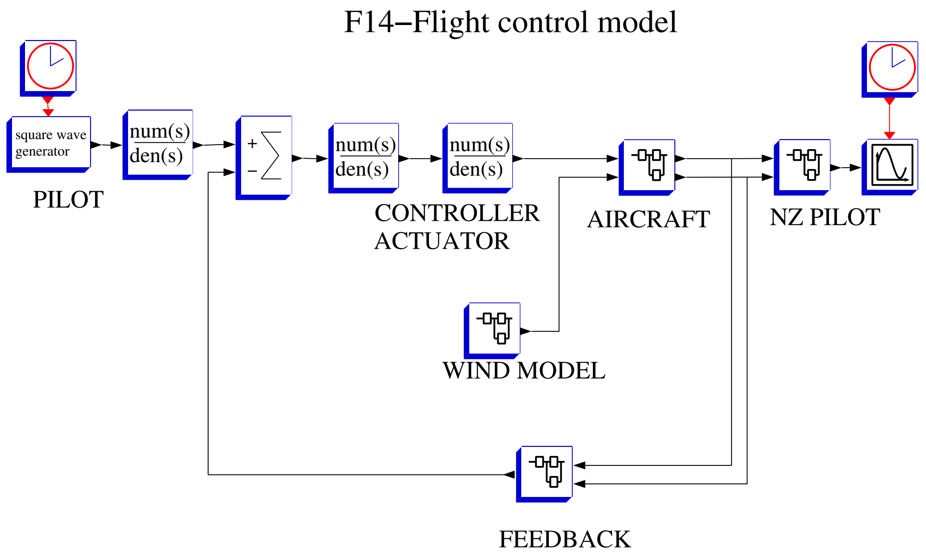


Figure 6.1.4: A system level modeling and design.

Even though causal models can in some sense be considered as special cases of acausal models, the restrictions imposed by causality implies that the behavior of the module can be implemented using an almost black box program which evaluates its outputs as a function of its inputs. The implementation of the behavior of the module in the acausal case is more complicated and requires either the construction of multiple causal modules or a formal description. In the latter case, the construction of the complete model requires formal manipulation of equations obtained from the description of acausal modules and their connection topology. It is for this reason that in Scicos we allow the co-existence of both type of modules: causal and acausal. So we can use both benefits of acausal and causal modeling in Scicos, e.g., we can design an acausal model controlled by a causal controller. To implement acausal models, *implicit blocks* has been introduced in Scicos, that will be explained in the next section.

6.2 Implicit Block vs Explicit Block

Implicit blocks have implicit ports. An implicit port is different from an input or an output port in that connecting two such ports imposes a constraint on the values at these ports but does not imply the transfer of information in an a-priori known direction as is the case when we connect an output port to an input port. For example, the implicit block `Capacitor` used in diagram of Figure 6.1.2 has current and voltage values on its ports but there is no way a-priori, without analyzing the full diagram, to designate any of them as input or output.

The behavior of an explicit block is given by an ODE or more generally a DAE:

$$\begin{cases} 0 = F(\dot{x}, x, u) \\ y = G(\dot{x}, x, u). \end{cases}$$

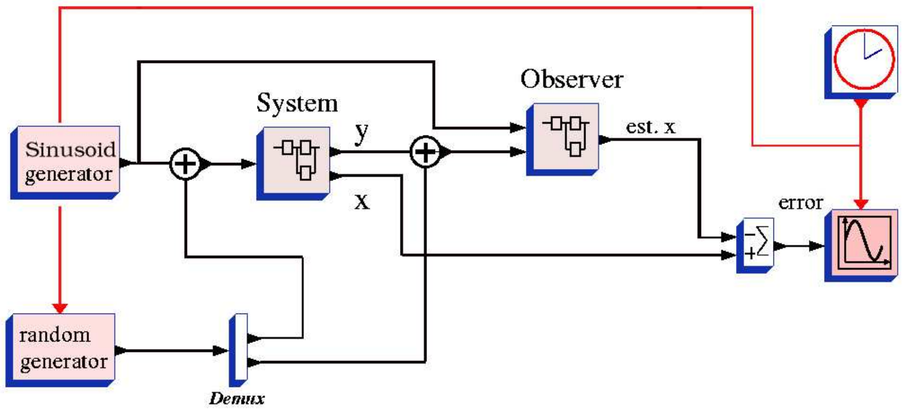


Figure 6.2.1: A system modeled in Scicos.

The model of Fig. 6.2.1 is composed of "explicit" blocks, i.e., block with explicitly identified inputs and outputs. Implicit blocks give the the user the ability to model physical systems without worrying about simplifying the equations and making them explicit; it is done automatically. All that the user should do is selecting the components and connecting them. Component level modeling allows the use of "implicit" blocks which are blocks with port connections which a-priori are not labeled as inputs or outputs [?]. Implicit blocks are essential for constructing models which include physical components such as resistors, capacitors, etc., in electricity, or pipes, nozzles, etc., in hydraulics. They are also useful in many other areas such as mechanics and thermodynamics. Implicit blocks are also called acausal blocks [?].

Contrary to explicit blocks, implicit blocks cannot be modeled as black box objects. The equations realizing the behavior of an implicit block must be available to the compiler for system reduction and code generation. To describe the behavior of implicit blocks and the construction of diagrams based on these blocks in Scicos, the *Modelica language* has been adopted [**?**].

6.3 Modelica Language

To extend the capacity of Scicos to allow component level modeling, we needed a language for describing the algebraic-differential constraints imposed on the input/outputs of the acausal blocks. We found Modelica [**?**] to be an excellent choice.

Modelica is primarily a modeling language that allows specification of mathematical models of complex natural or man made systems, e.g., for the purpose of computer simulation of dynamic systems where behavior evolves as a function of time. Modelica is primarily a modeling language, sometimes called a hardware description language, for specifying mathematical models of physical systems, in particular for the purpose of computer simulation. Modelica is a modern object-oriented programing language based on equations instead of assignment statements (it can thus be classified as a declarative language) This permits component level modeling to have a better reuse of classes since equations do not specify a certain data flow direction. Modelica has a multi-domain modeling capability, e.g., electrical, mechanical, thermodynamic, hydraulic, and control systems can be described by Modelica [**?**, **?**].

Modelica programs are built from classes, also called models. From a class definition, it is possible to create any number of objects that are known as instances of that class. A modelica class contains elements, the variable declarations, and equation sections containing equations. The equations specify the behavior of instances of that class. An example is a class specifying the model of a pendulum, composed of five equations:

$$\begin{aligned}
m\dot{v}_x &= -\frac{x}{L}F \\
m\dot{v}_y &= -\frac{y}{L}F - mg \\
\dot{x} &= v_x \\
\dot{y} &= v_y \\
x^2 + y^2 &= L^2.
\end{aligned}$$

This example is actually a mathematical model of a physical system, a planer pendulum, as depicted in Fig. 2.3.2. The equations are the Newton's equation of motion under influence of gravity, together with a geometry constraints. This is the Modelica model of the pendulum,

```
class pendulum ‘‘Planer Pendulum’’
  constant Real PI=3.1416;
  Parameter Real m=1, g=9.8, L=0.5;
  Real F;
  output Real x(start =0.5), y(start=0);
  output Real vx,vy;
equation
  m*der(vx)=-(x/L)*F;
  m*der(vy)=-(y/L)*F-mg;
  der(x)=vx;
  der(y)=vy;
  x^2+y^2=L^2;
end Peldulum;
```

6.3.1 Some Modelica Programing Features

Modelica is an object oriented programing language for modeling physical systems. To provide a software component model, Modelica includes the following three constructive elements:

- Components
- A connection mechanism
- A component framework

Components are connected via connection mechanism, which can be visualized in connection diagram. The component framework realizes components and connections, and ensures that communication works and constraints are maintained over the connections. For the systems composed of components or acausal blocks, the direction of data flow (the causality) is automatically deduced by the compiler at code generation time.

In this section we will explain some features of the Modelica language that are used to model hybrid dynamical systems. We will focus on those features who have been already implemented or are going to be implemented in Scicos.

Basic Language Elements

The basic structure element of Modelica is a class. There are seven restricted classes such as `model`, `type` and `connector`. All properties of a class are identical to the properties of all kinds of restricted classes. In Scicos only the `class` keyword is used.

In Modelica, basic data types are `Real`, `Integer`, `Boolean`, and `String`. They are built in `type` classes, with all properties of a class and the attributes are just parameters of the class. So far, Scicos accepts only the `Real` variables. With a specifier like `parameter` or `constant`, a component can receive a constant value and will keep being constant during simulation run-time. It can be changed when a component is reused or when the simulation is restarted.

The variables defined by `Real` keyword are continuous-time. The `discrete` keyword is used to define the discrete-time variables. The `start` keyword is used to assign an initial value to the variables. Here is an example of a class with different variable types.

```
class Myclass
  Real x, y (start=2.4);
  discrete Integer z1 ''a discrete variable'';
  discrete Real z2;
  parameter Real U=1 ``constant parameter'';
equation
  der(x)=12*x-3*y;
  x-12=U+z2;
  ....
end Myclass;
```

Connections

The `connector` keyword is a restricted class which can be used to define a physical connections, e.g., for the electrical circuits the Kirchoff's laws[1] are needed and some similar laws

[1]The currents of all wires connected at a node are summed to zero

are applied to flows in a piping network and to forces and torques in mechanical systems. By default, the connected variables are set equal. Such variables are called *across variables*. Real variables that should be summed to zero are declared with prefix `flow`. Such variables are also known as *through variables*, in Modelica we assume that such variables are positive when the flow (or corresponding vector) is into the component. The `connect` keyword is in the standard library and generates the equations by taking into account what kind of variables that are involved. Consider the definition of a `pin` in an electrical circuit:

```
connector Pin
  Voltage v;
  flow Current i;
end Pin;
```

Using this definition, we can define an electrical connection, i.e., a node.

```
Pin Pin1, Pin2;
connect (Pin1, Pin2);
```

This connection connects two pins so that they forms one node. This indicates a connection between two pins of two components. There is a built-in function `cardinality(C)` that returns the number of connections that has been made to a `connector(C)`. It is also possible to get information about the direction of a connection by using built-in function `direction(C)` (provided `cardinality(C)=1`).

Partial Models and Inheritance

A very important feature in order to build reusable classes is to define and reuse partial models. Since there are models which includes same types of components a class can be defined as a base for all these models using the keyword partial model. Using the keyword extends such a partial model can be extended or reused to build a complete model.

```
partial model TwoPin
  Pin p, n;
  Voltage v;
equation
  v = p.v - n.v;
  p.i + n.i = 0;
end TwoPin

model Inductor
  extends TwoPin;
  parameter Real L ( unit = 'H');
equation
  L * der(i) = v;
end Inductor
```

The feature is similar to inheritance in other languages. Modelica even supports multiple inheritance, i.e., several extends statements. Modelica does not use inheritance for classification and type checking. By inheriting all components of the base class, an extends clause can be used for creating a subtype relationship. But it is not the only way to create it. For example, if a class A is defined to be a subtype of class B and class A contains all the public components of B. B contains a subset of the components declared in A. This kind of subtype relationship is especially used for class parameterization.

Equations

Unlike most general purpose programing languages, the Modelica language is not primarily based on algorithms, but uses equations instead. For every model the programer can define a number of equations describing the properties of the model. The equations defines the relation between the different quantities in the simulation. The biggest reason why Modelica uses equation is the fact that every simulation problem in fact is a mathematical problem. It also give the language a high abstraction level, because an equation is often more intuitive than an algorithm. When Modelica compiles a model (which may consist of many interconnected models) it will put all these equations together in one equation system, making it possible to solve any variable. Modelica supports also the DAEs.

Matrices and Arrays

Besides solving equations using scalars it is also possible to use matrices and vectors. This makes it possible to define equations of matrices and lets Modelica solve them. This is very powerful in many applications such as PDEs. The usual algebraic operators $+, -, *, /$ works in the same way for matrices and vectors as for scalars, but division is only defined with scalars as denominator. To define a multidimensional variable (i.e., a vector or a matrix) the following syntax is used:

```
Real v[3]; // Defines a real vector of size 3.
Integer i[3][3]; // Defines a 3x3 integer matrix.
v = {1,1,1};
i = [1,0,0;
    0,1,0;
    0,0,1];
```

The 'For' Loop

In Modelica a sort of for-loop construct can be used to define the equations. It does not operate in the same way as a loop construct in an imperative language. Instead, it expands the equations in the loop to `N` equations (where `N` is the number of repeats in the loop). This is an example that shows how a value is assigned to the an array elements:

```
...
 Integer a[5]
...
a[1] = 1;
for i in 2:5 loop
  a[i] = a[i-1]*i;
end for;
```

The above `for` loop will be expanded to the following code:

```
a[1] = 1;
a[2] = a[1]*2;
a[3] = a[2]*3;
a[4] = a[3]*4;
a[5] = a[4]*5;
```

Discontinuous Models

It is possible to define discontinuous models in Modelica. Sometimes it is necessary to use different equations in different situations in a model. For example, one may need to set a limit to a variable or use a special case of a formula for a certain value. This can be done by using the `If`-statement in the equation clause. An example shows how the `abs()` function can be defined with the `If`-statement.

```
x2 = if x1 < 0 then -x1 else x1;
```

An `If`-statement often causes a discontinuity and the some precautions should be done to treat them and perform a simulation. There are, however, some `If`-statements that do not cause any discontinuity, so the numerical solver does not need to stop at these points. The `noEvent()` keyword is used to indicate this kind of `If`-statements, i.e.,

```
der(x2) = if noEvent(x < 1) then x
                            else x*x-x+1;
```

Discrete-time models

Modelica provides some specific features in order to model discrete-time systems. An important concept in discrete-time models is the notion of *event*. An event can, for example, be generated by the built-in function `sample`. The `sample` function takes two arguments, one `start` time and one `interval` and generates the events at specified time, i.e., the predictable events. An event is then generated at time instants `time=start+n*interval`. In order to generate unpredictable events the `when`-statement is used. In fact, to perform an actions at occurrence time of a certain event `when` statement is used. For example, the statement

```
when (x>2) then
```

defines a zero-crossing events (unpredictable or zero-crossing event). When the value of `x` becomes greater then 2, an event takes place and some actions will be done.

A specific built-in function that is used in discrete-time models, is the `pre()` function. This function will return the value of its argument (which is a discrete-time variable) before the sampling or the event. The Modelica code that returns the number of whole seconds passed since the start of the simulation follows:

```
model Seconds
  discrete output Real s = 0;
  parameter Real T0=1, Ts=1;
equation
  when sample(T0,Ts) then
     s = pre(s) + 1;
  end when;
end Seconds
```

At the initial time when simulation starts, `pre(z)=z.start` is identical to the start attribute value of its argument. Certain Modelica variables can be declared as constants or simulation parameters by using the prefixes constant and parameter respectively. Such variables do not change value during simulations. For a constant or parameter variable `c`, it is always `c=pre(c);` [?].

Discrete-time Event handling

To define zero-crossing events `when`-statements are used. When the event condition becomes true, a piece of code is activated and then executes certain actions, e.g., consider the `when`-statement in the following Modelica code.

```
  model disc
    discrete  Real z(start = 0);
    Real x(start = 0);
  equation
    x=Modelica.Math.sin(time);
    when (x> 0.1) then
      z =0.5*pre(z)-1;
    end when;
 end disc;
```

This is a Modelica model for

$$z(k+1) = 0.5\,z(k) - 1,$$

whenever the sinusoid signal (`x=sin(t)`) crosses `0.1` (in the positive direction), the value of z is updated. `When`-clauses are used to express equations that are only valid (become active) at events instants, e.g., at discontinuities or when certain conditions become true. In the body of `when`-statements discrete-time expressions can be used since they become active only at event instants. For example, two equations in the `when`-clause below become active at the event instant when the Boolean expression `x>2` becomes true.

```
when x > 2 then
  z2 = z1+1;
  z1 = pre(z1)+1;
end when;
```

In this example, at event times, the previous value of `z1` is used to compute the new value of `z1`, then it is used to compute `z2`.

The discrete-time variables in Modelica only change value at discrete-time points, i.e., at event instants, and keep their values constant between events. This is in contrast to continuous-time variables which may change value at any time, and usually evolve continuously over time. `when`-statement is used to update the discrete-time variables, but the continuous-time variables can also be reinitialized at event instants. The `reinit()` keyword is used to reinitialize the continuous-time variables.

```
when x > 2.0 then
  reinit(x,4.0);
end when;
```

When `x` becomes greater than `2`, the `reinit` statement assigns `x=4.0`.

Initialization Equations

A dynamic model describes how model variables evolve over time. For a well specified model, it must be possible to define the variable values at initial time in order to compute their value at `time>0`. To initialize the variables, a system of equations are used that are called *initialization equations*. These equations are used at `time=0` and assigns consistent initial values to all variables in the model, including derivatives of the continuous-time variables and `pre` of the discrete-time variables.

The `start` keyword is used to assign an initial value to the variables. In fact, `x(start=x0);` adds the equation (`x=x0;`) to the initialization equations. Consider this model,

```
class init
Real x (start=2.4, fixed=true);
...
equation
  der(x)=5-2*x;
end init;
```

During the initialization, the following equations are the initialization equations that should be solved.

```
der(x)=5-2*x;
x=2.4;
```

For the discrete-time variables the `z (start=z0)` statement adds the equation (`pre(z)=z0;`) to the initialization equations. Sometimes there are some constraints in the model that we cannot initialize all the variables of the system at the same time. For example, consider this model:

```
class init
  Real x (start=2.4);
  Real y (start=12);
equation
  der(x)=5-2*x;
  y=2*x;
end init;
```

In this model the statement (`y=2*x`) is a constraint, so the value of `x` and `y` cannot be initialized freely. In Modelica the boolean `fixed` keyword is used to specify which variable should be leaved constant during initialization. With the `fixed=true` we can select the variables that should left constant during initialization. For example, in the following model the value of `y` remains `12` and the value of `x` will become `6` after initialization.

```
class init
  Real x (start=2.4, fixed=false);
  Real y (start=12,  fixed=ture);
equation
  der(x)=5-2*x;
  y=2*x;
end init;
```

It should be noted that the derivative of continuous-time variables and the `pre` value of discrete-time variables cannot be initialized with `start` keyword. They can be initialized in `Initial Equations` construct. The `initial equations` construct provides a way to initialize all continuous-time variables. During initialization `der(.)` and `pre(.)` are treated as variables with unknown values. Complete initialization requires all equations used during simulation as well as `initial equations` to be solved. The following code is used to specify the derivative of a variable.

```
class init
  Real x (start=2.4, fixed=true);
initial equation
  der(x)=2*x-1;
equation
  der(x)=5-2*x;
end init;
```

In this model the initial values are computed with these equations:

```
der(x)=2*x-1;
x=2.4;
```

`initial()` statement is used to define the discrete-time equations that should be used in initialization equations. For example, this is a model for a monostable block. At the beginning of the simulation the block's output becomes `1`, if the input is `1` and the output value becomes `0` after two seconds. This initialization cannot be performed with the `start` keyword.

```
 when initial()  then
  y=u;
  Tr=if u=1 then t=2 else t=0;
elsewhen u=1 then
  y=1;
  Tr=time+2;
end when

when time>Tr then
  y=0;
end when
```

External functions

In Modelica, the user can use the functions that are in the library, e.g., in this code fragment some Mathematica functions are used in the equations.

```
y = Modelica.Math.asin(0.5);
der(x)=Modelica.Math.cos(y*time);
```

It is possible to call an extern function which has been implemented in C or Fortran. The body of an external function is marked with the keyword `external` in the Modelica external function declaration. The external function interface supports a number of advanced features such as in-out parameters, local work arrays, external function argument order, explicit specification row-major versus column-major array memory layout, etc. Here is an example that shows how the joint external C function of `multiply.mo` is defined and used in a Modelica class.

```
multiply.mo:
           function multiply
             input Real x, y;
             output Real z;
                external
           end multiply;

multiply.c:
           #include "multiply.h"
           double multiply(double x, double x) {
              z=x*y;
              return z;
           }

calling multiply() in a Modelica model
           class Use_multiply
             Real x1,x2,y;
           equation
             y=multiply(x1,x2);
           end Use_multiply;
```

6.3.2 A Modelica model Example

Here we explain how a complex system can be modeled in Modelica. As an example, suppose that the electrical system given in Fig. 6.3.1 has to be modeled.

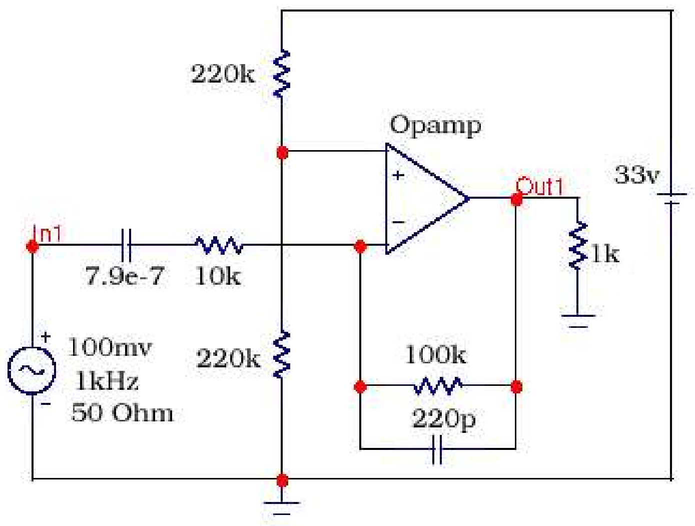


Figure 6.3.1: An audio amplifier stage with an Opamp.

The first step is the modeling of the very basic components, then connect them to construct a greater model or component. Therefore, the first step to model the electrical circuit is to model the basic electrical components in modelica. The Modelica model of ground, capacitor, operational amplifier, and sinusoidal voltage source follow.

```
connector Pin
  Voltage v;
  flow Current i;
end Pin;

class Ground "Ground"
  Pin p;
equation
  p.v = 0;
end Ground;

class Capacitor
  Pin p, n;
  Real v;
  Real i;
  parameter Real C=0.1 "Capacitance";
equation
  C*der(v) = i;
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end Capacitor;

class IdealOpAmp3Pin
  Pin in_p "Positive pin of the input port";
  Pin in_n "Negative pin of the input port";
  Pin out "Output pin";
equation
  in_p.v = in_n.v;
  in_p.i = 0;
  in_n.i = 0;
end IdealOpAmp3Pin;

class VsourceAC "Sin-wave V-source"
  Pin p, n;
  Real v;
  Real i;
  parameter Real VA=220 "Amplitude";
  parameter Real f=50 "Frequency";
  parameter Real PI=3.1415926 "PI";
equation
  v = VA*2*PI*f*time;
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end VsourceAC;
```

To construct the complete model, the component are connected as they are connected in a real electrical circuit. This can be compared to the way an engineer models an electrical circuit in SPICE [?]. The following listing is the Modelica program for the model of Circuit of Fig 6.3.1.

```
class Opampcircuit
  Capacitor      B1(C=7.9e-7, v(start=0));
  Ground         B2;
  VoltageSensor  B3;
  VsourceAC      B4(VA=0.1, f=1k);
  IdealOpAmp3Pin B5;
  Resistor       B6(R=50);
  Resistor       B7(R=10k);
  VoltageSensor  B8;
  Resistor       B9(R=220k);
  Capacitor      B10(C=220p, v(start=0));
  Resistor       B11(R=220k);
  Battery        B12(V=33v);
  Resistor       B13(R=100k);
  Resistor       B14(R=1k);
equation
  connect (B11.p,B2.p);
  connect (B4.p,B2.p);
  connect (B3.p,B2.p);
  connect (B8.p,B2.p);
  connect (B14.n,B2.p);
  connect (B12.n,B2.p);
  connect (B13.n,B1.n);
  connect (B5.out,B1.n);
  connect (B8.n,B1.n);
  connect (B14.p,B1.n);
  connect (B10.p,B9.n);
  connect (B3.n,B9.n);
  connect (B5.in_p,B6.n);
  connect (B1.p,B6.n);
  connect (B13.p,B6.n);
  connect (B7.p,B11.n);
  connect (B5.in_n,B11.n);
  connect (B9.p,B4.n);
  connect (B6.p,B10.n);
  connect (B12.p,B7.n);
end Opampcircuit;
```

6.4 Component level modeling in Scicos

To be able to use implicit blocks in addition to explicit ones in Scicos, several new features have been added to Scicos. So far, only two palettes with implicit blocks are available for testing purposes: the electrical and the thermo-hydraulic palettes. The thermo-hydraulics and electrical toolboxes and their available blocks are shown in Fig. 6.4.1 and 6.4.2.

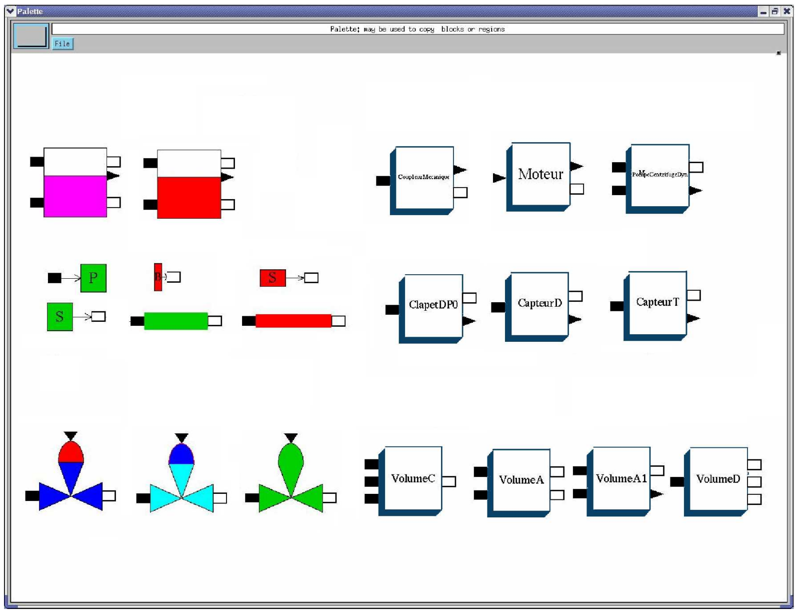


Figure 6.4.1: Thermo-hydraulics toolbox.

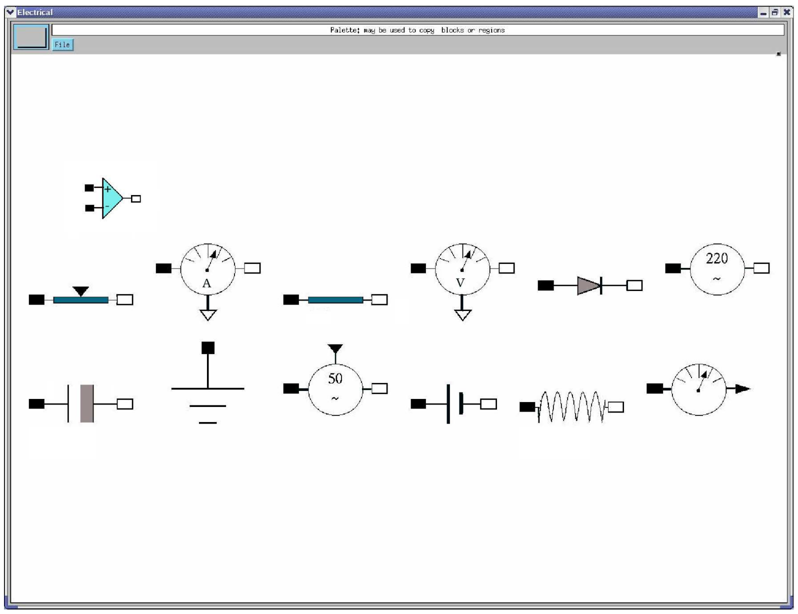


Figure 6.4.2: Electrical component toolbox.

Since the information available on the links connecting acausal models corresponds in general to physical quantities such as (voltage, current), (flow, pressure), etc, and are very different from information on links connecting causal models which transport signals, even a simple derivation placed on a link has a very different significance in two cases: a link that splits in two in the causal setting simply means that the information is duplicated whereas in the acausal setting, say in an electrical circuit, such a split must be implemented using the Kirchoff's voltage and current laws. Clearly it would be meaningless to allow the direct connection of a port of an acausal model to that of a causal model. That is why, special blocks were introduced in Scicos to be used as interfaces between the causal and the acausal world. Such blocks which have very clear physical significance, may have input ports, output ports *and* "acausal ports" An example of such a block is a voltmeter. This block has two acausal ports; physically they correspond to the place where the two wires are to be connected. But it also has an output where the measured value of the voltage is sent out of the block. In general, sensors and actuators make the interface between the component level part of the model and system level part.

Implicit blocks or components are interfaced via special links associated with physical quantities such as current or voltage in electronics, or, flow or pressure in hydraulics. As it is meaningless for a link representing a voltage to be connected to another link representing the output value of a PID controller, to distinguish between these two, two different link types have been defined: explicit and implicit links that interconnect explicit and implicit ports respectively. In Fig. 6.4.3 we have a hydraulic container which has four implicit ports (marked IP) representing liquid outlets and an explicit port (marked EO) representing a liquid level sensor output.

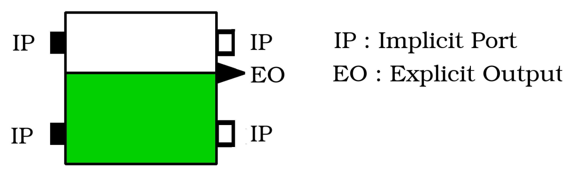


Figure 6.4.3: An implicit block can have implicit and explicit ports.

Implicit and regular Scicos blocks can be used in the same diagram. The addition of implicit blocks has been done without changing significantly the Scicos formalism. Even though implicit blocks can be used anywhere inside a Scicos diagram, they are grouped and replaced with a single block in a precompilation phase [?]. The mechanism, which can be compared to the way an `AMESim` [?] or `Dymola` [?] model is integrated in `Simulink`, is completely transparent to the user.

Consider for example the Scicos diagram in Fig. 6.4.4. Here we have a fluid level control system. To model this system in a natural way, a hydraulic source, a regulated valve, a container, a tube, and a well have been used. The container has a built-in level sensor which makes the interface with the explicit part of the system, similarly the valve is regulated through an input signal from the explicit part of model. The controller and the display mechanism have been implemented using explicit blocks and the blocks in gray are implicit blocks that have been developed in the Modelica language.

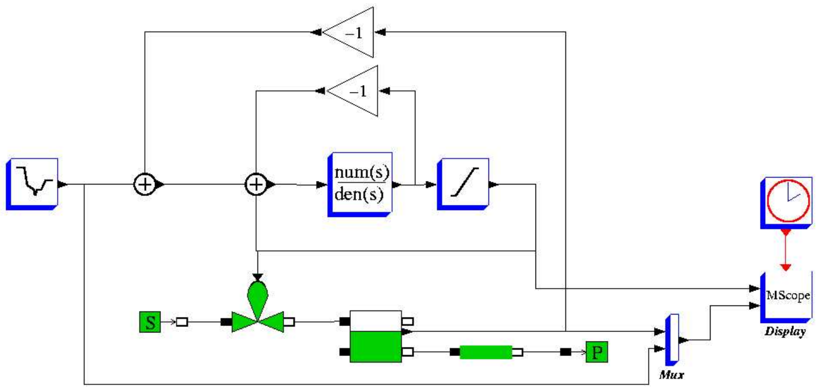


Figure 6.4.4: Scicos diagram containing both types of blocks.

6.4.1 Mixed Diagram Compiling

The way implicit blocks are handled by the Scicos editor is similar to the way regular blocks are. The compilation is again transparent for the user, however, the compiler performs a first stage compilation by grouping all the implicit blocks into a single internally implicit block, see Fig. 6.4.5 and 6.4.4. This is done by generating a Modelica program for the implicit part of the diagram.

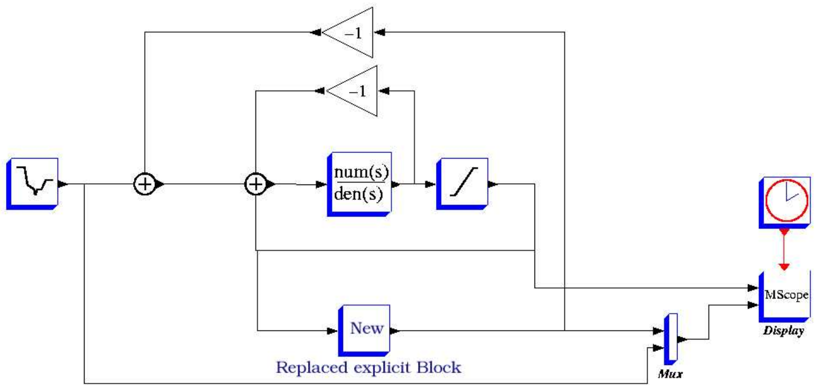


Figure 6.4.5: Scicos diagram of Fig 6.4.4; implicit parts are grouped into one explicit block.

For example, for the model of Fig. 6.4.4, all implicit block are grouped into a single block, see Fig. 6.4.5. The generated code expresses the behavior of the implicit part and is saved in a temporary file. For this model, the automatically generated Modelica code is:

```
class imppart_Hydraulics
  parameter Real P1;
  parameter Real P2;
  parameter Real P3;
```

```
    parameter Real P4;
    parameter Real P5;
    parameter Real P6;
    parameter Real P7;
    parameter Real P8;
    parameter Real P9;
    parameter Real P10;
    parameter Real P11;
    parameter Real P12;
    parameter Real P13;
    parameter Real P14;
    parameter Real P15;
    parameter Real P16;
    parameter Real P17;
    parameter Real P18;
    parameter Real P19;
    parameter Real P20;
    parameter Real P21;
    parameter Real P22;
    parameter Real P23;
    parameter Real P24;
    parameter Real P25;
    Source         B1(P0=P1, T0=P2, H0=P3, option_temperature=P4);
    VanneReglante  B2(Cvmax=P5, p_rho=P6);
    Bache          B3(Patm=P7, A=P8, ze1=P9, ze2=P10, zs1=P11, zs2=P12, z0=P13, T0=P14, p_rho=P15);
    PerteDP        B4(L=P16, D=P17, lambda=P18, z1=P19, z2=P20, p_rho=P21);
    Puits          B5(P0=P22, T0=P23, H0=P24, option_temperature=P25);
    OutPutPort     B6;
    OutPutPort     B7;
    InPutPort      B8;
  equation
    connect (B2.C1,B1.C);
    connect (B3.Ce1,B2.C2);
    connect (B4.C1,B3.Cs2);
    connect (B5.C,B4.C2);
    B3.yNiveau = B6.vi;
    B3.yNiveau = B7.vi;
    B8.vo = B2.Ouv;
  end imppart_Hydraulics;
```

The generated Modelica file is then processed by the a Modelica compiler which translates the Modelica code into a target code (in Scicos it is a C code). The generated C code describes the behavior of the implicit part of the model. Once the C code is compiled (this requires a C compiler) and incrementally linked with Scilab, Scicos sees this new block as a standard explicit block. In fact the new explicit (internally implicit) block replaces the implicit part of the diagram.

At the end of this procedure, the model is composed of only explicit blocks and can be compiled and simulated as usual. To illustrate our method, a flowchart given in Fig. 6.4.6 shows the compile process in Scicos. **Modelicac** is the Modelica compiler that is used in Scicos. Modelicac receives the automatically generated Modelica code for the Implicit part of the model and generates an equivalent C code for the implicit blocks.

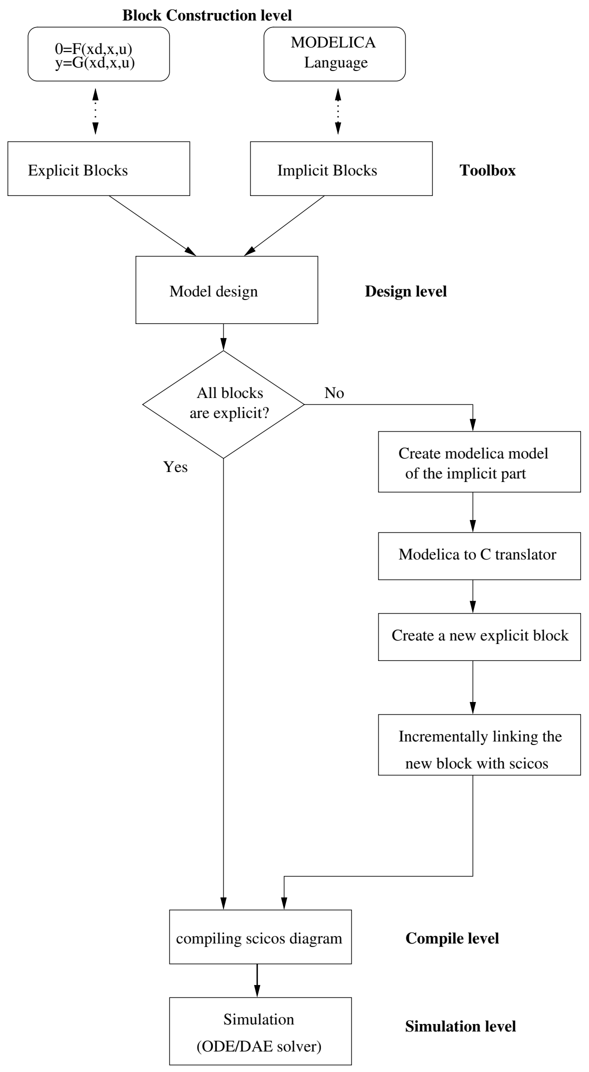


Figure 6.4.6: Scicos' Compiling and simulation flowchart.

6.5 Modelicac, a Modelica Compiler

Modelicac which stands for `Modelica` Compiler is a compiler for the subset of the Modelica language to cover the needs of simulating hybrid models in Scicos. `Modelicac` is an external tool, i.e., it is independent of Scilab, so one may use it like an ordinary compiler, e.g., like a C compiler. By default, `Modelicac` comes with a module that generates C code for the Scicos target. However, since `Modelicac` is free and open-source, it is possible to develop other code generators for other targets as well. `Modelicac` has been developed in Objective Caml which is a functional programing language developed at INRIA since 1985. This language is distributed with two compiler-development tools (Ocamllex and Ocamlyacc) which offer facilities to build compilers. Furthermore the Objective Caml compiler is free and open-source, that is why we adopted it to develop `Modelicac` [**?**, **?**].

Modelicac is invoked for two purposes: compiling basic models from libraries and generating code for the target simulation environment. To fulfill the first task, like generating an object file with a C compiler, Modelicac is invoked with the appropriate options from the command line to generate an object file with "*.moc" extension. The second task of Modelicac is compiling the "main" Modelica model (here provided by Scicos) and generating a code for the target (here, a C code). In this phase instead of generating an object file, Modelicac performs several simplification steps to generate a code as compact as possible. In Fig. 6.5.1 a flowchart shows how Modelicac generates a C file from Modelica model of a Scicos diagram.

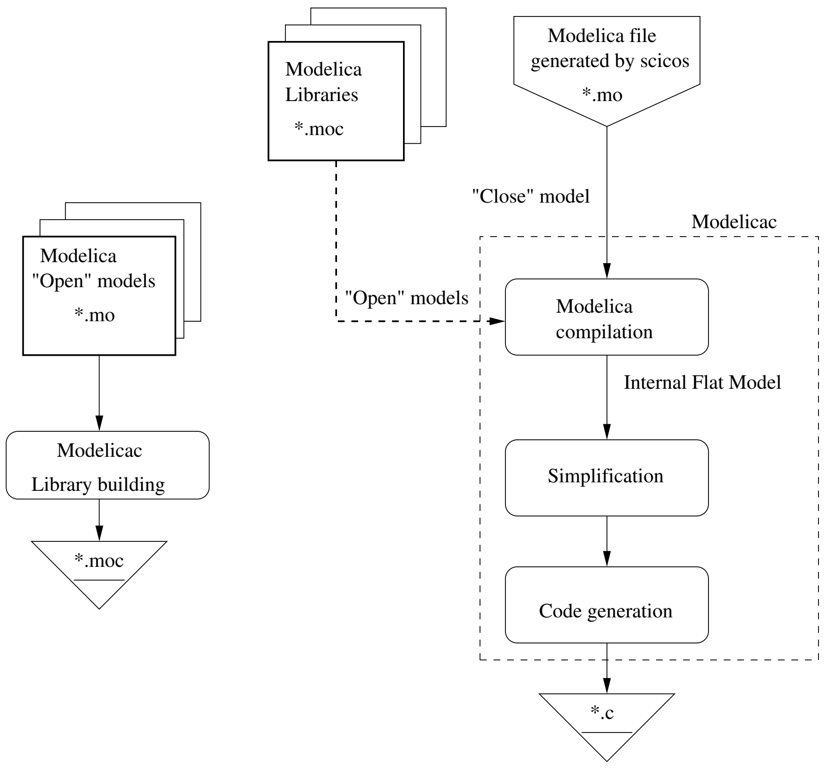


Figure 6.5.1: Modelicac translation flowchart.

6.5.1 Supported Modelica subset

The latest version of Modelicac (1.x.x) that has been used in this thesis does not support the full set of Modelica language constructs. It actually allows only the description of physical models at "equation" level. A physical model is built as the aggregation of sub-models or basic types with constraints between variables, and explicit event declarations (`When`). Currently Modelicac has the following main limitations:

- Only `Real` data type is supported.
- There is a minimum support for discrete-time variables, i.e., a discrete-time variable can be defined and updated in the when-statements.
- Inheritance is not currently supported.
- `Algorithm` is not supported but it can be defined as an external C function.

• `pre` and `fixed` keywords are not currently supported.

• `initial equation` construct is not currently supported. The user can only define the initial value for the differential variables and a guess value for the algebraic variables.

• The size of an array should be constant and cannot be a parameter.

• The `cardinality` and `direction` keywords are not supported.

6.5.2 Modelica Source Files

Modelica source files must contain only one class declaration, introduced either by the `class` keyword or by the `function` keyword. So a Modelica source file may define one of the following things:

• An `open` model is a model with free variables. There are more variables than equations (e.g., the model of a resistor in the electrical library). The open models are introduced by the `class` keyword,

• A `close` model is a model with an equal number of variables and constraints. It is also introduced by the `class` keyword,

• An external function, introduced by the `function` keyword.

It should be noted that only closed models can be simulated. To compile the `close` model `Modelicac` searches the libraries used in the current compilation directory and also in user-defined directories.

6.5.3 Model simplification

The following tasks are fulfilled by modelicac to simplify and generate a C source file from a Modelica source code and library object code files:

• Obtaining a flat model by replacing an aggregation of sub-models by the set of all their variables and equations merged together and replacing connection equations by ordinary equations. Symbolic manipulations in modelicac are performed using classical acyclic graph manipulation techniques

• Simplification of trivial or unnecessary equations using symbolic manipulations, e.g., in the following system

$$\begin{cases} \cos(x) + \sin(y) = 0 \\ \cos(x) - \sin(y) = 0 \\ z - x - y = 0 \\ f(x, y, z, v) = 0 \end{cases}$$

the first two equations are fully nonlinear and only the numerical solver can solve the system for x and y. But the third equation is trivial and z can be obtained in terms of x and y, so in the rest of equations z is replaced by $x + y$. Most of the variables used to connect Modelica components ("connection variables"), are eliminated in this way.

• Causality analysis, i.e., computation of system's Jacobian matrix. It will be explained further in the next section. s

Causality analysis

Causality analysis performs a few operations in order to find the so-called "strongly connected components" of a system of equations viewed as a directed bipartite graph [?]:

1. Constructing a bipartite graph whose nodes on the left represent variables in the system and whose nodes on the right represent constraints between variables (i.e., equations). There is an edge between a left-side node and a right-side node if and only if the variable represented by the left-side node appears in the equation represented by the right-side node.
2. Finding a coupling (using the Ford and Fulkerson method for instance)
3. Giving the edges an orientation depending on the results of the previous step. Edges that link two coupled nodes are all oriented in a given direction (either left-to-right or right-to-left) and the other ones in the opposite direction.
4. Finding strongly connected components in the resulting oriented graph (using Tarjan's algorithm for instance)
5. Sorting the resulting nodes in a topological order.

Each strongly connected component represents a subsystem of the whole system and it is now possible to perform symbolic simplification steps in order to reduce the number of variables in the system.

Modelicac simplification strategy

Symbolic simplifications typically involve variants of the Gauss method (to solve linear systems) and simple symbolic simplification methods based on a set of predetermined patterns (for efficiency reasons) to try to solve the remaining equations. In modelicac we focused on the second class of simplification methods. The problem when trying to solve a set of nonlinear equations is to determine a coupling in the bipartite graph described above that triggers as many simplifications as possible. So the Ford and Fulkerson (or equivalent) method is not enough for our purposes: instead of taking the first encountered coupling, we want in addition that the coupling satisfy a given criterion (e.g., maximizing the potential number of simplifications in the system) [?, ?]. Hence the use of a variant of the Hungarian method which can be seen in modelicac as a method for finding a coupling based on an additional constraint called the "satisfaction" [?, ?]. Practically, that is done in modelicac by associating a set of pairs (variable, weight) with each equation: given an equation, each weight indicates whether the equation is "easy" to solve with respect to its associated variable or not. For instance, if an equation contains only one variable, the weight associated with that variable is low whereas the weight associated with any other variable is infinite. Since modelicac associates low weights with variables that appear in linear systems, the Hungarian method "discovers" linear systems by itself and symbolic substitution techniques, when applied to those linear systems, achieve the same effect as Gaussian elimination. Even though the Gaussian elimination algorithms is not considered in modelicac, the results are satisfactory.

6.5.4 Scicos vs Modelica

In chapter (3), we discussed about Scicos and internal architecture of a Scicos block. Here we will see the correspondence between Modelica constructs and the Scicos blocks.

Variables and Equations: The variables defined by `Real` keyword are continuous-time variables in a Modelica code. These variables are equivalent to the continuous-time states that are defined in the ODE or DAE equations in a Scicos block. The derivative of a variable, defined with `der` operator, is equivalent to the the derivative of a continuous-time state. The `discrete` keyword is used to define the discrete-time variables in Modelica. The variables declared with `Real discrete` in Modelica are equivalent to the discrete-time states in Scicos. During code generation, Modelicac uses this correspondence to generate differential and difference equations.

If-statement: A discontinuity in model is modeled with an `If`-statement in Modelica and `If`-statements that do not cause discontinuity are used with a `noEvenet`. In a Scicos block, a discontinuity should be used with a `mode` variable and a `zero-crossing` function to locate the discontinuity. The `mode` variable is update in `flag=9, phase==1`. For a Modelica code such as

```
f3 = if x1 < 2 then f1 else f2;
```

Modelicac generates the following code:

```
...
res[.] = f3 - (mode[0]==1)?  f1; f2;
...
if (flag==9){
  g[0]=-x[0] + 2;
  if (phase==1) then{
    if (g[0]>0) mode[0]=1; else mode[0]=2;
  }
}
```

If this `If`-statement is specified with `noEvent`, then the following code will be generated.

```
res[.] = -x[0]+2>0 ? f1; f2;
```

when-statement: `when`-statements are used to define unpredictable or zero-crossing events. A when construct has two parts: a condition and an action to perform when the condition becomes true. This construct is in fact used to generate an event and then do some actions. In Scicos, a zero-crossing function can be used to generate such an event. When this event is activated, the block is called with `flag=2`. In order to distinguish between a call due to an external event and a call due to an internal zero-crossing event, the block is called with `flag=2` and a negative `nevprt` (external event have always a positive `nevprt` that indicates the activating input event port). Here is an example of a `when`-statement and the C code that has been generated by Modelicac.

```
// Modelica source code:
when x > 2.0 then
  z1 = z1+1;
  reinit(x,2.0);
end when;


// the generated C code:
......
if (flag == 2 && nevprt < 0) {
    if (jroot[0] == 1) {
       x[0]=2.0;
       z[1] = 1.0+z[1];
    }
  }
  ....
  if (flag == 9) {
    g[0]=x[0];
  }
```

Modelica programs do not generally contain sequences of commands like program written in classical programming languages such as FORTRAN. Most of the time, statements in Modelica denote constraints to be verified during simulation, except when explicitly introduced by the `algorithm` keyword[2]. Therefore, there are some points that the user should be aware about. That is the execution order of statements in a `when`-statement and the execution order of the `when`-statements that may be executed at the same time. For example, for this Modelica code,

```
discrete Real z0 (start=0);
discrete Real z1 (start=0);
...
when (time>2) then
  z0 = z1+1
end when;

when (time>2) then
  z1 = z1+1
end when;
```

the generated Scicos code can be either

```
if (flag==2 & nevprt < 0){
  if (jroot[0]==1){
    z[1]=z[1]+1;
  }
  if (jroot[1]==1){
    z[0]=z[1]+1;
  }
}
```

or

```
if (flag==2 & nevprt < 0){
  if (jroot[0]==1){
    z[0]=z[1]+1;
  }
  if (jroot[1]==1){
    z[1]=z[1]+1;
  }
}
```

[2]`Algorithm` is not discussed in this thesis

In this example, two `when`-statements are executed at the same time, so their execution order is undetermined. After executing the event, the value of `z0` can either be `2` or `1`.

6.5.5 A Code Generation Example

First, we present the Modelica source code of a few electrical models from electrical library and then show how to use these models to construct and compile elaborated electrical models with modelicac.

Connectors

In Scicos libraries "connectors" are the most basic open models. Each particular domain (e.g., electrical, hydraulic, etc.) has a its own connectors that connect two or more models and exchange quantities. The are two connector types:

- Internal connectors, that allow connection of two Modelica components, such as "p" and "n" pins used in electrical resistor model.

```
class Pin
  Real v;
  flow Real i;
end Pin;
```

- External connectors, that allow communication of a Modelica component with an external environment (Explicit part of model in Scicos environment, for instance). Instances of "InPutPort" and "OutPutPort" are examples of these connectors types

```
class InPutPort
  input Real vi;
end InPutPort;

class OutPort
  output Real v;
end OutPort;
```

These types of connectors are used in sensor and actuator blocks that can be seen in Fig. 6.4.4 and 6.5.2.

Main class

In order to use these models to construct and compile elaborate electrical models with `Modelicac`. In order to perform the simulation of an electrical circuit one normally has to describe the circuit using Modelica by defining the components involved (i.e., giving their names and the value of their parameters) and the connections to establish. Then, `Modelicac` should be invoked with the appropriate options and arguments. This task is done by Scicos, provided that the appropriate library exist in Scicos.

Like other compilers, Modelicac can generate intermediate object files with `*.moc` suffixes. These files contain all necessary information about the compiled Modelica file. These object files are used to generate a target C file for Scicos. Modelicac can be evoked form command line as follows,

```
modelicac [-c] [-o <outputfile>] <inputfile> [options]
```

where

- `-c`, to compile the file without C code generation, i.e., generate `*.moc` file.
- `-o <outputfile>`, to give a name to the output file.
- `-jac`, to generate the analytical Jacobian of the DAE.
- `-keep-all-variables`, to generate the code without simplification.
- `-L <directory>`, to indicate the directory of library files (`*.moc`).
- `-hpath <directory>`, to indicate the directory where the external function files have been stored.

In Scicos, all the process of calling Modelicac is automatic. In fact, it is not necessary to write any Modelica code to build a model. The user can assemble components using the Scicos editor and then Scicos automatically builds the Modelica source code from the graphical specification and invokes `Modelicac` to convert Modelica code into C code. Fig. 6.5.2 shows the model of electrical circuit of Fig. 6.3.1 implemented in Scicos using implicit blocks.

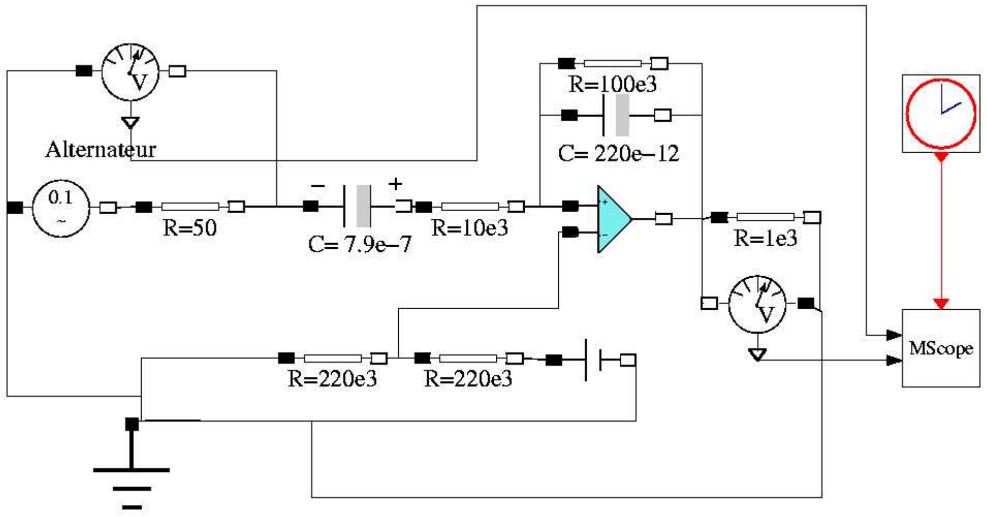


Figure 6.5.2: Scicos' Implementation of electrical circuit of Fig. 6.3.1.

For this model `Modelicac` generates a C code. The generated C code is composed of several 'sections', defined by `flag` value and perform a predefined task. For example, in section where (`flag==0`) the residual or DAE is defined. This C code is incrementally linked with Scicos to be used as a standard block. The computation function, beside an automatically generated interfacing function, defines the new explicit block to be replaced by implicit blocks. Here is the generated code for the model in Fig. 6.3.1.

```
/*
number of discrete variables = 0
number of variables = 6
number of inputs = 0
number of outputs = 2
number of modes = 0
number of zero-crossings = 0
```

```
I/O direct dependency = false
*/

#include <math.h>
#include <scicos/scicos_block.h>


/* Utility functions */

double ipow_(double x, int n)
{
        double y;
        y = n % 2 ? x : 1;
        while (n >>= 1) {
                x = x * x;
                if (n % 2) y = y * x;
        }
        return y;
}

double ipow(double x, int n)
{
        /* NaNs propagation */
        if (isnan(x) || x == 0.0 && n == 0) return exp(x * log((double)n));
        /* Normal execution */
        if (n < 0) return 1.0 / ipow_(x, -n);
        return ipow_(x, n);
}


/* Scicos block's entry point */

void imppart_opamp2(scicos_block *block, int flag)
{
        double *rpar = block->rpar;
        double *z = block->z;
        double *x = block->x;
        double *xd = block->xd;
        double **y = block->outptr;
        double **u = block->inptr;
        double *g = block->g;
        double *res = block->res;
        int *jroot = block->jroot;
        int *mode = block->mode;
        int nevprt = block->nevprt;
        int property[6];

        /* Intermediate variables */
        double v0, v1;

        if (flag == 0) {
                res[0] = xd[0]*rpar[0]+x[2]+x[4];
                v0 = -x[1]*rpar[9];
                v1 = Get_Jacobian_parameter();
                res[1] = (rpar[10]+v0-rpar[5]*x[1])*v1;
                res[2] = (sin(6.28318530718*get_scicos_time()*rpar[3])
                        *rpar[2]+x[3]+x[1]*rpar[9]-rpar[6]*x[2]-rpar[4]*x[2])*v1;
                res[3] = x[2]+xd[3]*rpar[7];
                res[4] = (x[4]*rpar[11]-x[0])*v1;
                res[5] = (x[0]+v0-rpar[12]*x[5])*v1;
        } else if (flag == 1) {
                if (get_phase_simulation() == 1) {
                        v0 = -x[1]*rpar[9];
                        y[0][0] = v0+rpar[4]*x[2]-x[3]; /* main.B15.vo */
                        y[1][0] = x[0]+v0; /* main.B16.vo */
                } else {
                        v0 = -x[1]*rpar[9];
                        y[0][0] = v0+rpar[4]*x[2]-x[3]; /* main.B15.vo */
                        y[1][0] = x[0]+v0; /* main.B16.vo */
                }
        } else if (flag == 2 && nevprt < 0) {
        } else if (flag == 4) {
                x[0] = rpar[1]; /* main.B1.v */
                x[1] = 0.0; /* main.B7.n.i */
                x[2] = 0.0; /* main.B9.n.i */
                x[3] = rpar[8]; /* main.B10.v */
                x[4] = 0.0; /* main.B13.p.i */
                x[5] = 0.0; /* main.B14.n.i */
        } else if (flag == 6) {
        } else if (flag == 7) {
                property[0] = 1; /* main.B1.v (state variable) */
                property[1] = -1; /* main.B7.n.i (algebraic variable) */
                property[2] = -1; /* main.B9.n.i (algebraic variable) */
                property[3] = 1; /* main.B10.v (state variable) */
                property[4] = -1; /* main.B13.p.i (algebraic variable) */
                property[5] = -1; /* main.B14.n.i (algebraic variable) */
                set_pointer_xproperty(property);
        } else if (flag == 9) {
        }
        return;
}
```

6.6 Jacobian Matrix Evaluation

The introduction of implicit blocks has had two consequences for the numerical solver:

• The global system to be simulated is a DAE (with explicit blocks the resulting global system was an ODE),

• The implicit part of the dynamics, defined in Modelica, contains symbolic expressions which can be used to obtain an analytical expressions for the Jacobian.

Mostly, system models with spatially lumped elements appear as implicit nonlinear DAEs. Only in special cases their formulation as explicit state equations is possible [?]. The simulation of such systems requires specific numerical software such as DASSL. Jacobian evaluations are needed by these solvers. They can either be done numerically or symbolically. The analytical expression for the Jacobian obtained through symbolic computation is more accurate and leads to better simulation performance. So if the system description is available symbolically, Jacobian evaluation by symbolic manipulation should be performed, if possible. There are situations, however, where symbolic information is only partially available. This happens in particular when the system is defined in a modular way, for example by a block diagram, and only the dynamics of some of the blocks are available symbolically, the others are defined by computer programs. Consider the Scicos diagram illustrated in Fig. 6.6.1. This model contains both explicit and implicit blocks.

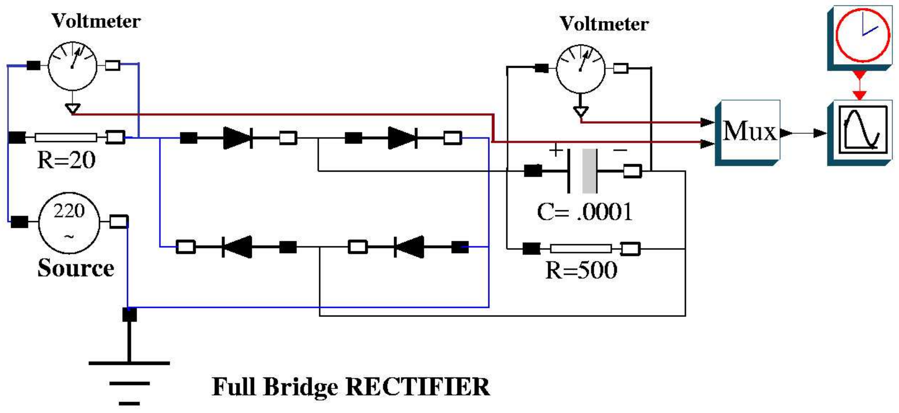


Figure 6.6.1: A system consisting several implicit blocks (components).

In this case all of the continuous-time dynamics is in the implicit parts so that when Scicos generates the Modelica program for this part, it generates a C code from Modelica program and replaces this part with an explicit block whose internal dynamics is described by this program, the resulting explicit Scicos model is given in Fig. 6.6.2.

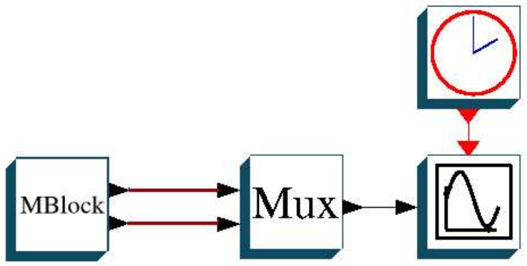


Figure 6.6.2: Replacing Implicit blocks of Fig. 6.6.1 with an explicit block.

In this case the continuous-time dynamics is in the block `MBlock` and the DAE equations are symbolically available. This means that the expression of the Jacobian can be computed analytically and made available to the numerical solver. Unfortunately, this is not always the case as it can be seen in the example illustrated in Fig. 6.6.3.

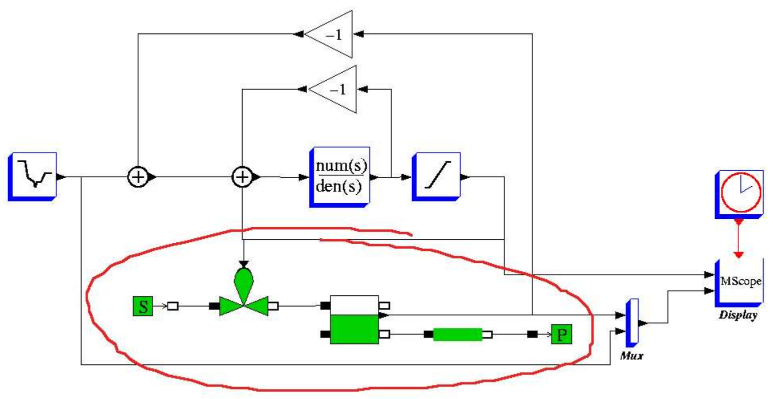


Figure 6.6.3: Selection of implicit blocks in a mixed-model Scicos diagram.

In this case, after the substitution of the implicit part by an explicit block we obtain the diagram in Fig. 6.6.4. Note that in this case the explicit part contains also continuous-time dynamics (in particular, a linear system defined in terms of its transfer function). So symbolic information is not available for the computation of the Jacobian, or more specifically only partial information is available.

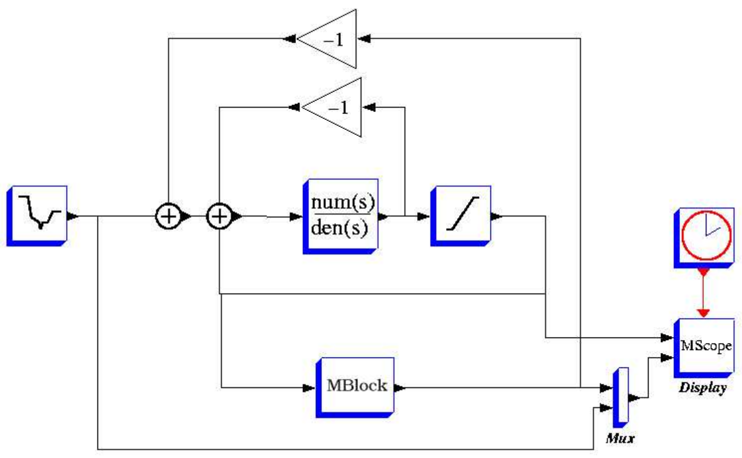


Figure 6.6.4: Abstraction of implicit blocks into one explicit block.

Replacing all the explicit parts also with a single explicit block, we obtain the diagram in Fig. 6.6.5 or more simply the me in Fig. 6.6.6. In general, this is the type of system that the numerical solver has to deal with [?, ?].

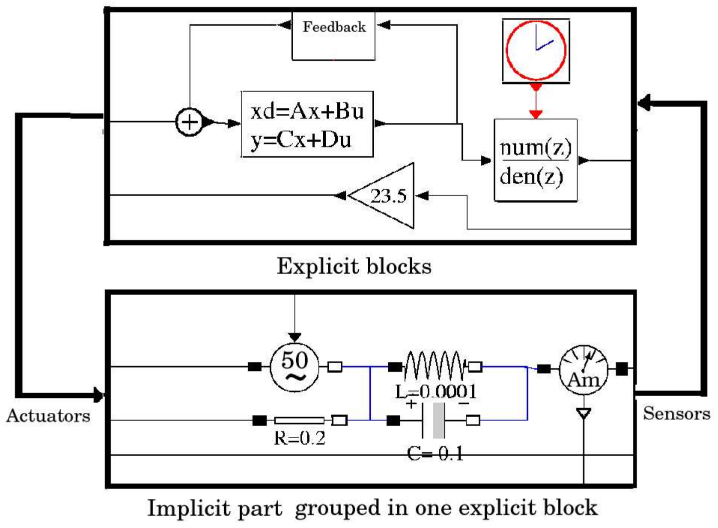


Figure 6.6.5: Abstraction of model in two explicit blocks.

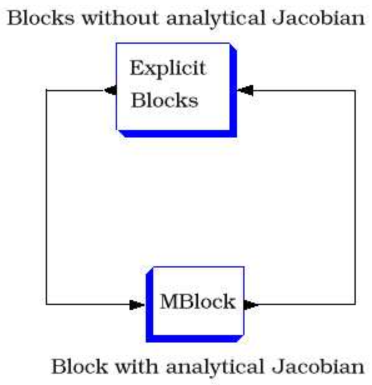

Blocks without analytical Jacobian
Explicit Blocks
MBlock
Block with analytical Jacobian

Figure 6.6.6: Abstraction of model in two explicit blocks.

In Scicos mixed-models, we have only the analytical expression of the Jacobian matrix of the implicit part of the model, and the Jacobian of the rest of the model should be computed numerically. In this section we will explain how this partial information can be exploited to improve the accuracy of the global Jacobian.

6.6.1 Jacobian Evaluation of Mixed-Model Systems

Consider the mixed-model DAE system illustrated in Fig 6.6.7. This system can be represented as follows:

$$0 = \phi(\dot{x}, x), \tag{6.6.1}$$

where x, the state vector, is given by

(6.6.2)
$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix},$$

we assume the analytical expressions of F and G are available and thus can be differentiated symbolically. On the other hand H and L can only be evaluated numerically.

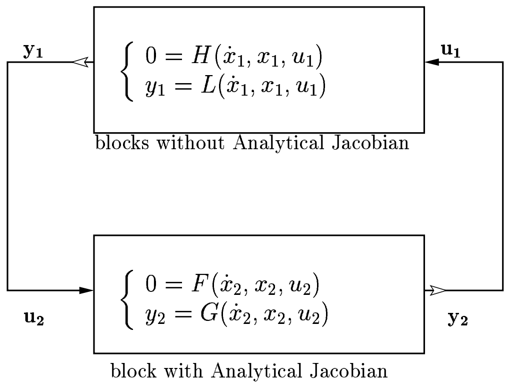

y_1
$\begin{cases} 0 = H(\dot{x}_1, x_1, u_1) \\ y_1 = L(\dot{x}_1, x_1, u_1) \end{cases}$
u_1
blocks without Analytical Jacobian
$\begin{cases} 0 = F(\dot{x}_2, x_2, u_2) \\ y_2 = G(\dot{x}_2, x_2, u_2) \end{cases}$
u_2
y_2
block with Analytical Jacobian

Figure 6.6.7: Scicos model from simulator's view.

Solving this system numerically requires the evaluation of the Jacobian

$$J = \alpha \frac{\partial \phi}{\partial \dot{x}} + \frac{\partial \phi}{\partial x},$$

for any parameter α.

Theorem:

Consider the DAE system illustrated in Fig 6.6.7 and assume that it contains no algebraic loop, i.e.,

(6.6.3)
$$\frac{\partial M}{\partial u_1} = 0,$$

(6.6.4)
$$\frac{\partial N}{\partial u_2} = 0,$$

where

$$M = G(\dot{x}_2, x_2, L(\dot{x}_1, x_1, u_1)),$$
$$N = L(\dot{x}_1, x_1, G(\dot{x}_2, x_2, u_2)).$$

Let

$$J = \alpha \frac{\partial \phi}{\partial \dot{x}} + \frac{\partial \phi}{\partial x}.$$

Then

$$J = \begin{pmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{pmatrix},$$

where

$$(6.6.5) \quad \begin{cases} J_{11} = \frac{\partial H}{\partial x_1} + \alpha\frac{\partial H}{\partial \dot{x}_1} + \frac{\partial H}{\partial u_1}\frac{\partial G}{\partial u_2}(\frac{\partial L}{\partial x_1} + \alpha\frac{\partial L}{\partial \dot{x}_1}) \\ J_{12} = \frac{\partial H}{\partial u_1}(\frac{\partial G}{\partial x_2} + \alpha\frac{\partial G}{\partial \dot{x}_2}) \\ J_{21} = \frac{\partial F}{\partial u_2}(\frac{\partial L}{\partial x_1} + \alpha\frac{\partial L}{\partial \dot{x}_1}) \\ J_{22} = \frac{\partial F}{\partial x_2} + \alpha\frac{\partial F}{\partial \dot{x}_2} + \frac{\partial F}{\partial u_2}\frac{\partial L}{\partial u_1}(\frac{\partial G}{\partial x_2} + \alpha\frac{\partial G}{\partial \dot{x}_2}). \end{cases}$$

Proof:

Note that the system equations are the followings

$$\begin{aligned} 0 &= H(\dot{x}_1, x_1, u_1) \\ 0 &= F(\dot{x}_2, x_2, u_2) \\ y_1 &= L(\dot{x}_1, x_1, u_1) \\ y_2 &= G(\dot{x}_2, x_2, u_2) \\ u_1 &= y_2 \\ u_2 &= y_1, \end{aligned}$$

from which we obtain the following expressions

$$(6.6.6) \quad J_{11} = \frac{\partial H}{\partial x_1} + \alpha\frac{\partial H}{\partial \dot{x}_1} + \frac{\partial H}{\partial u_1}\frac{\partial u_1}{\partial x_1}$$

$$(6.6.7) \quad J_{12} = \frac{\partial H}{\partial u_1}\frac{\partial u_1}{\partial x_2}$$

$$(6.6.8) \quad J_{21} = \frac{\partial F}{\partial u_2}\frac{\partial u_2}{\partial x_1}$$

$$(6.6.9) \quad J_{22} = \frac{\partial F}{\partial x_2} + \alpha\frac{\partial F}{\partial \dot{x}_2} + \frac{\partial F}{\partial u_2}\frac{\partial u_2}{\partial x_2}.$$

In (6.6.6), the term $\frac{\partial u_1}{\partial x_1}$ can be rewritten as

$$\begin{aligned} \frac{\partial u_1}{\partial x_1} &= \frac{\partial y_2}{\partial x_1} \\ &= \frac{\partial y_2}{\partial u_2}\frac{\partial u_2}{\partial x_1} \\ &= \frac{\partial G}{\partial u_2}(\frac{\partial L}{\partial x_1} + \alpha\frac{\partial L}{\partial \dot{x}_1} + \frac{\partial L}{\partial u_1}\frac{\partial u_1}{\partial x_1}). \end{aligned}$$

From (6.6.3) we have $\frac{\partial G}{\partial u_2}\frac{\partial L}{\partial u_1} = 0$, thus we obtain

(6.6.10) $$\frac{\partial u_1}{\partial x_1} = \frac{\partial G}{\partial u_2}(\frac{\partial L}{\partial x_1} + \alpha\frac{\partial L}{\partial \dot{x}_1}).$$

Hence from (6.6.6) and (6.6.10) we obtain the J_{11} expression in (6.6.5).

For J_{12}, (6.6.7) can be rewritten as

$$\begin{aligned} J_{12} &= \frac{\partial H}{\partial u_1}\frac{\partial u_1}{\partial x_2} \\ &= \frac{\partial H}{\partial u_1}\frac{\partial y_2}{\partial x_2} \\ &= \frac{\partial H}{\partial u_1}(\frac{\partial G}{\partial x_2} + \alpha\frac{\partial G}{\partial \dot{x}_2} + \frac{\partial G}{\partial u_2}\frac{\partial u_2}{\partial x_2}) \\ &= \frac{\partial H}{\partial u_1}(\frac{\partial G}{\partial x_2} + \alpha\frac{\partial G}{\partial \dot{x}_2} + \frac{\partial G}{\partial u_2}\frac{\partial L}{\partial u_1}\frac{\partial u_1}{\partial x_2}). \end{aligned}$$

Since $\frac{\partial G}{\partial u_2}\frac{\partial L}{\partial u_1} = 0$, the desired expression for J_{12} in (6.6.5) can be obtained.

To obtain J_{21}, (6.6.8) can be rewritten

$$\begin{aligned} J_{21} &= \frac{\partial F}{\partial u_2}\frac{\partial u_2}{\partial x_1} \\ &= \frac{\partial F}{\partial u_2}\frac{\partial y_1}{\partial x_1} \\ &= \frac{\partial F}{\partial u_2}(\frac{\partial L}{\partial x_1} + \alpha\frac{\partial L}{\partial \dot{x}_1} + \frac{\partial L}{\partial u_1}\frac{\partial u_1}{\partial x_1}) \\ &= \frac{\partial F}{\partial u_2}(\frac{\partial L}{\partial x_1} + \alpha\frac{\partial L}{\partial \dot{x}_1} + \frac{\partial L}{\partial u_1}\frac{\partial G}{\partial u_2}\frac{\partial u_2}{\partial x_2}). \end{aligned}$$

From (6.6.4) we have $\frac{\partial L}{\partial u_1}\frac{\partial G}{\partial u_2} = 0$, thus we obtain the desired expression for J_{21} in (6.6.5).

To obtain J_{22}, the term $\frac{\partial u_2}{\partial x_2}$ in (6.6.9) can be rewritten as

$$\begin{aligned} \frac{\partial u_2}{\partial x_2} &= \frac{\partial y_1}{\partial x_2} \\ &= \frac{\partial y_1}{\partial u_1}\frac{\partial u_1}{\partial x_2} \\ &= \frac{\partial L}{\partial u_1}(\frac{\partial G}{\partial x_2} + \alpha\frac{\partial G}{\partial \dot{x}_2} + \frac{\partial G}{\partial u_2}\frac{\partial u_2}{\partial x_2}). \end{aligned}$$

Since $\frac{\partial L}{\partial u_1}\frac{\partial G}{\partial u_2} = 0$ we have

(6.6.11) $$\frac{\partial u_2}{\partial x_2} = \frac{\partial L}{\partial u_1}\left(\frac{\partial G}{\partial x_2} + \alpha\frac{\partial G}{\partial \dot{x}_2}\right).$$

Combining (6.6.9) and (6.6.11) we get J_{22} expression in (6.6.5). □

6.6.2 Scicos Implementation and Simulation

The numerical solver in Scicos needs the Jacobian obtained in (6.6.5). In (6.6.5), the following can be computed analytically because their expressions are symbolically available:

(6.6.12) $$\frac{\partial F}{\partial x_2}, \quad \frac{\partial F}{\partial \dot{x}_2}, \quad \frac{\partial F}{\partial u_2}, \quad \frac{\partial G}{\partial x_2}, \quad \frac{\partial G}{\partial \dot{x}_2}, \quad \frac{\partial G}{\partial u_2}.$$

These expressions are indeed computed by the Modelica parser and compiler integrated in Scicos. On the other hand, numerical differentiation is used to obtain the following expressions.

(6.6.13) $$\frac{\partial H}{\partial x_1} + \alpha\frac{\partial H}{\partial \dot{x}_1}, \quad \frac{\partial L}{\partial x_1} + \alpha\frac{\partial L}{\partial \dot{x}_1}, \quad \frac{\partial H}{\partial u_1}, \quad \frac{\partial L}{\partial u_1}.$$

To compute $\frac{\partial H}{\partial x_1} + \alpha\frac{\partial H}{\partial \dot{x}_1}$ and $\frac{\partial L}{\partial x_1} + \alpha\frac{\partial L}{\partial \dot{x}_1}$ numerically, x_1 and $\dot{x}_1$ are perturbed in one of their elements (keeping the others constant) and H and L are evaluated and compared with their previous values (when x_1 and $\dot{x}_1$ are not perturbed). Thus the approximate value of the ij^{th} element of $\frac{\partial H}{\partial x_1} + \alpha\frac{\partial H}{\partial \dot{x}_1}$ and $\frac{\partial L}{\partial x_1} + \alpha\frac{\partial L}{\partial \dot{x}_1}$ are

$$\begin{aligned} &\approx \frac{H_i(\dot{x}_1 + \alpha\sigma_j, x_1 + \sigma_j, u_1) - H_i(\dot{x}_1, x_1, u_1)}{\sigma_j}, \\ &\approx \frac{L_i(\dot{x}_1 + \alpha\sigma_j, x_1 + \sigma_j, u_1) - L_i(\dot{x}_1, x_1, u_1)}{\sigma_j}. \end{aligned}$$

To compute the remaining terms of 6.6.13, the inputs of explicit blocks (outputs of generated explicit block) are disconnected and perturbed and H and L are evaluated and compared with their previous values (when u_1 is not perturbed). Thus the approximate value of the ij^{th} elements of $\frac{\partial H}{\partial u_1}$ and $\frac{\partial L}{\partial u_1}$ are

$$\begin{aligned} \frac{\partial H_i}{\partial u_{1j}} &\approx \frac{H_i(\dot{x}_1, x_1, u_1 + \sigma_j) - H_i(\dot{x}_1, x_1, u_1)}{\sigma_j}, \\ \frac{\partial L_i}{\partial u_{1j}} &\approx \frac{L_i(\dot{x}_1, x_1, u_1 + \sigma_j) - L_i(\dot{x}_1, x_1, u_1)}{\sigma_j}. \end{aligned}$$

The size of the perturbation is obviously important for the accuracy of the approximation. The choice of perturbation is a trade off between making the perturbation as small as possible to improve the theoretical accuracy, while at the same time not making it so small that the accuracy of the computer becomes limiting. A σ not smaller than the square root of the inaccuracy in x minimizes the total error (truncation error plus condition error) in the numerical Jacobian, that is,

$$\sigma_j = \sqrt{\epsilon}\, x_j,$$

where ϵ is the smallest positive value such that

$$1.0 + \epsilon = 1.0.$$

In DASKR, the following code is used to determine the size of perturbation.

```
DEL=SQUR*MAX(ABS(Y(I)),ABS(H*YPRIME(I)),ABS(1.D0/EWT(I)))
```

In Scicos, since in most cases nonlinearities and stiffness are found in the implicit part, the use of exact expressions improves a great deal the precision of the evaluated Jacobians furnished to the solver. This results in more accurate simulation and can even avoid simulation failures in some cases.

An Example

Introducing analytical Jacobian reduces the error and in many stiff cases it helps the solver to integrate the models that it was not able to integrate with a numerical Jacobian. The example shown in Fig. 6.6.1 is simulated in Scicos for two cases. First we use the numerical Jacobian calculated by DASKR. In this case, the solver stops integration at 0.0012 with this error message

```
IDID = -7  -- The nonlinear system solver in the time integration could not converge.
```

For the same example but using the analytical Jacobian, the integration finishes successfully at 0.1, as shown in Fig. 6.6.8.

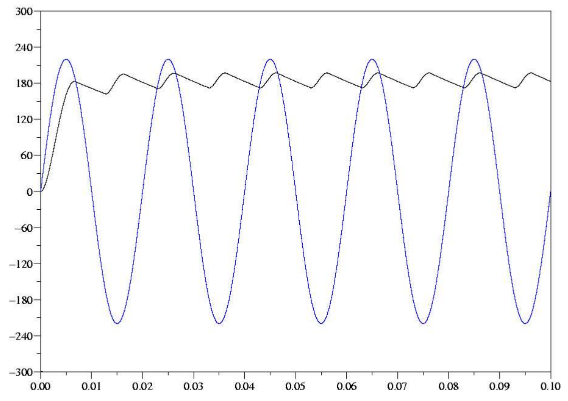


Figure 6.6.8: Simulation result of model of Fig. 6.6.1 with analytical Jacobian.

6.7 Condition Number Improvement

In section 2, we saw that for a semi-explicit DAE

$$\begin{aligned} \dot{x} &= f(x,y,t) \\ 0 &= g(x,y,t), \end{aligned}$$

if we scale the DAE by α, i.e.,

$$\begin{aligned} \dot{x} &= f(x,y,t) \\ 0 &= \alpha g(x,y,t), \end{aligned}$$

where α is a coefficient provided by the solver, then the condition number of the Jacobian matrix can be reduced. A reduced condition number results in a less round-off error and a higher accuracy. In order to implement this feature, we need to have a DAE in a semi-explicit form and know the algebraic and differential equations of the DAE. Modelicac code generator was modified to to generate a DAE of the form,

$$\begin{aligned} M\dot{x} &= f(x,y,t) \\ 0 &= \alpha\, g(x,y,t), \end{aligned}$$

where M is a linear non-singular matrix.

6.8 Numerical Difficulties, an Open Problem

Modelica builds a DAE set from the Modelica model provided by Scicos. This DAE is delivered to the numerical integrator, which is DASKR. The obtained DAE may be a large system of equations. For example, the electrical circuit model in Fig. 6.6.1 may generate about 20 states. This may be more than what is necessary to describe the system.

The person using a simulation package has some leeway in how they describe the problem. The developers of the simulation package also have some leeway in how they choose to take the user's description and formulate the DAE that is to be integrated. Selecting and eliminating auxiliary states may be important as the size of the system grows. However, this must be done carefully. Reducing system size may reduce sparsity of the coefficients and actually slow down the simulation. Selecting different subsets of states to eliminate may result in different sets of equations. Although from a mathematical viewpoint all these equations are equivalent, the result of simulating the models may be quite different. In the following we examine some difficulties raised in simulation concerning alternative model formulations. The main difference when using implicit blocks, as far as the simulation is concerned, is that the resulting global system is very often given as a set of DAEs. In extensive testing during the development of Scicos, we have found that for some surprisingly simple problems, the formulation can make a big difference in ways that cannot always be predicted.

An Example

For instance, consider the bridge rectifier of Fig.6.6.1. The final DAE provided by the Scicos parser that would be integrated by DASKR when simulation is requested of Scicos is:

$$
\begin{aligned}
r_0 &= 220\\
r_1 &= 50\\
r_2 &= 0.0001\\
r_3 &= 500\\
r_4 &= 20
\end{aligned}
$$

$$
\begin{aligned}
v_0 &= r_0 \sin(\pi r_1 t)\\
v_1 &= r_3(x_3 - f(-x_2) - f(x_1 - x_2))\\
v_2 &= r_4(f(x_2 + v_1) - f(-x_2))\\
v_3 &= x_2 + v_0 + v_1 + v_2\\
0 &= x_1 + v_0 + v_2\\
0 &= x_4 + v_1\\
0 &= f(x_1 - x_2) + f(-x_2) - f(x_2 + v_1) - f(v_3)\\
0 &= r_2\dot{x}_4 - x_3,
\end{aligned}
\tag{6.8.1}
$$

where the function $f(x)$, which is the characteristic curve of an electrical diode, is defined as:

$$
\begin{aligned}
R &= 10^8\\
I_d &= 10^{-6}\\
V_t &= 0.04\\
N &= 15\\
f(x) &= \begin{cases} \frac{x}{R} - I_d + I_d e^{\frac{x}{V_t}} & \text{if } x < NV_t\\ \frac{x}{R} - I_d + I_d(\frac{V_t - NV_t + x}{V_t})e^N & \text{if } x >= NV_t. \end{cases}
\end{aligned}
$$

The graph of $f(x)$ is shown in Fig.6.8.1. For $x > 0.6$ the graph is linear.

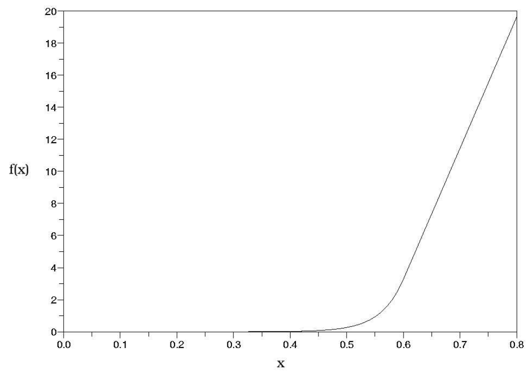


Figure 6.8.1: $f(x)$ (Diode I-V characteristic).

This example is a DAE composed of 4 equations. The 4 states are given by the x_1, x_2, x_3, and x_4, of which x_4 is a differential variable and x_1, x_2, x_3 are algebraic variables. The

Scicos parser has greatly reduced the problem size from more than 20 states to 4. This model of a bridge rectifier is a stiff system at the points where conducting diodes turn off and the two others turn on. To simulate this model, it was fed to DASKR with the initial conditions

$$\begin{aligned} x(0) &= [0,0,0,0] \\ \dot{x}(0) &= [0,0,0,0], \end{aligned}$$

but DASKR cannot integrate the DAE and stops at the first stiff point (i.e., $t = 6.12\text{E-}3$) and raises the following error message

```
Error Flag: -7
DASKR--  AT T (=R1) AND STEP SIZE H (=R2) THE  NONLINEAR SOLVER FAILED TO CONVERGE
         REPEATEDLY OR WITH ABS(H)=HMIN
         In above, R1 =  0.6127527006389E-02  R2 =  0.2065780285931E-12
```

However, by replacing v_1 by $-x_4$ (see the second DAE equation in (6.8.1)) and substituting it in the rest of equations, the system becomes

$$\begin{aligned} v_0 &= r_0 \sin(\pi r_1 t) \\ v_1 &= r_3(x_3 - f(-x_2) - f(x_1 - x_2)) \\ v_2 &= r_4(f(x_2 - x_4) - f(-x_2)) \\ v_3 &= x_2 + v_0 - x_4 + v_2 \\ 0 &= x_1 + v_0 + v_2 \\ 0 &= x_4 + v_1 \\ 0 &= f(x_1 - x_2) + f(-x_2) - f(x_2 - x_4) - f(v_3) \\ 0 &= r_2 \dot{x}_4 - x_3. \end{aligned} \tag{6.8.2}$$

This second model (6.8.2) for the model of Fig. 6.6.1 can be successfully integrated by DASKR. Note that this reformulation does not change the index of the DAE nor the fact that it is semi-explicit index one.

In order to examine this example more carefully we computed the Jacobian used by DASKR analytically. With the analytical Jacobian and a smaller time intervals (for (6.8.1)), DASKR can integrate both of the DAEs (6.8.1) and (6.8.2). However, for the modified one it uses a larger step size specially at stiff points which results in a faster simulation. For instance, at t =6.16E-3 the step sizes are

- For (6.8.1), H=0.46609E-07
- For (6.8.2), H=0.13031E-05

The number of calls to Residual and Jacobian functions for integrating at t=[0, 0.1] are

- For (6.8.1): $\text{N}_{fun} = 165345$, and $\text{N}_{jac} = 81140$
- For (6.8.2): $\text{N}_{fun} = 18498$, and $\text{N}_{jac} = 1879$

These results show a remarkable difference between integration of the two DAEs. The reason may lay in the condition number of the Jacobian of iteration matrix of the DAEs. The condition numbers are

- For (6.8.1), CN=14172
- For (6.8.2), CN=1140

Another interesting point is the sensitivity of the condition number in term of state variables. The first DAE is mush more sensitive than the second one.

$$\begin{aligned} \|\frac{\Delta CN_1}{\Delta x}\| &= 6.160E+09 \\ \|\frac{\Delta CN_2}{\Delta x}\| &= 13706. \end{aligned}$$

6.8.1 Location of Nonlinearities

To more easily discuss the roles of nonlinearities, we consider some small, but illustrative, types of DAE models. These examples show how a simple substitution can change the difficulty of a problem. Here f is a general nonlinear function. The first example is:

$$\begin{aligned} 0 &= x_1 - x_2 \\ 0 &= \dot{x}_1 + f(x_2). \end{aligned}$$

With is a highly nonlinear function $f(x)$, it may be difficult to find consistent initial conditions if $x_1(t_0), \dot{x}_1(t_0)$ are given and we need to find $x_2(t_0)$. However, upon replacing x_2 with x_1 in the second equation, the DAE becomes

$$\begin{aligned} 0 &= x_1 - x_2 \\ 0 &= \dot{x}_1 + f(x_1), \end{aligned}$$

which is no longer hard to find initial conditions for.

For example, for the DAE

$$\begin{aligned} 0 &= x_2 - x_3 \\ 0 &= x_1 + g(g(x_3) - g(-x_3))) \\ 0 &= x_3 - \dot{x}_2, \end{aligned}$$

where

$$g(z) = z\ \sin(z)\ \exp(-z),$$

and with initial condition

$$\begin{aligned} x(0) &= [2, 1, -1] \\ \dot{x}(0) &= [0, -2, 0], \end{aligned}$$

DASKR cannot find the consistent initial condition and raises this error message:

```
Error Flag: -12
DASKR--  AT T (=R1) AND STEPSIZE H (=R2) THE INITIAL (Y,YPRIME) COULD NOT BE COMPUTED
     In above,  R1 =  0.0000000000000E+00  R2 =  0.9999999974752E-13
```

On the other hand, if we substitute x_2 for x_3, to give

$$\begin{aligned} 0 &= x_2 - x_3 \\ 0 &= x_1 + g(g(x_2) - g(-x_2))) \\ 0 &= x_2 - \dot{x}_2, \end{aligned}$$

DASKR will succeed in finding the consistent initial condition and then integrating the DAE. Here, the problem lays in finding the consistent initial conditions. In the first DAE, given x_2, DASKR should find x_3 then x_1 which is not trivial, while in the second, it's just evaluating the $g(x_2)$ and assigning it to x_1.

For nonlinear problems like this one, depicted in Fig. 6.8.1, at $x < 0.3$, $f(x)$ is nearly flat which makes it difficult for the solver to find the initial condition while for $x > 0.6$, it is easier for the solver.

As another example, consider the index one DAE

$$0 = \dot{x}_2 + x_2 \tag{6.8.3}$$
$$0 = \dot{x}_2 + f(x_1 + \dot{x}_2) \tag{6.8.4}$$
$$0 = \dot{x}_2 - x_3, \tag{6.8.5}$$

where

$$f(z) = z^3.$$

With this consistent initial condition

$$\begin{aligned} x(0) &= [2, 1, -1] \\ \dot{x}(0) &= [0, -1, 0], \end{aligned}$$

DASKR will have problem and stops the integration with this error message

```
Error Flag: -7
DASKR--  AT T (=R1) AND STEPSIZE H (=R2) THE  NONLINEAR SOLVER FAILED TO CONVERGE
         REPEATEDLY OR WITH ABS(H)=HMIN
         In above,  R1 =  0.6999999823165E-06  R2 =  0.7629394338515E-12
```

To improve the situation, we note that from (6.8.5) it is deduced that $\dot{x}_2 = x_3$. As a result, we can replace $\dot{x}_2$ by x_3 in (6.8.3) and (6.8.4), and we will get

$$0 = x_3 + x_2 \tag{6.8.6}$$
$$0 = x_3 + f(x_1 + x_3) \tag{6.8.7}$$
$$0 = \dot{x}_2 - x_3. \tag{6.8.8}$$

This problem can be integrated easily by DASKR. Notice that in the above substitution the new system (6.8.6)–(6.8.8) only has nonlinearities in terms of the state variables. When solving a system such as $(0 = F(\dot{x}, x, u))$, the Jacobian used by DASKR takes the form of $\frac{\alpha}{h}F_{x'} + F_x$ where h is the step size. Since h is usually small, we have that error in the computing of $F_{x'}$ gets amplified much more than error in F_x. Thus making $(F(.))$ simpler in terms of $\dot{x}$ can increase the accuracy of Jacobian and their conditioning. In general, in the original implicit formulation,

$$\begin{aligned}\dot{u} &= f_1(u, v)\\ 0 &= f_2(\dot{u}, v, u),\end{aligned}$$

and the alternate (semi-explicit) form

$$\begin{aligned}\dot{u} &= f_1(u, v)\\ 0 &= f_2(f_1(u, v), v, u),\end{aligned}$$

the semi-explicit form is expected to do better numerically. The two are equivalent, at the level of the nonlinear systems being solved for the BDF method. But the Newton iterations are not the same, because the predicted and subsequent iterate values of $\dot{u}$ in the equation $f_2(\dot{u}, v, u) = 0$ are not the same as the predicted and subsequent values of the function $\dot{u} = f_1(u, v)$. Between the two, the f_1 value is expected to be more accurate, hence the semi-explicit form is expected to have better convergence properties. As the iteration matrix is the same for the integration and initial value calculation, we expect that the second DAE will provide a better result in consistent initial condition computing as well.

The inclusion of implicit blocks makes it easier to derive and formulate a wide variety of problems. These models naturally lead to differential algebraic equations. However, as the examples show, mathematically equivalent models can have different numerical properties.

Chapter 7

Applications

In this chapter, three models will be presented that have been selected to illustrate the potential application fields of hybrid systems simulation with Scicos. The purpose of these examples is also to demonstrate the flexibility of Scicos to simulate the models developped in Modelica language.

In Section 7.1, a multi body dynamic system is presented to illustrate multi-mode and discontinuity handling in the Modelica language. This system is from [?]. In Section 7.2, a heat transfer system is modeled with Modelica and then simulated in Scicos. The example shows the possibility of solving PDEs in Scicos. Finally, in the Section 7.3, the use of Modelica language in modeling a real physical system is illustrated. An intermidiate cooling system for auxiliary machines used in a power plant is modeled nad simulated in Scicos. This model is taken from EDF[1].

7.1 Example 1: Sticky balls

In chapter (2), we modeled a sticky mass system in a Scicos block using the finite state machine method. In this example, we model the same system using the Modelica language. Here is the model written in Modelica.

```
class Balls
 parameter Real c1=1 "mass of 1st ball";
 parameter Real c2=2 "mass of 2nd ball";
 parameter Real b1=1 "equilibrium distance for mass 1";
 parameter Real b2=2 "equilibrium distance for mass 2";
 parameter Real joint=1;
 parameter Real disjoint=-1;

 Real x1(start=0);
 Real v1(start=0);
 Real x2(start=3);
 Real v2(start=0);
 Real y1;
 Real y2;
 Real Adh(start=0);
 discrete Real z(start=1);
```

[1]Electricité De France

```
equation

der(x1)=v1;
der(x2)=v2;

der(v1)=if (z>0) then c1*(b1-x1)
                 else (c1*b1+c2*b2-(c1+c2)*x1)/2;

der(v2)=if (z>0) then c2*(b2-x2)
                 else (c1*b1+c2*b2-(c1+c2)*x2)/2;

der(Adh)=if (z>0) then 0
                  else -Adh;


when (x1-x2)>0 then
   reinit (v1,(v1+v2)/2);
   reinit (v2,(v1+v2)/2);
   reinit (Adh,10);
   z=joint;
end when;

when (Adh-3+x1)<0 then
   z=disjoint;
end when;

y1=x1;
y2=x2;

end Balls;
```

`z` is a discrete-time variable that is used to indicate the current status of the balls (`joint` or `disjoint`). When the balls collide, they stick. The collision is defined by `when (x1-x2)>0 then`. In the `when`-statement, the state variables should be reinitialized and `z` become `joint`. When the adherence force becomes weak, the balls disjoint and as the state variables are identical for two balls, no reinitialization is required, just the `z` should be `disjoint`. The dynamic of each ball has been defined with two ODEs; one for position and one for speed. The ODEs are discontinuous and they change as a function of `z`.

In order to use this Modelica code in Scicos, a Scicos interfacing function should be created and associated with the Modelica code. The following listing is the interfacing function of the block. Here the position of two masses are defined as the explicit outputs of the block. Then we can visualize them with a `Scope` block (see Fig. 7.1.1 and Fig. 7.1.2).

```
function [x,y,typ]=Balls1(job,arg1,arg2)
  x=[];y=[];typ=[]
  select job
   case 'plot' then
    standard_draw(arg1)
   case 'getinputs' then
    [x,y,typ]=standard_inputs(arg1)
   case 'getoutputs' then
    [x,y,typ]=standard_outputs(arg1)
   case 'getorigin' then
    [x,y]=standard_origin(arg1)
   case 'set' then
    x=arg1;
   case 'define' then
  model=scicos_model()
  model.in=[];
  model.out=[1;1];
  model.sim='Balls'
  model.blocktype='c'
  model.dstate=[1]
  model.dep_ut=[%t %f]
  mo=modelica()
  mo.model='Balls'
  mo.inputs='';
  mo.outputs=['y1';'y2']
  model.equations=mo
  exprs=[string([1])]
  gr_i=['txt=[''Balls''];';
        'xstringb(orig(1),orig(2),txt,sz(1),sz(2),''fill'')']

  x=standard_define([2 2],model,exprs,gr_i)
  x.graphics.in_implicit=[]
  x.graphics.out_implicit=['E';'E']
  end
endfunction
}
```

This block has been used in a Scicos diagram in Fig. 7.1.1, and the simulation result has been shown in Fig. 7.1.2.

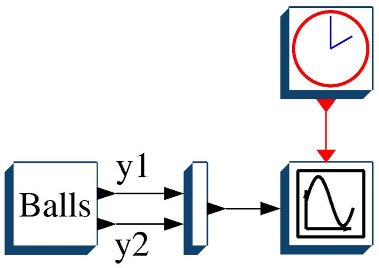


Figure 7.1.1: Sticky balls written in Modelica and modeled in Scicos.

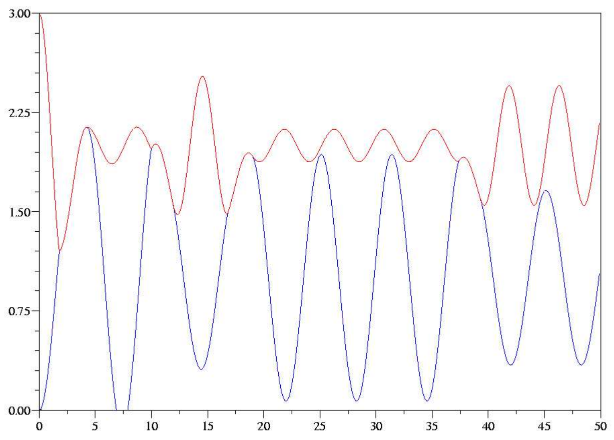


Figure 7.1.2: Simulation result of model of diagram of Fig. 7.1.1.

7.2 Example 2: Solving a PDE via Finite Difference Method

In mathematics, a partial differential equation (PDE) is an equation involving partial derivatives of an unknown function. In fact, a function is indirectly described by a relation between itself and its partial derivatives, rather than writing down a function explicitly. The relation should connect the function and its derivatives at the same point. A solution of the equation is any function satisfying this relation. Since a PDE usually has several solutions, a problem often includes additional boundary conditions which constrain the solution set. Partial differential equations are ubiquitous in science, as they describe phenomena such as fluid flow, gravitational fields, and heat propagation. In the special case of heat propagation in an isotropic and homogeneous medium in the 1-dimensional space, this equation is used,

$$u_t = ku_{xx},$$

where $u(t,x)$ is temperature as a function of time and position, u_t is the rate of change of temperature at a point over time, u_{xx} is the second spatial derivatives (thermal conductions) of temperature in the x directions, and k is a material-specific constant (thermal diffusivity). To solve a PDE, there are several methods. We use finite difference method. The approach taken by finite difference methods for partial differential equations is to approximate differential operators such as $u_x(x)$ and u_{xx} by two difference operators such as

$$u_x(x) \approx \frac{u(x) - u(x-h)}{h},$$

and

$$u_{xx}(x) \approx \frac{u(x+h) - 2u(x) + u(x-h)}{h^2},$$

for some small but finite h. Doing this substitution for a large enough number of points in the domain of definition (for instance $0, h, 2h, ..., 1$ in the case of the unit interval) gives a system of equations that can be solved by the numerical solvers.

For a case study, consider a metal rod with length L, heated from both ends. The temperature of every point along the rod at time $t = 0$, is given with,

$$u(x,0) = g(x) \qquad \text{for} \qquad 0 \le x \le L.$$

This function is known as the initial temperature distribution. Since heat can only enter or exit the rod at its boundaries we must define some "boundary conditions," for the rod. Therefore we need to define the conditions of the rod at the boundaries of the rod $(0, L)$,

$$u(0,t) = f_0(t), \quad u(L,t) = f_L(t) \qquad \text{for all} \qquad t > 0.$$

Combining the heat equation with the initial conditions and boundary conditions, we get

$$\begin{aligned}
u_t(x,t) &= ku_{xx}(x,t), && \text{for} \quad 0 \le x \le L \quad \text{and} \quad t > 0,\\
u(x,0) &= g(x), && \text{for} \quad 0 \le x \le L,\\
u(0,t) &= f_0(t), && \text{for} \quad t > 0,\\
u(L,t) &= f_L(t), && \text{for} \quad t > 0.
\end{aligned}$$

For a rod of length $L = 2$, and $k = 0.02$, setting the initial temperature distribution, $g(x) = 0$, and $f_0(t) = 25 + 20\sin(0.2\pi t)$ and $f_L(t) = 100$, then we can write a Modelica code to model this PDE system. The rod is divided into 20 sections and the difference equation is written for the each section. The dynamic model for each section is then

$$\frac{du}{dt} = k\frac{u(x+h) - 2u(x) + u(x-h)}{h^2}.$$

Here is the Modelica Code:

```
class fdm
    parameter Real k=0.02  "Thermal diffusivity";
    parameter Real L=2     "The rod's Length";
    Real u[21] "The rod element's temperature";
    PortH in1,in2;
    Real h;
    Real measure;
  equation
    measure = u[10];
    in1.T = u[1];
    in2.T = u[21];
    h=L/20;

    for i in 2:20 loop
       der(u[i])=k*(u[i+1]-2*u[i]+u[i-1])/h^2;
    end for;

end fdm;

class SourceCi
    PortH C ;
    Real  u "input";
  equation
```

```
    C.T=u;
end SourceCi;

class PortH
    Real T;
end PortH;
```

The temperature of the two ends of the rod are given by external signals in1 and in2. This is quit useful because the user can easily modify the f_0 and f_L in Scicos environment, without touching the Modelica code. The desired output is the temperature of the middle of the rod, i.e., measure=u[10]. The Scicos block diagram for this system has been depicted in the Fig. 7.2.1.

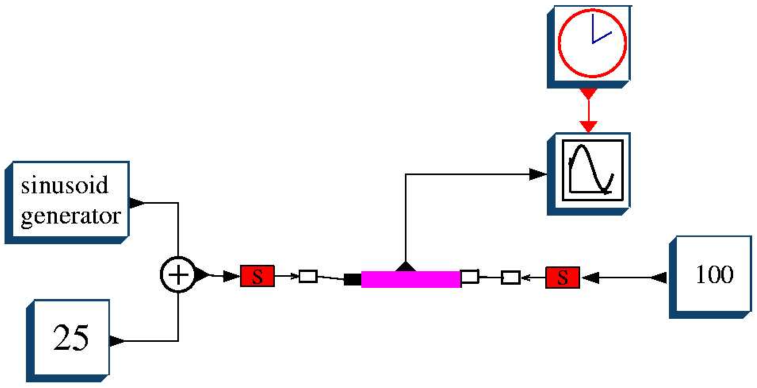


Figure 7.2.1: A PDE, written in the Modelica language, modeled in Scicos.

This is the generated C code for the model:

```
/*
number of discrete variables = 0
number of variables = 19
number of inputs = 2
number of outputs = 1
number of modes = 0
number of zero-crossings = 0
I/O direct dependency = false
*/

#include <math.h>
#include <scicos/scicos_block.h>


/* Utility functions */

double ipow_(double x, int n)
{
        double y;
        y = n % 2 ? x : 1;
        while (n >>= 1) {
                x = x * x;
                if (n % 2) y = y * x;
        }
        return y;
}

double ipow(double x, int n)
{
        /* NaNs propagation */
        if (isnan(x) || x == 0.0 && n == 0) return exp(x * log((double)n));
        /* Normal execution */
        if (n < 0) return 1.0 / ipow_(x, -n);
        return ipow_(x, n);
}


/* Scicos block's entry point */

void imppart_PDE(scicos_block *block, int flag)
{
        double *rpar = block->rpar;
        double *z = block->z;
        double *x = block->x;
        double *xd = block->xd;
        double **y = block->outptr;
        double **u = block->inptr;
        double *g = block->g;
        double *res = block->res;
        int *jroot = block->jroot;
        int *mode = block->mode;
        int nevprt = block->nevprt;
        int property[19];

        /* Intermediate variables */
        double v0, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12, v13, v14, v15;
        double v16, v17;

        if (flag == 0) {
                v0 = ipow(0.05*rpar[0], -2);
                v1 = -x[1];
                res[0] = xd[0]+rpar[1]*v0*(v1+2.0*x[0]-u[1][0]);
                v2 = -x[2];
                res[1] = xd[1]+rpar[1]*(v2+2.0*x[1]-x[0])*v0;
                v3 = -x[3];
                res[2] = xd[2]+rpar[1]*(v3+2.0*x[2]+v1)*v0;
                v4 = -x[4];
                res[3] = xd[3]+rpar[1]*(v4+2.0*x[3]+v2)*v0;
                v5 = -x[5];
                res[4] = xd[4]+rpar[1]*(v5+2.0*x[4]+v3)*v0;
                v6 = -x[6];
                res[5] = xd[5]+rpar[1]*(v6+2.0*x[5]+v4)*v0;
                v7 = -x[7];
                res[6] = xd[6]+rpar[1]*(v7+2.0*x[6]+v5)*v0;
                v8 = -x[18];
                res[7] = xd[7]+rpar[1]*(2.0*x[7]+v6+v8)*v0;
                v9 = -x[9];
                res[8] = xd[8]+rpar[1]*(v9+2.0*x[8]+v8)*v0;
                v10 = -x[10];
                v11 = -x[8];
                res[9] = xd[9]+rpar[1]*(v10+2.0*x[9]+v11)*v0;
                v12 = -x[11];
                res[10] = xd[10]+rpar[1]*(v12+2.0*x[10]+v9)*v0;
                v13 = -x[12];
                res[11] = xd[11]+rpar[1]*(v13+2.0*x[11]+v10)*v0;
                v14 = -x[13];
                res[12] = xd[12]+rpar[1]*(v14+2.0*x[12]+v12)*v0;
                v15 = -x[14];
                res[13] = xd[13]+rpar[1]*(v15+2.0*x[13]+v13)*v0;
                v16 = -x[15];
                res[14] = xd[14]+rpar[1]*(v16+2.0*x[14]+v14)*v0;
                v17 = -x[16];
                res[15] = xd[15]+rpar[1]*(v17+2.0*x[15]+v15)*v0;
                res[16] = xd[16]+rpar[1]*(2.0*x[16]+v16-x[17])*v0;
                res[17] = xd[17]+rpar[1]*v0*(2.0*x[17]+v17-u[0][0]);
                res[18] = xd[18]+rpar[1]*(v11+v7+2.0*x[18])*v0;
        } else if (flag == 1) {
                if (get_phase_simulation() == 1) {
                        y[0][0] = x[18]; /* main.B4.vo */
                } else {
                        y[0][0] = x[18]; /* main.B4.vo */
                }
        } else if (flag == 2 && nevprt < 0) {
        } else if (flag == 4) {
                x[0] = 0.0; /* main.B3.u[2] = heat */
                x[1] = 0.0; /* main.B3.u[3] = heat */
                x[2] = 0.0; /* main.B3.u[4] = heat */
                x[3] = 0.0; /* main.B3.u[5] = heat */
                x[4] = 0.0; /* main.B3.u[6] = heat */
                x[5] = 0.0; /* main.B3.u[7] = heat */
                x[6] = 0.0; /* main.B3.u[8] = heat */
                x[7] = 0.0; /* main.B3.u[9] = heat */
                x[8] = 0.0; /* main.B3.u[11] = heat */
                x[9] = 0.0; /* main.B3.u[12] = heat */
                x[10] = 0.0; /* main.B3.u[13] = heat */
                x[11] = 0.0; /* main.B3.u[14] = heat */
                x[12] = 0.0; /* main.B3.u[15] = heat */
                x[13] = 0.0; /* main.B3.u[16] = heat */
                x[14] = 0.0; /* main.B3.u[17] = heat */
                x[15] = 0.0; /* main.B3.u[18] = heat */
                x[16] = 0.0; /* main.B3.u[19] = heat */
                x[17] = 0.0; /* main.B3.u[20] = heat */
                x[18] = 0.0; /* main.B4.vo */
        } else if (flag == 6) {
        } else if (flag == 7) {
                property[0] = 1; /* main.B3.u[2] = heat (state variable) */
                property[1] = 1; /* main.B3.u[3] = heat (state variable) */
                property[2] = 1; /* main.B3.u[4] = heat (state variable) */
                property[3] = 1; /* main.B3.u[5] = heat (state variable) */
                property[4] = 1; /* main.B3.u[6] = heat (state variable) */
                property[5] = 1; /* main.B3.u[7] = heat (state variable) */
                property[6] = 1; /* main.B3.u[8] = heat (state variable) */
                property[7] = 1; /* main.B3.u[9] = heat (state variable) */
                property[8] = 1; /* main.B3.u[11] = heat (state variable) */
                property[9] = 1; /* main.B3.u[12] = heat (state variable) */
                property[10] = 1; /* main.B3.u[13] = heat (state variable) */
                property[11] = 1; /* main.B3.u[14] = heat (state variable) */
                property[12] = 1; /* main.B3.u[15] = heat (state variable) */
                property[13] = 1; /* main.B3.u[16] = heat (state variable) */
                property[14] = 1; /* main.B3.u[17] = heat (state variable) */
                property[15] = 1; /* main.B3.u[18] = heat (state variable) */
                property[16] = 1; /* main.B3.u[19] = heat (state variable) */
                property[17] = 1; /* main.B3.u[20] = heat (state variable) */
                property[18] = 1; /* main.B4.vo (state variable) */
                set_pointer_xproperty(property);
        } else if (flag == 9) {
        }

        return;
}
```

The simulation result, the temperature of the middle of the rod, has been shown in Fig. 7.2.2.

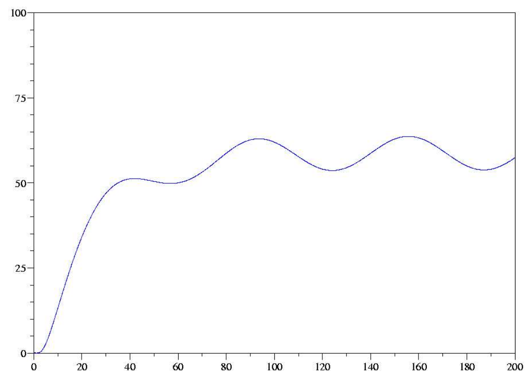


Figure 7.2.2: Simulation result of the PDE model of Fig. 7.2.1.

To obtain a more elaborated model, we can define the initial temperature as a function of distance from the both ends, i.e., instead of all zeros in the previous model, and also define the number of meshes as a variable. This code, however, is not still acceptable by Modelica compiler of Scicos, because an array with a parametric size and the `initial equation` keyword are not acceptable in Scicos yet.

```
class fdm
  parameter Real k=0.02  "Thermal diffusivity";
  parameter Real L=2     "Rod's Length";
  parameter Integer N=20     "Rod's Length";

  Real u[N+1] "heat";
  PortH in1,in2;
  Real h;
  Real measure;
initial equation

  for i in 1:N loop
      u[i]= 10+6*Modelica.Math.sin(((i-1)/N)
  end for;

equation
  measure = u[N/2];
  in1.T = u[1];
  in2.T = u[N+1];
  h=L/N;

  for i in 2:N loop
     der(u[i])=k*(u[i+1]-2*u[i]+u[i-1])/h^2;
  end for;

end fdm;
```

7.3 Example 3: A Secondary Cooling System for a Pressurized Water Reactor

In this example, we will show how a cooling system that is used in a power plant has been modeled with Modelica and simulated in Scicos. A pressurized water reactor (PWR) is a type of nuclear power reactor that uses ordinary light water for both coolant and for neutron moderator. In a PWR, the primary coolant loop is pressurized so water can be heated to 315 Celsius while still remaining liquid. In Fig. 7.3.1 a simplified diagram of PWR has been shown.

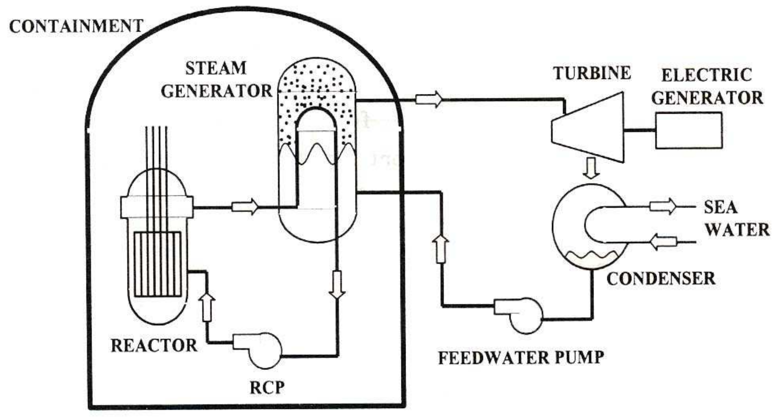


Figure 7.3.1: A Pressurized Water Reactor.

A PWR has three separate systems of pipes, or loops for moving heat. Water in these loops never mix together. However, heat energy from one-loop moves to another. Because of the heat produced by the fission reaction, water that is circulated through the core becomes extremely hot. As water is pressurized, it can be heated to 315 Celsius while still remaining liquid. The heat exchangers called steam generators are used to transmit heat to a secondary coolant which is allowed to boil and to produce steam for electricity generation. This is because water in the second loop is under less pressure. The second loop caries the steam to the turbine. A turbine is basically a pinwheel with many blades that are spun by steam. At power plants, turbines are attached to generators, which change the mechanical energy of the spinning turbine into electrical energy. After turning the turbine, the steam in the second loop has lost most of its heat energy. It is cooled and turned back into water so that it can be used again in the second loop. In the condenser, the second loop transfers some of its heat to the third loop. Again, heat is transferred from a heated substance to a cooler one, such as a river or sea water.

As an industrial application, we have taken the **SRI**[2] (intermediate cooling system for installment) models. The SRI system is an intermediate cooling circuit containing demineralized water that is used for the auxiliary machines in the conventional parts. For these parts ordinary water cannot be used. This system is composed of two parts; the physical systems and the control mechanisms. The simplified SRI[3] system includes, a cooling system, a pumping station, a heat source, and a control mechanism, see Fig. 7.3.2.

[2]Système Refroidissement Intermédiaire

[3]In the complete SRI model, there are some components that have been developped using some Modelica programing structures that are not acceptable by the version of Modelicac that we used. That is why we cut off the auxiliary parts of the model. It should be noted that these simplifications do not change the overall operation of the system

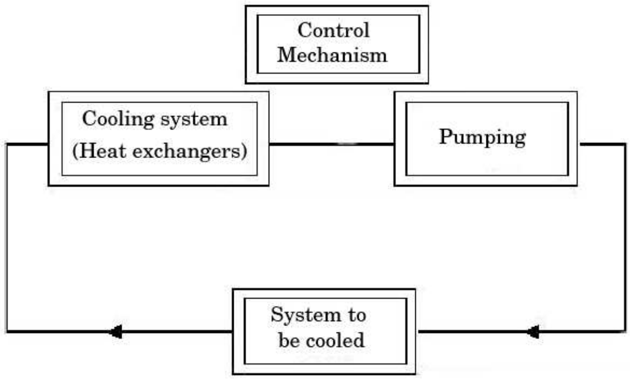


Figure 7.3.2: The simplified SRI system

A model for the simplified SRI system is depicted in Fig. 7.3.3. This system has been designed to provide a 17 Celsius water at the output of the heat exchanger with an almost constant flow rate. To achieve this objective, a heat exchanger parallel with a bypass branch are used in the main circuit. Each branch has an adjustable hydraulic valve that is controlled by the control mechanism. For example, when the temperature of the coolant increases, the control mechanism lets more water passes through the heat exchanger and reduce the flow rate in the bypass branch.

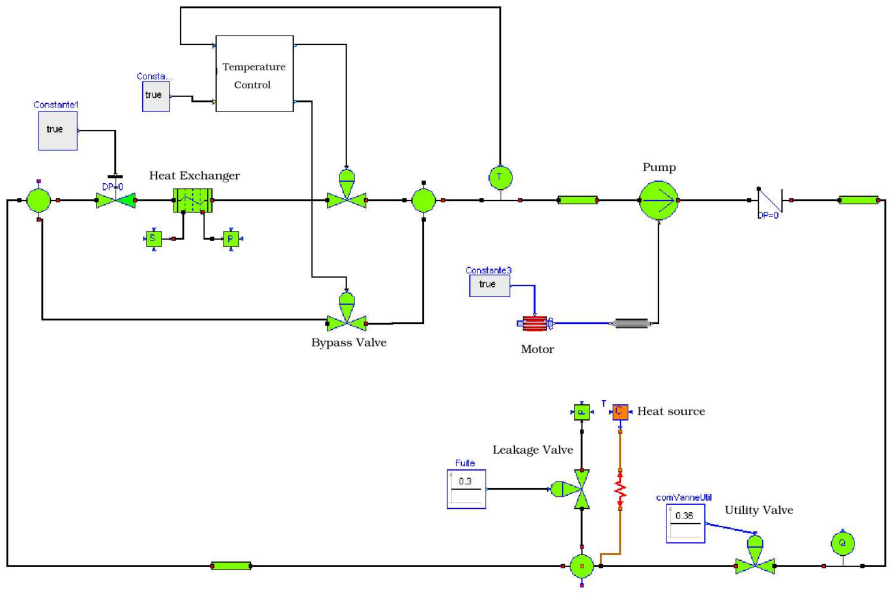


Figure 7.3.3: Model of a cooling system for auxiliary machines in a PWR power plant (courtesy of EDF).

The model of components has been developed in the Modelica language and can be found in the thermo-hydraulic toolbox of Scicos. As an example, the model of the hydraulic pump and the heat exchanger are given in Appendix B.

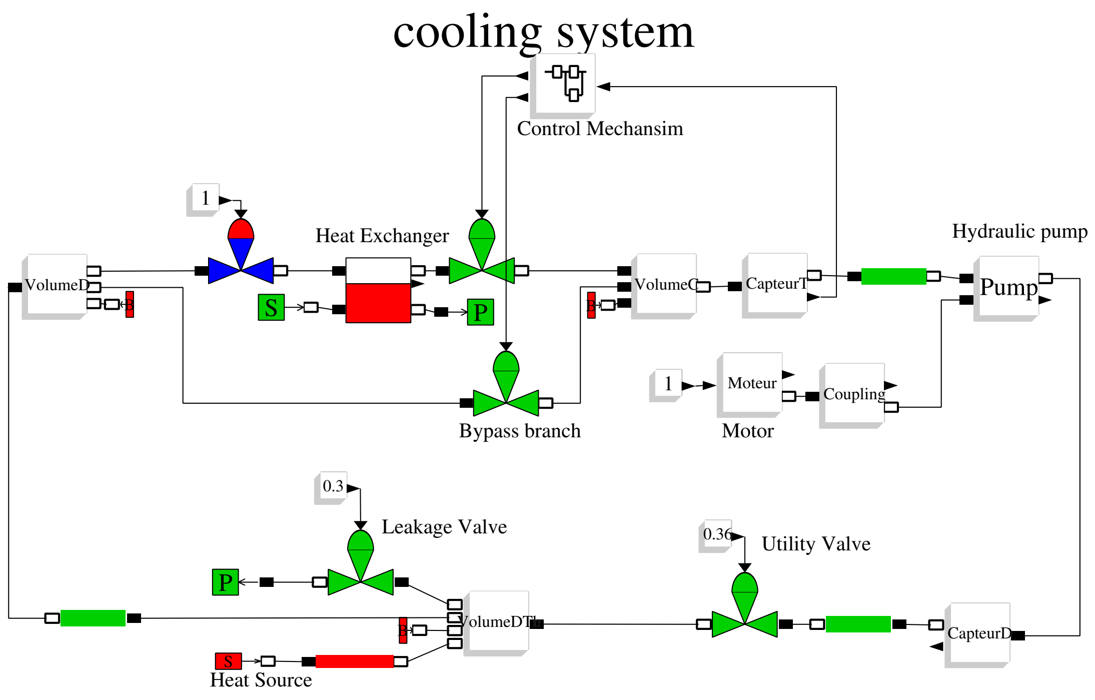


Figure 7.3.4: Scicos model of the cooling system of Fig 7.3.3.

The Scicos model of the simplified SRI system of Fig. 7.3.3 is depicted in Fig. 7.3.4. The machine that should be cooled down have been modeled with the **heat source** block. An electrical **motor** coupled with a **centrifuge pump** runs the water through the pipes from the heat source into the **heat exchanger**. The utility valve is used to model the pressure loss in the circuit. The leakage valve is used to model a possible leakage in the circuit. The (**CapteurT**) block measures the temperature of the mixture of the cooled water and water of the bypass branch. This measure is used in the feedback control mechanism (depicted in Fig. 7.3.5 and 7.3.6) to control the water flow through valves.

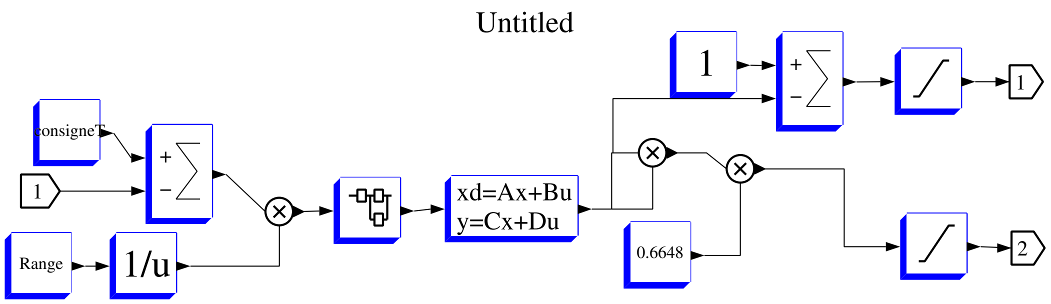


Figure 7.3.5: The control mechanism in Fig 7.3.4.

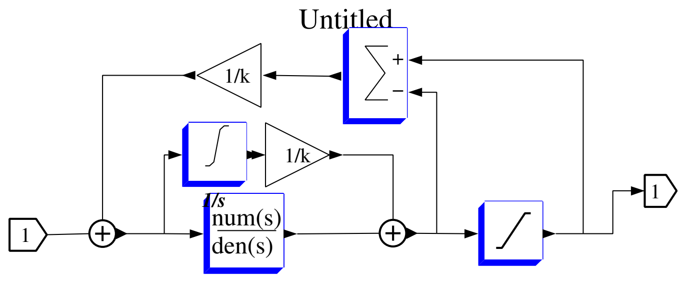


Figure 7.3.6: The control mechanism in Fig 7.3.4 (continue).

The Scicos code generator generates a DAE with 55 states, 75 `mode` variables and 75 zero-crossing functions for the implicit part of the model. The explicit part (the control mechanism) contains 2 states and 3 zero-crossing functions. The initial water temperature is 290 degrees Kelvin, and the ideal temperature is 290.15. The motor is running at the steady state speed, but it is not coupled to the pump. At time=0, the heat source (with temperature of 323.15 degrees Kelvin) is introduced in the circuit and the pump is coupled to the motor. At the beginning of the simulation, there are some transient behaviors then after the water temperature reaches 290.15 degrees Kelvin, which is desired temperature. Simulation result of some system variables such as the water temperature in the main circuit (Tw), water flow rate in the hot inlet of the heat exchanger (Qhot), water flow in the main circuit (Qt), and the rotational speed of the centrifuge pump (Wp) have been shown in Fig. 7.3.7. The simulation results were validated in EDF with the Dymola software.

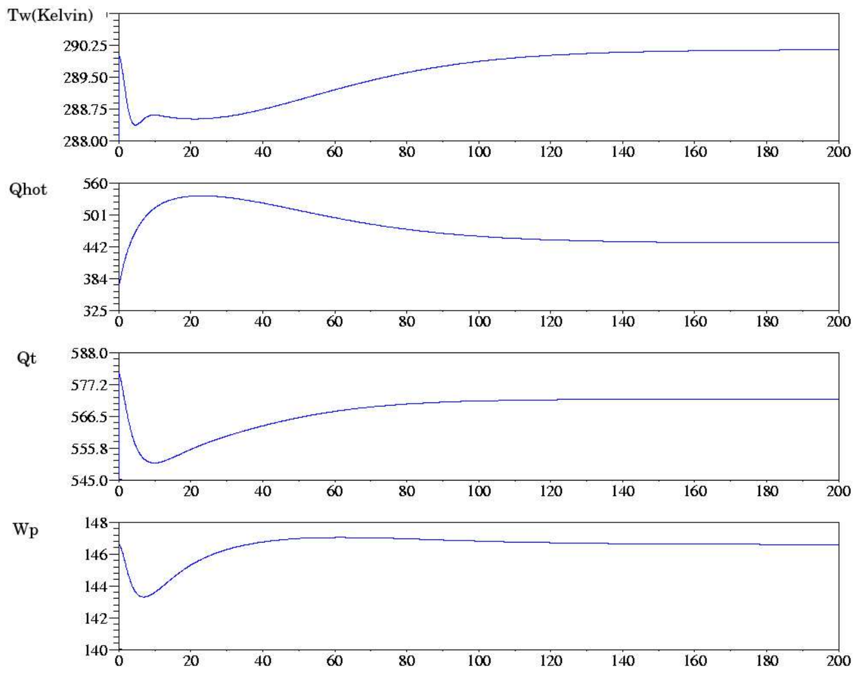


Figure 7.3.7: Simulation Result of the Model of Fig. 7.3.4.

Chapter 8

Conclusions

Hybrid system modeling is being increasingly applied in several engineering fields. Such modeling requires a hybrid specification formalism. Also, it is desirable to be able to simulate hybrid models. In this thesis, Scicos as a hybrid modeling and simulation software has been presented. In this chapter, this work is evaluated and suggestions for future work are summarized.

8.1 Contributions

Scicos can be used to model and simulate a wide range of dynamical systems. Modeling of physical system often ends up to an ODE or a DAE. A numerical solver should be used to integrate the differential equations. In this thesis, the DASKR solver was interfaced with Scicos to integrate DAEs. The ODE and DAE solvers have many control parameters which should be used to optimize the simulation in different situations. Most of the time, the user does not care about these parameters and obtaining a result is the main concern. Indeed, controlling the solver which should be done automatically and should remain transparent to the user, is a complex matter due to the interactions between the continuous-time dynamics and the rest of the system that can be a discrete-time system.

The discrete-time part affects the continuous-time part through simple connections and events. At events times, the value of the discontinuous signals jumps, creating discontinuities in the continuous-time signal. In such situations, for example, the simulator must make sure that the ODE/DAE solver does a cold restart otherwise it may fail. But cold restarting the numerical solver is costly and time consuming, so the solver should be cold restarted only if it is necessary. In this thesis, an event classification is introduced to distinguish the events that can cause a discontinuity in the continuous-time signals, which are called critical events. The critical events are used in two ways by the solver; First, just after the activation of a critical event, since a discontinuity may have been taken place, the numerical solver should be cold-restarted. Then, the activation time of the nearest critical event can be defined as the *stopping time*. Normally, the solver is allowed to step beyond the final integration time and return the value at the final time by interpolation. The stopping time option can be used to forbid the solver to advance the time beyond the discontinuity point that may occur after the critical events.

The continuous-time can also affect the discrete-time parts of the model by generating an event. In order to generate such an event, the the numerical solver must know when to stop the integration and generate a signal. This is done using the *zero-crossing* facility available in most numerical solvers. Using this facility, however, in the hybrid context like in Scicos, is not straightforward. In many cases in hybrid systems we need the direction in which a signal has crossed the zero. In fact, we need to find the crossing direction as well as the crossing time. In the original DASKR/LSODAR codes, only the crossing time of the zero-crossing is given. In this thesis, some modifications were made in DASKR/LSODAR codes to provide the direction of the crossing zero. This modification in DASKR was proposed to its authors and is now included in the official version of DASKR.

The original zero-crossing facility of the DASKR/LSODAR solvers has a problem with zero sticking, i.e., when a zero-crossing surface remains on zero at least for one integration steps, the solver stops the integration and does not advance. This phenomenon is common in hybrid systems where the system model may change just after passing a threshold (which is, of course, a zero-crossing surface). This situation is not properly dealt with in the solvers. In order to detect the zero-crossings, the sign of the zero-crossing surfaces are checked at the two ends of each integration step. There are some flaws in this procedure; First, at the beginning of the integration, the solver takes a very small step with an explicit first order Euler method to find weather there is a stuck zero or not. If there is a stuck, the solver announces an error and does not integrate. Furthermore, the small step taken at the beginning, can cause problem in certain cases. And finally, at the end of each integration step, if there is a sign change in the zero-crossing values, the solver returns back in time to find the exact crossing point. But if the surface value is zero, the same point is announced as a zero, which is not always the leftmost zero point. With an appropriate modeling or programing some of this situations can be properly handled, but for an ordinary user who does not know or does not want to bother with the details, this is an obstacle. In order to handle these problems, in this thesis, the automatic masking and unmasking of stuck zeros was introduced and implemented in the DASKR/LSODAR code. With this modification, at the beginning of the integration, without taking the small step, if there is a zero valued zero-crossing surface, it is masked until its value becomes non-zero. When a zero-crossing surface sticks to zero, unlike the original solver, it comes back in time and pinpoints the exact point where the signal touches zero. When a surface is stuck at zero, the solver monitors the stuck zeros and if a stuck zero detaches from zero the exact detaching time is found and solver stops to report an unmasking.

Most of the problems in using standard solvers in a hybrid environment are common to both ODE and DAE solvers. However, there is an additional difficulty with the DAE case: the problem of re-initialization and finding a consistent initial conditions. DASKR can find a consistent initial condition. DASKR treats the DAE as a nonlinear system of equations and solves it for the algebraic states and the derivatives of the differential states. In this thesis, the DASKR solver was used to perform two tasks, i.e., computing initial condition of DAE and then integration the DAE. This permits to have more flexibility in the simulator. DASKR uses the Newton's method to solve the initialization system of equation. Since the initialization system is a nonlinear function of variables, the Jacobian matrix has to be computed and factorized in each iteration. In DASKR, the computation of Jacobian is not performed at every step. This, of course, is reasonable to do during simulation for efficiency and justified by the fact that during the simulation, we have a very good initial guess for every nonlinear problem we need to solve. But in computing the initial condition, the initial guess can be far off due to the discrete-time behavior of the system. For most problems, however, we do not encounter initialization problem and the performance of DASKR is satisfactory, in particular, when the initial guess concerning the initial condition is not far from the actual initial condition. But this is not always the case and this does occur in Scicos applications. We have seen some problems that when the guess values are far from the solution, the old Jacobian does not indicate the right direction and the initialization fails. This problem is more likely to happen when the solver takes big steps and there is a large scaling difference between the variables. In this thesis, the DASKR code was modified to have a more frequent Jacobian evaluation. This modification increases the chances of converges. In fact, we have taken a conservative approach by forcing a Jacobian re-evaluation at every step during initialization. The overhead is not considerable because this concerns only Jacobian computations for the purpose of computing consistent initial conditions. Such computations are rare events (cold restarts) compared to the simulation effort which is permanent.

When the simulation time is long, the time precision at large time points, is reduced. In DASKR the `hmin` (minimum step-size) parameter is computed as a function of the simulation time. As time advances, the `hmin` value increases. In some application we saw that this value was bigger that what it should be and it causes some problems specially with zero-crossing detection. In this thesis, the DASKR code was modified to provide a minimum step-size smaller that what was proposed by DASKR. This modification helped us to simulate some applications that we could not simulate with the official version of DASKR.

In initial condition calculation, the differential variables are considered to be known and the DAE is solved for the algebraic variables and the derivatives of the differential variable. This classification emphasizes the physical role of the variables. The distinction allows specification of initialization before a solver cold-restart. In this thesis this classification was used in Scicos for the DAE initialization. There is another classification, i.e., differential and algebraic equations in an implicit ODE or a semi-explicit DAE. This classification can be used to reduce the round off error. In order to reduce the round-off error, a semi-explicit DAE can be scaled. In fact, scaling the algebraic equations of a semi-explicit DAE reduces the condition number of the Jacobian matrix. In this thesis, the Scicos simulator was developed to scale the DAEs. This scaling results in a less round-off error and a better convergence, specially for the stiff problems.

Modeling a physical system very often leads to a differential equation with discontinuities or a multi-model differential equation. A multi-model formulation is a way of describing non-smooth multi-model systems in terms of a finite number of smooth systems. In general, a multi-model system is defined via some conditional statements (If-then-Else, Switch). In other word, a hybrid model may be composed of several models such that each model is valid in a certain region. This regional separation with smoothness assumption in each region is very important because non-smooth systems cannot be fed directly to the solver. Because the numerical solvers assume that the state variables and their first derivatives are continuous. To overcome this problem, the numerical solver should use only one ODE/DAE up to the discontinuity point and then use the next ODE/DAE. In this thesis, the Scicos simulator was developed to handle the multi-model and the discontinuous systems. In most cases, the discontinuity is unpredictable, i.e., it should be detected. So it was proposed to define zero-crossing functions to detect and localize the discontinuities. With this zero-crossing function, the solver can find the exact discontinuity time. But for localizing this point, the solver should step over this discontinuity point. That means using another set of ODE/DAE. Using the second set of ODE/DAEs beyond the discontinuity point, normally results in failure in simulation. In this thesis, a `mode` variable was proposed to associate with the zero-crossing functions of the discontinuities of the multi-model system. The `mode` variables assign the DAE/ODE set that should be used by the solver to be integrated. The ODE/DAE is integrated until the solver detects a discontinuity, then another equation set is used.

Computing a consistent initial condition is a major task in integration of DAEs and this initialization becomes particularly difficult when the DAE is not smooth and, in particular, when it is a multi-model DAE. The search for the initial condition in this case is not just the classical problem of finding zeros of smooth functions but it is interlaced with searching for the correct `mode` set. Indeed, each `mode` set implies a smooth function and the solution of this function may imply another `mode` set. In this thesis, piecewise linear, homotopy, and the trail-and-error methods were tested. The two first methods provide a global convergence but they need an initial direction that is difficult to obtain. Therefore, the third method was adopted and implemented in Scicos.

Besides the modifications in the numerical solvers and the development of the Scicos simulator, an important contribution of this thesis is the simulation of component based models in Scicos. To extend the capacity of Scicos to allow component level modeling, we adopted the Modelica language for describing the algebraic-differential constraints imposed on the input/outputs of the implicit blocks. Modelica is primarily a modeling language that allows specification of mathematical models of complex natural or man made systems. To be able to simulate a model, written in the Modelica language, Modelica model should be translated into a code usable by the Scicos compiler and simulator. A Modelica compiler has been developed to translate the Modelica code into a standard Scicos block. Modelica compiler takes the model's symbolic equations and after causality analysis and several simplifications, generates a C code for a new Scicos block. During this thesis, in order to test and debug the Modelica compiler of Scicos[1], several industrial models from EDF company were tested in Scicos and validated with EDF results. These tests leaded us to develop the Scicos simulator and showed us what was needed to be developed in Scicos and in the Modelica Compiler.

the Modelica compiler builds a DAE set from the Modelica model provided by Scicos. This DAE is delivered to the numerical integrator. The obtained DAE may be a large system of equations. Selecting and eliminating auxiliary states may be important as the size of the system grows. However, this must be done carefully. Reducing system size may reduce sparsity of the coefficients and actually slow down the simulation. Selecting different subsets of states to eliminate may result in different sets of equations. Although from a mathematical viewpoint all these equations are equivalent, the result of simulating the models may be quite different. During this thesis an extensive testing was done, it was found that for some surprisingly simple problems that formulation can make a big difference in ways that cannot always be predicted. In some cases reformulations can be suggested that often improve performance. In the adaption of DASKR to Scicos and the need to be able to perform "black box" simulations for a variety of users we have had to tune the code in several ways some of which are mentioned in this thesis. The numerical difficulties that we faced during this thesis leaded us to perform some new developments to have a better simulation. This adaptation is an ongoing process.

The generated DAE for the implicit part of the Scicos model can be very stiff and very difficult to integrate by the solver. In the stiff regions, the solver reduces the step-size to meet the requested accuracy. But with a very small step-size, the numerical Jacobian does not necessarily indicate the minimum descent direction. So an analytical Jacobian evaluation is suggested in these cases. Since in the Modelica code generation we had the symbolic information, we used them to compute the analytical Jacobian for the model. The analytical

[1]Modelica compiler of Scicos is developed by Imagine Co.

Jacobian was implemented and used to improve the simulation accuracy. But in Scicos, the symbolic information is only available for the components or implicit blocks and the explicit blocks are defined with computer programs and are considered as black boxes. So the global analytical Jacobian is not available. In this thesis, the partial available analytic Jacobian was used to obtain the global Jacobian. Since most of the time the stiff part of the model is found in the implicit part, this global Jacobian provides a more accurate Jacobian than a purely numerical Jacobian.

AMESim is a modeling and simulation software for special domains such as hydraulics, mechanics, etc. In this thesis, an interface between Scicos and AMESim was established in order to model the systems in AMESim and simulate them in Scicos. Now, the user can develop a model in AMESim and import it to Scicos. The imported AMESim model is treated as a single Scicos block and the user can complete the design process and simulate it in Scicos. This permits the user to use the control, optimization, signal processing, etc capabilities of Scilab/Scicos softwares to design and simulate the models. This interface provides an easy, natural, direct-link between the two packages and enables a single engineer to run both packages on one machine or alternatively for two specialist engineers to run the software packages on adjacent machines.

8.2 Future Works

In this thesis, several areas have been identified for future research and development. In Scicos, the Modelica language is used to model the implicit blocks in component level modeling. The future works are mainly consists of developing the Modelica compiler of Scicos. The capabilities of the Modelica compiler of Scicos can further be enhanced by the following language extensions:

- **Model simplification:** the models written in Modelica language are usually big and they include many trivial relations between variables. The first task of the compiler is the reduction of the systems of equations by symbolic manipulation. This stage can be further optimized to get simpler models. This simplification, however, should be performed with care, in order to obtain a non-stiff DAE.

- **High index DAEs:** The mathematical models of many physical systems are high index DAEs. These DAEs cannot be integrated directly with DASKR, unless the index is reduced to one. In the literature, several algorithms have been introduced for index reduction or direct integration of high index DAEs. However, using high index DAEs is a challenging task, because, in high index system not all differential variables can be initialized freely, as it is currently assumed for differential variables. Therefore, this approach requires some developments in the Scicos simulator.

- **DAE initialization:** a DAE should be initialized before being integrated. So far, only the differential variables can be initialized. This option is not enough and the `initial equations` construct is required to give more flexibility to the initialization. These equations determines the initial values of the variables (differential or algebraic variables) with an algebraic equation set, rather than defining the initial values with constants. In order to be able to perform these initializations, the Scicos simulator, should be developed to handle multiple initialization cases of models.

- **Discrete models:** even though Modelica is a rich language having the capacity to handle continuous-time and discrete-time behaviors, Scicos uses mainly Modelica and implicit blocks to model continuous-time dynamics. Currently, only minimal support is provided for discrete-time behavior. The next stage in developing Scicos is considering discrete-time event part of Modelica and generating an appropriate code. The development will specially concerns the generating and handling of discrete-time events in Modelica. In order to use these developments in Scicos, the Scicos simulator should also be modified to handle the new features.

- **Algorithms statement:** unlike other programming languages, Modelica models are represented by physical equations. `Algorithms` provides a way to define a series of code that are executed sequentially. `Algorithms` in Modelica can be seen as a non invertible functions that provide computing several variables simultaneously.

- **Inverse model:** in some industrial applications, we are looking for a parameter or an input value that produces some specified output value. The first step in these applications is looking for the possibility of obtaining the inverse dynamical model of the system in terms of the specified inputs and outputs. The Modelica compiler can be developed to check and construct the inverse model.

8.3 Publications

- M. Najafi, R. Nikoukhah, "On Jacobian evaluation for numerical integration of DAEs with partial symbolic information", 44th IEEE Conference on Decision and Control, December, 2005, Seville, Spain.

- M. Najafi, R. Nikoukhah, Serge Steer, Sebastien Furic, "New features and new challenges in modeling and simulation in Scicos", IEEE conference on control application, August, 2005, Toronto, Canada.

- M. Najafi, R. Nikoukhah, and S. L. Campbell, "The role of model formulation in DAE integration: Experience gained in developing Scicos", 17th IMACS World Congress Mathematics and Computers in Simulation, July, 2005, Paris, France.

- M. Najafi, S. Furic, R. Nikoukhah, "Scicos: a general purpose modeling and simulation environment", 4th International Modelica Conference, March, 2005, Hamburg, Germany.

- M. Najafi, A.Azil, and R. Nikoukhah, "Extending scicos from system to component level simulation", ESMc2004 international Conference, October, 2004, Paris, France.

- M. Najafi, R. Nikoukhah,"ODE and DAE solvers in Scicos environment", Iasted International Conference on applied simulation and modeling, ASM2004, June, 2004, Rhodes, Greece.

- M. Najafi, R. Nikoukhah, S. L. Campbell, "Computation of consistent initial conditions for multi-mode DAEs: Application to Scicos", IEEE International Symposium on Intelligent Control Computer Aided Control Systems Design, September, 2004, Taipei.

- A. Azil, M. Najafi and R. Nikoukhah, "Conditioning and Scicos diagram compiler", 5th EUROSIM Congress on Modeling and Simulation, September, 2004, Paris, France.
- A. Azil, M. Najafi, and R. Nikoukhah, "Génération de code dans Scicos: le cas continu", Journée Nationale LMCS 2004, November, 2004, Paris, France.
- M. Najafi, A. Azil, R. Nikoukhah, "Implementation of continuous-time dynamics in Scicos", 15TH European Simulation Symposium and Exhibition (ESS), 2003, Delft, The Netherlands.

Chapter 9

Conclusions (en français)

La modélisation des systèmes hybrides est de plus en plus appliquée dans les domaines de technologie. Cette thèse traite la modélisation et la simulation des modèles hybrides en utilisant le logiciel Scicos. Dans ce chapitre on va présenter les nouveautés de Scicos réalisées durant cette thèse et quelques perspectives pour les futurs travaux.

9.1 Nouveautés

Scicos peut être employé pour la modélisation et la simulation des systèmes dynamiques hybrides. La modélisation du système physique finit souvent par une EDO ou par une EAD. Un solveur numérique doit être employé pour intégrer les équations. Dans cette thèse, le solveur DASKR a été interfacé avec Scicos pour intégrer les EADs. Les solveurs d'EDO et d'EAD ont de nombreux paramètres de commande qui devraient être employés pour optimiser la simulation dans les diverses situations. La plupart du temps, l'utilisateur ne se préoccupe pas de ces paramètres et l'obtention d'un résultat est son souci principal. En effet, le contrôle du solveur doit être automatique et transparent pour l'utilisateur. Ce contrôle peut être compliqué à cause des interactions entre la dynamique temps continu et le reste du système qui peut être un système temps discret. La partie temps discret affecte la partie temps continu par les interconnexions et les événements. Au moment des événements, la valeur des signaux discontinus varie brusquement, créant des discontinuités dans le signal temps continu. Dans une telle situation, par exemple, le simulateur doit s'assurer que le solveur d'EDO/EAD redémarre à froid, sans quoi la simulation peut échouer. Mais le redémarrage à froid du solveur numérique est long et coûteux. Ce redémarrage à froid ne doit donc est effectué que si nécessaire. Dans cette thèse, une classification d'événements est présentée pour repérer les événements qui peuvent causer une discontinuité dans les signaux temps continu. Ces événements s'appellent événements critiques. Le solveur emploie les événements critiques de deux manières : d'abord, un événement critique peut causer une discontinuité, dans ce cas le solveur numérique doit redémarrer à froid. Ensuite, le temps d'activation de l'événement critique le plus proche (programmé dans le futur) peut être défini comme le temps d'arrêt du solveur. Normalement, on permet au solveur d'intégrer au-delà du temps final d'intégration et de renvoyer le résultat à l'instant final en faisant une interpolation. L'option de temps d'arrêt peut être employée pour interdire au solveur d'avancer au-delà du point de discontinuité qui peut se produire après des événements critiques. La partie temps continu peut également affecter la partie temps discret du modèle en produisant des événements. Afin de produire un tel événement, le solveur numérique doit savoir quand arrêter l'intégration et produire un signal ou un événement. Ceci est fait en utilisant l'option de zéro-croisement disponible dans la plupart des solveurs numériques. Cependant, l'utilisation de cette option dans un contexte hybride comme Scicos n'est pas évident. Dans de nombreux cas de systèmes hybrides, nous avons besoin de la direction dans laquelle un signal a croisé le zéro. En fait, nous devons trouver la direction, aussi bien que le temps, de croisement. Dans les codes originaux de DASKR/LSODAR, seul le temps de croisement du zéro est indiqué. Dans cette thèse, quelques modifications ont été faites dans ce code pour fournir la direction du croisement zéro. Cette modification dans DASKR a été proposée à ses auteurs et elle est maintenant incluse dans la version officielle.

La routine originale de zéro-croisement des solveurs DASKR/LSODAR pose problème avec des zéros coincés. En effet, quand une surface de zéro-croisement demeure sur zéro pour quelques pas d'intégration, le solveur s'arrête et renvoie un message d'erreur. Ce phénomène est commun dans les systèmes hybrides où le modèle peut changer juste après avoir passé un seuil (qui est, naturellement, une surface de zéro-croisement). Cette situation n'est pas correctement gérée dans les solveurs. Afin de détecter les zéro-croisements, les signes des surfaces de zéro-croisement sont vérifiés au début et à la fin de chaque pas d'intégration. Il y a quelques failles dans ce procédé ; d'abord, au début de l'intégration, le solveur prend un pas très petit en appliquant la méthode explicite d'Euler d'ordre 1, pour détecter si il y a des surfaces collées à zéro ou pas. Si oui, le solveur renvoie un message d'erreur et arrête l'intégration. En outre, le petit pas pris au début peut poser des problèmes dans certains cas. Enfin, à la fin de chaque étape d'intégration, s'il y a un changement de signe des valeurs de surfaces des zéro-croisements, le solveur revient en arrière dans le temps afin de trouver le point exact de croisement. Si la valeur de surface à la fin du pas est zéro, le même point zéro est annoncé, ce qui n'est pas toujours correct. Avec une modélisation ou programmation correcte, une partie de ces situations peut être correctement manipulée, mais pour un utilisateur ordinaire qui ne sait pas ou ne veut pas se tracasser avec les détails, c'est un obstacle. Afin de surmonter ces problèmes, dans cette thèse, le masquage et le démasquage automatique des surfaces collées à zéro a été présenté et mis en application dans le code DASKR/LSODAR. Avec cette modification, au début de l'intégration, sans prendre le petit pas, s'il y a une surface collée à zéro, elle est masquée jusqu'à ce que sa valeur devienne différente de zéro. Quand une surface de zéro-croisement se colle à zéro, la routine modifiée de zéro-croisement retrouve le point exact où la surface colle à zéro. Quand une surface est coincée sur zéro, le solveur masque cette surface et la surveille jusqu'à ce qu'elle se décolle de zéro. Le solveur, ensuite, retrouve et annonce le temps exact. La plupart des problèmes en employant les solveurs standard dans un environnement hybride sont communs aux solveurs d'EDO et d'EAD. Cependant, il y a des difficultés additionnelles dans le cas d'EAD : le problème de la réinitialisation est de trouver des conditions initiales consistantes. DASKR peut trouver un état initial consistant. DASKR traite l'EAD comme un système non linéaire des équations et la résout pour les états algébriques et les dérivées des états differentiels. Dans cette thèse, le solveur DASKR a été employé pour exécuter deux tâches : d'abord pour trouver l'état initial de calcul d'EAD, ensuite pour intégrer l'EDA. Ceci donne plus de flexibilité au simulateur de Scicos.

DASKR emploie la méthode de Newton pour résoudre le système d'initialisation de l'équation. Puisque ce système est une fonction non linéaire des variables, la matrice Jacobienne doit être calculée et factorisée à chaque itération. Dans DASKR , le calcul de cette matrice ne s'effectue pas à chaque étape. Ceci est raisonnable, car pendant la simulation, la matrice Jacobienne change peu et cette approximation est satisfaisante. Mais en calculant l'état initial, l'estimation initiale peut être très loin de la solution à cause des discontinuités causées par la partie temps discret du système. Pour la plupart des problèmes, cependant, nous ne rencontrons pas le problème d'initialisation et l'exécution de DASKR est satisfaisante, en particulier quand l'état initial n'est pas loin de l'état initial consistant. Mais ce n'est pas toujours le cas et ceci se produit dans des applications de Scicos. Nous avons vu dans quelques problèmes que, quand les valeurs initiales sont loin de la solution, l'ancienne matrice Jacobienne n'indique pas la bonne direction et l'initialisation échoue. Ce problème se produit quand le solveur prend de grands pas de temps et qu'il y a une grande différence d'échelle entre les variables. Dans cette thèse, le code DASKR a été modifié pour avoir une évaluation plus fréquente de la matrice Jacobienne. Cette modification augmente les chances de convergence. En fait, nous avons adopté une approche conservatrice en forçant une réévaluation de Jacobienne à chaque étape pendant l'initialisation. La surcharge n'est pas considérable, parce-que ceci concerne seulement des calculs de Jacobienne pour calculer des conditions initiales consistantes.

Quand le temps de simulation est long, la précision aux grands temps est réduite. Dans DASKR le pas minimum `hmin` est calculé en fonction du temps de simulation. Quand la simulation avance, le temps augmente et la valeur de `hmin` augmente. Dans une certaine application, nous avons vu que cette valeur était trop grande et cela pose quelques problèmes, particulièrement avec la détection de zéro-croisement. Dans cette thèse, le code DASKR a été modifié pour fournir un pas d'intégration plus petit que ce qui était proposé dans la version originale. Cette modification nous a aidés à simuler quelques applications que nous ne pourrions pas simuler avec la version officielle DASKR .

Dans le calcul des conditions initiales, les variables sont réparties en deux classes : les variables différentielles et les variables algébriques. Les variables différentielles sont considérées connues et l'EAD est résolue pour les variables algébriques et les dérivées des variables différentielles. Cette classification souligne le rôle physique des variables différentielles. Dans cette thèse cette classification a été employée pour l'initialisation d'EAD. Il existe une autre classification : les équations différentielles et algébriques pour les EAD semi-explicites. Cette classification peut être employée pour réduire l'erreur d'intégration. Dans cette thèse, le simulateur de Scicos a été développé pour profiter de cette classification afin d'augmenter la précision d'intégration des EADs.

La modélisation d'un système physique mène très souvent à une équation avec des discontinuités ou des équations multi-modèles. Une formulation multi-modèles est une manière de décrire les systèmes non lisses en termes de nombres fini de systèmes lisses. En général, un système multi-modèles est défini par les phases conditionnelles (If-Then-Else, Switch). Un modèle hybride peut se composer de plusieurs modèles tels que chaque modèle est valide dans une certaine région. Cette séparation régionale est très importante parce que des systèmes non lisses ne peuvent pas être donnés directement au solveur : c'est à cause des solveurs numériques qui supposent la continuité des variables d'état et de leurs premières dérivées. Pour surmonter ce problème, le solveur numérique doit employer seulement une EDO/EAD jusqu'au point de discontinuité puis employer l'autre EDO/EAD au-delà. Dans cette thèse, le simulateur de Scicos a été développé pour manipuler les systèmes multi-modèles et les systèmes discontinus. Dans la plupart des cas, la discontinuité est imprévisible, c'est-à-dire qu'elle doit être détectée. Ainsi on lui a proposé de définir des fonctions de zéro-croisement pour détecter et localiser les discontinuités. Avec cette fonction de zéro-croisement, le solveur peut trouver le temps exact des discontinuités. Mais pour localiser ce point, le solveur devrait doit faire un pas au-delà de ce point de discontinuité. Ceci signifie utiliser une autre

EDO/EAD, et a normalement pour conséquence l'échec de la simulation. Dans cette thèse, nous avons proposé la variable `mode` pour s'associer aux fonctions de zéro-croisement des discontinuités du système de multi-modèles. La variable `mode` désigne l'EDO/EAD qui devrait doit être employée par le solveur à intégrer. L'EDO/EAD est intégrée jusqu'à ce que le solveur détecte une discontinuité, puis un autre système d'équations est intégré.

Le calcul d'un état initial consistant est un passage obligatoire et important, ceci devient particulièrement difficile quand l'EAD n'est pas lisse et, en particulier, quand il s'agit d'une EAD multi-modèles. Dans ce cas, la recherche de l'état initial n'est pas simplement le problème classique de trouver des zéros de fonctions lisses mais elle est accompagnée d'une recherche de configuration de mode correcte. En effet, chaque configuration de mode implique une EDO/EAD lisse et la solution de cette EDO/EAD peut impliquer une autre configuration de mode. Dans cette thèse, les méthodes linéaires par morceaux, ≪ Homotopy ≫, et les essais ≪trial and error≫ ont été examinés. Les deux premières méthodes fournissent une convergence globale mais elles ont besoin d'une direction initiale qui est difficile à obtenir. Par conséquent, la troisième méthode a été adoptée et appliquée dans Scicos.

Outre les modifications dans les solveurs numériques et le développement du simulateur de Scicos, une partie importante de cette thèse a été attribuée à la simulation des modèles basés sur des composants dans Scicos. Pour étendre la capacité de Scicos à modéliser des systèmes hybrides, nous avons adopté le langage Modelica pour décrire les contraintes algébro-différentielles imposées à l'entrée/sortie des blocs implicites. `Modelica` est principalement un langage de modélisation qui permet de spécifier les modèles mathématiques de systèmes. Afin de pouvoir simuler un modèle écrit en langage `Modelica`, les modèles `Modelica` doivent être traduits en code utilisable par le compilateur et le simulateur de Scicos. Un compilateur `Modelica` a été développé pour traduire le code Modelica en bloc standard de Scicos. Le compilateur Modelica prend les équations symboliques du modèle et après une analyse de causalité et plusieurs simplifications, produit un code C pour un nouveau bloc Scicos. Pendant cette thèse, afin d'examiner et corriger le compilateur Modelica de Scicos [1], plusieurs modèles industriels de la compagnie EDF [2] ont été examinés dans Scicos et validés avec des résultats EDF. Ces essais nous ont permis de développer le simulateur de Scicos et de nous démontrer ce qui était nécessaire pour le développement du simulateur de Scicos et du compilateur Modelica.

Le compilateur Modelica génère un ensemble d'EAD à partir d'un modèle Modelica fourni par Scicos. Cette EAD, livrée au solveur numérique, peut être un grand système d'équations. Le choix et l'élimination des états auxiliaires peuvent être importants quand la taille du système augmente. La réduction de la taille du système peut directement affecter et faire ralentir la simulation. Le choix de différents sous-ensembles d'états pour éliminer peut avoir pour conséquence différents ensembles d'équations. Bien que d'un point de vu mathématique toutes ces équations soient équivalentes, les résultat de la simulation des modèles peuvent être tout à fait différents. Pendant cette thèse un essai étendu a été effectué, et il a été constaté que pour quelques problèmes étonnamment simples la formulation peut faire une grande différence. Les difficultés numériques rencontrées pendant cette thèse nous ont permis d'effectuer quelques nouveaux développements pour obtenir une meilleure simulation.

L'EAD générée pour la partie implicite d'un modèle Scicos peut être très raide et très difficile à intégrer par le solveur. Dans les régions raides, le solveur réduit le pas d'intégration

[1] le compilateur Modelica de Scicos est développé chez par la société Imagine.

[2] Électricité De France

pour parvenir à la précision demandée. Mais avec un pas très petit, la matrice Jacobienne numérique n'indique pas nécessairement la bonne direction de descente vers le minimum. Ainsi une évaluation analytique de Jacobienne est suggérée dans ce cas. Puisque dans la génération de code Modelica nous disposons de l'information symbolique du modèle, nous l'avons utilisée pour calculer la matrice Jacobienne analytique. La matrice Jacobienne analytique a été mise en application et employée pour améliorer la précision de simulation. Mais dans Scicos, l'information symbolique est disponible uniquement pour la partie implicite où les composants et les blocs explicites sont définis avec des programmes machine et sont considérés comme des boîtes noires. Ainsi la matrice Jacobienne analytique global n'est pas disponible. Dans cette thèse, la matrice Jacobienne analytique, disponible partiellement, a été employé pour obtenir la matrice Jacobienne globale. Puisque la plupart du temps la partie raide du modèle se trouve dans la partie implicite, cette matrice Jacobienne globale fournit une matrice Jacobienne plus précise qu'une matrice Jacobienne purement numérique.

`AMESim` est un logiciel de modélisation et de simulation utilisé dans des domaines particuliers tels que l'hydraulique, la mécanique, etc. Dans cette thèse, une interface entre Scicos et `AMESim` a été établie afin de modéliser des systèmes dans `AMESim` et les simuler dans Scicos. Maintenant, l'utilisateur peut développer un modèle dans `AMESim` et l'importer dans Scicos. Le modèle importé est traité comme bloc simple de Scicos et l'utilisateur peut compléter le processus de conception et le simuler dans Scicos. Ceci permet à l'utilisateur d'employer les boites à outils contrôle commande, l'optimisation, le traitement du signal, etc. des logiciels Scilab/Scicos pour concevoir et simuler les modèles.

9.2 Perspectives

Dans cette thèse, plusieurs domaines ont été identifiés pour les futures recherches et développements. Dans Scicos, le langage Modelica est employé pour la modélisation des blocs implicites au niveau composant. Les travaux futurs se composent principalement du développement du compilateur Modelica de Scicos. Le compilateur Modelica de Scicos peut être développé pour couvrir les cas suivants:

- **Simplification de modèles :** les modèles écrits en langage Modelica sont habituellement grands et ils incluent beaucoup de relations insignifiantes entre les variables. La première tâche du compilateur est la réduction des systèmes d'équations en manipulation symbolique. Cette étape peut être encore optimisée pour obtenir des modèles plus simples. Cette simplification, cependant, doit être exécutée avec soin, afin d'éviter d'obtenir une EAD raide.

- **EADs d'indice élevé :** les modèles mathématiques de nombreux systèmes physiques sont des EADs d'indice élevé. Ces EADs ne peuvent pas être intégrées directement avec DASKR, à moins que l'indice soit réduit à 1. Dans la littérature, plusieurs algorithmes ont été présentés pour la réduction d'indice ou l'intégration directe des EADs d'indice élevé. Cependant, employer les EADs d'indice élevés est très difficile, parce que dans un système d'indice élevé, les variables différentielles ne peuvent pas être initialisées librement. Par conséquent, cette approche exige quelques développements dans le simulateur de Scicos.

- **Initialisation d'EAD :** une EAD doit être initialisée avant d'être intégrée. Jusqu'à maintenant, seules les variables différentielles peuvent être initialisées. Cette option n'est pas suffisante, la construction de `initial equation` de Modelica est nécessaire pour donner plus de flexibilité à l'initialisation. Ces équations déterminent les valeurs initiales des variables (variables différentielles ou algébriques) avec un ensemble algébrique d'équations. Afin de pouvoir exécuter ces initialisations, le simulateur de Scicos doit être développé pour manipuler des cas multiples d'initialisation de modèles.

- **Modèles discrets :** quoique Modelica soit un langage riche ayant la capacité de manipuler des comportements temps continu et de temps discret, Scicos emploie principalement Modelica et blocs implicites pour la modélisation des systèmes temps continu. Actuellement, seul un support minimal est fourni pour le comportement temps discret. La prochaine étape de travail sur Scicos est de développer les événements temps discret de Modelica et de produire un code approprié. Afin d'employer ces développements dans Scicos, le simulateur de Scicos doit également être modifié pour manipuler les nouveaux dispositifs.
- **Algorithm :** les modèles Modelica sont représentés par des équations physiques. L'`algorithm` fournit une méthode pour définir une série de codes qui sont exécutés séquentiellement. Les `algorithms` dans Modelica peuvent être vus en tant que fonctions non inversibles qui fournissent le calcul de plusieurs variables simultanément.
- **Modèle inverse :** dans quelques applications industrielles, nous recherchons un paramètre ou une valeur d'entrée qui produit une certaine valeur indiquée de sortie. La première étape dans ces applications est de chercher la possibilité d'obtenir le modèle dynamique inverse du système en termes d'entrées et sorties indiquées. Le compilateur de Modelica peut être développé pour vérifier et construire le modèle inverse.

9.3 Publications

- M. Najafi, R. Nikoukhah, "On Jacobian evaluation for numerical integration of DAEs with partial symbolic information", 44th IEEE Conference on Decision and Control, December, 2005, Seville, Spain.
- M. Najafi, R. Nikoukhah, Serge Steer, Sebastien Furic, "New features and new challenges in modeling and simulation in Scicos", IEEE conference on control application, August, 2005, Toronto, Canada.
- M. Najafi, R. Nikoukhah, and S. L. Campbell, "The role of model formulation in DAE integration: Experience gained in developing Scicos", 17th IMACS World Congress Mathematics and Computers in Simulation, July, 2005, Paris, France.
- M. Najafi, S. Furic, R. Nikoukhah, "Scicos: a general purpose modeling and simulation environment", 4th International Modelica Conference, March, 2005, Hamburg, Germany.
- M. Najafi, A.Azil, and R. Nikoukhah, "Extending scicos from system to component level simulation", ESMc2004 international Conference, October, 2004, Paris, France.
- M. Najafi, R. Nikoukhah,"ODE and DAE solvers in Scicos environment", Iasted International Conference on applied simulation and modeling, ASM2004, June, 2004, Rhodes, Greece.

- M. Najafi, R. Nikoukhah, S. L. Campbell, "Computation of consistent initial conditions for multi-mode DAEs: Application to Scicos", IEEE International Symposium on Intelligent Control Computer Aided Control Systems Design, September, 2004, Taipei.
- A. Azil, M. Najafi and R. Nikoukhah, "Conditioning and Scicos diagram compiler", 5th EUROSIM Congress on Modeling and Simulation, September, 2004, Paris, France.
- A. Azil, M. Najafi, and R. Nikoukhah, "Génération de code dans Scicos: le cas continu", Journée Nationale LMCS 2004, November, 2004, Paris, France.
- M. Najafi, A. Azil, R. Nikoukhah, "Implementation of continuous-time dynamics in Scicos", 15TH European Simulation Symposium and Exhibition (ESS), 2003, Delft, The Netherlands.

Appendix A

New DASKR's root finder

```
c     ================================================================
      SUBROUTINE DRCHEK2 (JOB, RT, NRT, NEQ, TN, TOUT, Y, YP, PHI, PSI,
     *     KOLD, R0, R1, RX, JROOT, IRT, UROUND, INFO3, RWORK, IWORK,
     *     RPAR, IPAR)
C
C***  BEGIN PROLOGUE  DRCHEK
C***  REFER TO DDASKR
C***  ROUTINES CALLED  DDATRP, DROOTS, DCOPY, RT
C***  REVISION HISTORY  (YYMMDD)
C     020815  DATE WRITTEN
C     021217  Added test for roots close when JOB = 2.
C***  END PROLOGUE  DRCHEK
C
      IMPLICIT DOUBLE PRECISION(A-H,O-Z)
C Pointers into IWORK:
      PARAMETER (LNRTE=36, LIRFND=37)
C     Pointers into RWORK:
      PARAMETER (LT0=51, LTLAST=52)
      EXTERNAL RT
      INTEGER JOB, NRT, NEQ, KOLD, JROOT, IRT, INFO3, IWORK, IPAR
      DOUBLE PRECISION TN, TOUT, Y, YP, PHI, PSI, R0, R1, RX, UROUND,
     *     RWORK, RPAR
      DIMENSION Y(*), YP(*), PHI(NEQ,*), PSI(*),
     *          R0(*), R1(*), RX(*), JROOT(*), RWORK(*), IWORK(*)
      INTEGER I, JFLAG, LMASK
      DOUBLE PRECISION H
      DOUBLE PRECISION HMINR, T1, TEMP1, TEMP2, X, ZERO
      LOGICAL ZROOT,Mroot
c     -------------- masking -----------------
      PARAMETER (LNIW=17)
      DATA ZERO/0.0D0/

c     -------------- masking -----------------

C-------------------------------------------------------------------
C This routine checks for the presence of a root of R(T,Y,Y') in the
C vicinity of the current T, in a manner depending on the
```

```
C input flag JOB.  It calls subroutine DROOTS to locate the root
C as precisely as possible.
C
C In addition to variables described previously, DRCHEK
C uses the following for communication..
C JOB    = integer flag indicating type of call..
C          JOB = 1 means the problem is being initialized, and DRCHEK
C                  is to look for a root at or very near the initial T.
C          JOB = 2 means a continuation call to the solver was just
C                  made, and DRCHEK is to check for a root in the
C                  relevant part of the step last taken.
C          JOB = 3 means a successful step was just taken, and DRCHEK
C                  is to look for a root in the interval of the step.
C R0     = array of length NRT, containing the value of R at T = T0.
C          R0 is input for JOB .ge. 2 and on output in all cases.
C R1,RX  = arrays of length NRT for work space.
C IRT    = completion flag..
C          IRT = 0  means no root was found.
C          IRT = -1 means JOB = 1 and a zero was found both at T0 and
C                   and very close to T0.
C          IRT = -2 means JOB = 2 and some Ri was found to have a zero
C                   both at T0 and very close to T0.
C          IRT = 1  means a legitimate root was found (JOB = 2 or 3).
C                   On return, T0 is the root location, and Y is the
C                   corresponding solution vector.
c          IRT = 2  A zero-crossing surface has detached from zero
c
C T0     = value of T at one endpoint of interval of interest.  Only
C          roots beyond T0 in the direction of integration are sought.
C          T0 is input if JOB .ge. 2, and output in all cases.
C          T0 is updated by DRCHEK, whether a root is found or not.
C          Stored in the global array RWORK.
C TLAST  = last value of T returned by the solver (input only).
C          Stored in the global array RWORK.
C TOUT   = final output time for the solver.
C IRFND  = input flag showing whether the last step taken had a root.
C          IRFND = 1 if it did, = 0 if not.
C          Stored in the global array IWORK.
C INFO3  = copy of INFO(3) (input only).
C-----------------------------------------------------------------------
C
      H = PSI(1)
      IRT = 0
      LMASK=IWORK(LNIW)-NRT
      HMINR = (ABS(TN) + ABS(H))*UROUND*100.0D0
      GO TO (100, 200, 300), JOB
C
C Evaluate R at initial T (= RWORK(LT0)); check for zero values.--------
 100  CONTINUE

      DO 103 I = 1,NRT
         JROOT(I) = 0
         IWORK(LMASK+I)=0
 103  CONTINUE

      CALL DDATRP1(TN,RWORK(LT0),Y,YP,NEQ,KOLD,PHI,PSI)
      CALL RT (NEQ, RWORK(LT0), Y, YP, NRT, R0, RPAR, IPAR)
      IWORK(LNRTE) = 1
      DO 110 I = 1,NRT
         IF (DABS(R0(I)) .EQ. ZERO) THEN
            IWORK(LMASK+I)=1
         ENDIF
 110  CONTINUE
      RETURN
C     ==================================================================
 200  CONTINUE


c     in the previous call there was not a root, so this part can be ignored.
c      IF (IWORK(LIRFND) .EQ. 0) GO TO 260
       DO 203 I = 1,NRT
          JROOT(I) = 0
 203      IWORK(LMASK+I)=0
C     If a root was found on the previous step, evaluate R0 = R(T0). -------
       CALL DDATRP1 (TN, RWORK(LT0), Y, YP, NEQ, KOLD, PHI, PSI)
       CALL RT (NEQ, RWORK(LT0), Y, YP, NRT, R0, RPAR, IPAR)
       IWORK(LNRTE) = IWORK(LNRTE) + 1
       DO 210 I = 1,NRT
          IF (dABS(R0(I)) .EQ. ZERO) THEN
             IWORK(LMASK+I)=1
          ENDIF
 210   CONTINUE
C     R0 has no zero components.  Proceed to check relevant interval. ------
 260   IF (TN .EQ. RWORK(LTLAST)) RETURN
C     ====================================================
 300   CONTINUE
C     Set T1 to TN or TOUT, whichever comes first, and get R at T1. --------
       IF (INFO3 .EQ. 1 .OR. (TOUT - TN)*H .GE. ZERO) THEN
          T1 = TN
          GO TO 330
       ENDIF
       T1 = TOUT
       IF ((T1 - RWORK(LT0))*H .LE. ZERO) RETURN
 330   CALL DDATRP1 (TN, T1, Y, YP, NEQ, KOLD, PHI, PSI)
       CALL RT (NEQ, T1, Y, YP, NRT, R1, RPAR, IPAR)
       IWORK(LNRTE) = IWORK(LNRTE) + 1
C     Call DROOTS to search for root in interval from T0 to T1. -----------
      JFLAG = 0

      DO 340 I = 1,NRT
         JROOT(I)=IWORK(LMASK+I)
 340  CONTINUE

 350  CONTINUE
      CALL DROOTS2(NRT, HMINR, JFLAG,RWORK(LT0),T1, R0,R1,RX, X, JROOT)
      IF (JFLAG .GT. 1) GO TO 360
      CALL DDATRP1 (TN, X, Y, YP, NEQ, KOLD, PHI, PSI)
      CALL RT (NEQ, X, Y, YP, NRT, RX, RPAR, IPAR)
      IWORK(LNRTE) = IWORK(LNRTE) + 1
      GO TO 350

 360  CONTINUE
      if (JFLAG.eq.2) then      ! root found
         ZROOT=.false.
         MROOT=.false.
         DO 320 I = 1,NRT
            if(IWORK(LMASK+I).eq.1) then
               if(ABS(R1(i)).ne. ZERO) THEN
                  JROOT(I)=SIGN(2.0D0,R1(I))
                  Mroot=.true.
               ELSE
                  JROOT(I)=0
               ENDIF
            ELSE
               IF (ABS(R1(I)) .EQ. ZERO) THEN
                  JROOT(I) = -SIGN(1.0D0,R0(I))
                  zroot=.true.
               ELSE
                  IF (SIGN(1.0D0,R0(I)) .NE. SIGN(1.0D0,R1(I))) THEN
                     JROOT(I) = SIGN(1.0D0,R1(I) - R0(I))
                     zroot=.true.
                  ELSE
                     JROOT(I)=0
                  ENDIF
               ENDIF
            ENDIF
 320     CONTINUE

         CALL DDATRP1 (TN, X, Y, YP, NEQ, KOLD, PHI, PSI)

         if (Zroot) then
            DO 380 I = 1,NRT
               IF(ABS(JROOT(I)).EQ.2) JROOT(I)=0
 380        CONTINUE
            MROOT=.false.
            IRT=1
         endif
         IF (MROOT) THEN
            IRT=2
         ENDIF
      ENDIF
      RWORK(LT0) = X
      CALL DCOPY (NRT, RX, 1, R0, 1)
      RETURN
C----------------------END OF SUBROUTINE DRCHEk2 -----------------------
      END
      SUBROUTINE DROOTS2(NRT, HMIN, JFLAG, X0, X1, R0, R1, RX, X, JROOT)
C
C***BEGIN PROLOGUE  DROOTS
C***REFER TO DRCHEK
C***ROUTINES CALLED DCOPY
C***REVISION HISTORY  (YYMMDD)
C   020815  DATE WRITTEN
C   021217  Added root direction information in JROOT.
C***END PROLOGUE  DROOTS
C
      INTEGER NRT, JFLAG, JROOT
      DOUBLE PRECISION HMIN, X0, X1, R0, R1, RX, X
      DIMENSION R0(NRT), R1(NRT), RX(NRT), JROOT(NRT)
C-----------------------------------------------------------------------
C This subroutine finds the leftmost root of a set of arbitrary
C functions Ri(x) (i = 1,...,NRT) in an interval (X0,X1).  Only roots
C of odd multiplicity (i.e. changes of sign of the Ri) are found.
C Here the sign of X1 - X0 is arbitrary, but is constant for a given
C problem, and -leftmost- means nearest to X0.
C The values of the vector-valued function R(x) = (Ri, i=1...NRT)
C are communicated through the call sequence of DROOTS.
C The method used is the Illinois algorithm.
C
C Reference:
C Kathie L. Hiebert and Lawrence F. Shampine, Implicitly Defined
C Output Points for Solutions of ODEs, Sandia Report SAND80-0180,
C February 1980.
C
C Description of parameters.
C
C NRT    = number of functions Ri, or the number of components of
C          the vector valued function R(x).  Input only.
C
C HMIN   = resolution parameter in X.  Input only.  When a root is
C          found, it is located only to within an error of HMIN in X.
C          Typically, HMIN should be set to something on the order of
C               100 * UROUND * MAX(ABS(X0),ABS(X1)),
C          where UROUND is the unit roundoff of the machine.
C
C JFLAG  = integer flag for input and output communication.
C
C          On input, set JFLAG = 0 on the first call for the problem,
C          and leave it unchanged until the problem is completed.
C          (The problem is completed when JFLAG .ge. 2 on return.)
C
C          On output, JFLAG has the following values and meanings:
C          JFLAG = 1 means DROOTS needs a value of R(x).  Set RX = R(X)
C                    and call DROOTS again.
C          JFLAG = 2 means a root has been found.  The root is
C                    at X, and RX contains R(X).  (Actually, X is the
C                    rightmost approximation to the root on an interval
C                    (X0,X1) of size HMIN or less.)
C          JFLAG = 3 means X = X1 is a root, with one or more of the Ri
```

```
C                    being zero at X1 and no sign changes in (X0,X1).
C                    RX contains R(X) on output.
C          JFLAG = 4 means no roots (of odd multiplicity) were
C                    found in (X0,X1) (no sign changes).
C
C X0,X1  = endpoints of the interval where roots are sought.
C          X1 and X0 are input when JFLAG = 0 (first call), and
C          must be left unchanged between calls until the problem is
C          completed.  X0 and X1 must be distinct, but X1 - X0 may be
C          of either sign.  However, the notion of -left- and -right-
C          will be used to mean nearer to X0 or X1, respectively.
C          When JFLAG .ge. 2 on return, X0 and X1 are output, and
C          are the endpoints of the relevant interval.
C
C R0,R1  = arrays of length NRT containing the vectors R(X0) and R(X1),
C          respectively.  When JFLAG = 0, R0 and R1 are input and
C          none of the R0(i) should be zero.
C          When JFLAG .ge. 2 on return, R0 and R1 are output.
C
C RX     = array of length NRT containing R(X).  RX is input
C          when JFLAG = 1, and output when JFLAG .ge. 2.
C
C X      = independent variable value.  Output only.
C          When JFLAG = 1 on output, X is the point at which R(x)
C          is to be evaluated and loaded into RX.
C          When JFLAG = 2 or 3, X is the root.
C          When JFLAG = 4, X is the right endpoint of the interval, X1.
C
C JROOT  = integer array of length NRT.  Output only.
C          When JFLAG = 2 or 3, JROOT indicates which components
C          of R(x) have a root at X, and the direction of the sign
C          change across the root in the direction of integration.
C          JROOT(i) =  1 if Ri has a root and changes from - to +.
C          JROOT(i) = -1 if Ri has a root and changes from + to -.
C          Otherwise JROOT(i) = 0.
C-----------------------------------------------------------------------
      INTEGER I, IMAX, IMXOLD, LAST, NXLAST,ISTUCK,IUNSTUCK
      DOUBLE PRECISION ALPHA, T2, TMAX, X2, ZERO,FRACINT,FRACSUB,TENTH
     $     ,HALF,FIVE
      LOGICAL ZROOT, SGNCHG, XROOT
      SAVE ALPHA, X2, IMAX, LAST
      DATA ZERO/0.0D0/, TENTH/0.1D0/, HALF/0.5D0/, FIVE/5.0D0/

      IF (JFLAG .EQ. 1) GO TO 200
C JFLAG .ne. 1.  Check for change in sign of R or zero at X1. ----------
      IMAX = 0
      ISTUCK=0
      IUNSTUCK=0
      TMAX = ZERO
      ZROOT = .FALSE.
      DO 120 I = 1,NRT
         if ((JROOT(I) .eq. 1).AND.((ABS(R1(I)) .GT. ZERO))) IUNSTUCK=I
         IF (ABS(R1(I)) .GT. ZERO) GO TO 110
         if (JROOT(I) .eq. 1) GOTO 120
         ISTUCK=I
         GO TO 120
C     At this point, R0(i) has been checked and cannot be zero. ------------
 110     IF (SIGN(1.0D0,R0(I)) .EQ. SIGN(1.0D0,R1(I))) GO TO 120
         T2 = ABS(R1(I)/(R1(I)-R0(I)))
         IF (T2 .LE. TMAX) GO TO 120
         TMAX = T2
         IMAX = I
 120  CONTINUE
      IF (IMAX .GT. 0) GO TO 130
      IMAX=ISTUCK
      IF (IMAX .GT. 0) GO TO 130
      IMAX=IUNSTUCK
      IF (IMAX .GT. 0) GO TO 130

      SGNCHG = .FALSE.
      GO TO 140
 130  SGNCHG = .TRUE.
 140  IF (.NOT. SGNCHG) GO TO 420
C There is a sign change.  Find the first root in the interval. --------
      XROOT = .FALSE.
      NXLAST = 0
      LAST = 1
C
C Repeat until the first root in the interval is found.  Loop point. ---
 150  CONTINUE
      IF (XROOT) GO TO 300
      IF (NXLAST .EQ. LAST) GO TO 160
      ALPHA = 1.0D0
      GO TO 180
 160  IF (LAST .EQ. 0) GO TO 170
      ALPHA = 0.5D0*ALPHA
      GO TO 180
 170  ALPHA = 2.0D0*ALPHA
 180  if((ABS(R0(IMAX)).EQ.ZERO).OR.(ABS(R1(IMAX)).EQ.ZERO)) THEN
         X2=(X0+ALPHA*X1)/(1+ALPHA)
      ELSE
         X2 = X1 - (X1-X0)*R1(IMAX)/(R1(IMAX) - ALPHA*R0(IMAX))
      ENDIF
      IF (ABS(X2 - X0) .LT. HALF*HMIN) THEN
        FRACINT = ABS(X1 - X0)/HMIN
        IF (FRACINT .GT. FIVE) THEN
          FRACSUB = TENTH
        ELSE
          FRACSUB = HALF/FRACINT
        ENDIF
        X2 = X0 + FRACSUB*(X1 - X0)
      ENDIF

      IF (ABS(X1 - X2) .LT. HALF*HMIN) THEN
        FRACINT = ABS(X1 - X0)/HMIN
        IF (FRACINT .GT. FIVE) THEN
```

```
          FRACSUB = TENTH
        ELSE
          FRACSUB = HALF/FRACINT
        ENDIF
        X2 = X1 - FRACSUB*(X1 - X0)
      ENDIF
c     ---------------------- Hindmarsh ----------------
      JFLAG = 1
      X = X2
C     Return to the calling routine to get a value of RX = R(X). ----
      RETURN
C     Check to see in which interval R changes sign. ----------------
 200  IMXOLD = IMAX
      IMAX = 0
      ISTUCK=0
      IUNSTUCK=0
      TMAX = ZERO
      ZROOT = .FALSE.
      DO 220 I = 1,NRT
         if ((JROOT(I).eq. 1).AND.((ABS(RX(I)) .GT. ZERO))) IUNSTUCK=I
         IF (ABS(RX(I)) .GT. ZERO) GO TO 210
         if (JROOT(I) .eq. 1) go to 220
         ISTUCK=I
         GO TO 220
C     Neither R0(i) nor RX(i) can be zero at this point. -------------------
 210     IF (SIGN(1.0D0,R0(I)) .EQ. SIGN(1.0D0,RX(I))) GO TO 220

         T2 = ABS(RX(I)/(RX(I) - R0(I)))
         IF (T2 .LE. TMAX) GO TO 220
         TMAX = T2
         IMAX = I
 220  CONTINUE
      IF (IMAX .GT. 0) GO TO 230
      IMAX=ISTUCK
      IF (IMAX .GT. 0) GO TO 230
      IMAX=IUNSTUCK
      IF (IMAX .GT. 0) GO TO 230
      SGNCHG = .FALSE.
      IMAX = IMXOLD
      GO TO 240
 230  SGNCHG = .TRUE.
 240  NXLAST = LAST
      IF (.NOT. SGNCHG) GO TO 260
C Sign change between X0 and X2, so replace X1 with X2. ----------------
      X1 = X2
      CALL DCOPY (NRT, RX, 1, R1, 1)
      LAST = 1
      XROOT = .FALSE.
      GO TO 270

 260  CONTINUE
      CALL DCOPY (NRT, RX, 1, R0, 1)
      X0 = X2
      LAST = 0
      XROOT = .FALSE.
 270  IF (ABS(X1-X0) .LE. HMIN) XROOT = .TRUE.
      GO TO 150
C
C Return with X1 as the root.  Set JROOT.  Set X = X1 and RX = R1. -----
 300  JFLAG = 2
c     exit with root findings
      X = X1
      CALL DCOPY (NRT, R1, 1, RX, 1)
      RETURN
C No sign changes in this interval.  Set X = X1, return JFLAG = 4. -----
 420  CALL DCOPY (NRT, R1, 1, RX, 1)
      X = X1
      JFLAG = 4
      RETURN
C----------------------- END OF SUBROUTINE DROOTS ----------------------
      END
```

Appendix B

Modelica Model of two Hydraulic components

B.1 A Hydraulic pump

```
class PompeCentrifugeDyn "Pompe centrifuge dynamique"
  parameter Real V=1 "Volume";
  parameter Real VRotn=1400 "Vitesse de rotation de reference";
  parameter Real J=10 "Moment d'inertie de la pompe";
  parameter Real Cf0=10 "Coef. de frottement";
  parameter Real p_rho = 0;
  parameter Real a1=-88.67 "Coef. x^2 de la caracteristique hn = f(Q) (s2/m5)";
  parameter Real a2=0 "Coef. x de la caracteristique hn = f(Q) (s/m2)";
  parameter Real a3=43.15 "Coef. constant de la caracteristique hn = f(Q) (m)";
  parameter Real b1=-3.7751 "Coef. x^2 de la caracteristique rh = f(Q) (s2/m6)";
  parameter Real b2=3.61 "Coef. x de la caracteristique rh = f(Q) (s/m3)";
  parameter Real b3=-0.0075464 "Coef. constant de la caracteristique rh = f(Q) (s.u.)";
  parameter Real mode=1;

  parameter Real rhmin=0.01 "Rendement hydraulique minimum";

  parameter Real g=9.80665 "Accélération de la pesanteur";
  parameter Real pi=3.141592654 "pi";
  parameter Real eps=1.e-6 "Limite inf.";

  Real rh "Rendement hydraulique";
  Real hn(start=10) "Hauteur manométrique";
  Real VRot "Vitesse de rotation";
  Real w(start=146.6) "Vitesse de rotation angulaire";
  Real R "Rapport VRot/VRotn (s.u.)";
  Real Q(start=581.608) "Débit de masse";
  Real Qv(start=0.5) "Débit volumique";
  Real Cm "Couple moteur total";
  Real Cht "Couple hydraulique utile (couple de mise en charge)";
  Real Ch "Couple hydraulique";
  Real Cf "Couple de frottement";
  Real rho "Masse volumique";
  Real deltaP "Mise en charge de la pompe";
  Real deltaH "Différence d'enthalpie spécifique entre la sortie et l'entrée";
  Real Tm(start=290) "Température moyenne";
  Real Pm(start=110984) "Pression moyenne";
  Real Hm(start=70836.4) "Enthalpie spécifique moyenne";
  Real pompe "Fonctionnement en pompe";
  Real region (start=1);
  Real Wh;
  Real Wf;
  PortCW M;
  PortPHQ1 C1;
  PortPHQ2 C2;
equation
  deltaP = C2.P - C1.P;
  deltaH = C2.H - C1.H;
  C1.Q = C2.Q;
  Q = C1.Q;
  Q = Qv*rho;
  0 = if (Q > 0) then C1.H - C1.Hm else C2.H - C2.Hm;
  VRot = (30/pi)*w;
  R = VRot/VRotn;

  /* Bilan d'énergie */
   V*rho*der(Hm) = C1.Q*C1.H - C2.Q*C2.H + Wh + Wf;
   Wh = Ch*w;
   Wf = Cf*w;

 /* "Hauteur manométrique" */
  hn = (a1*Qv*abs(Qv) + a2*Qv*R + a3*R*abs(R));
  deltaP = rho*g*hn;

  /* Mise en rotation de la pompe */
  Cm = M.Ctr;
  w = M.w;
  Cm = Cf + Ch + J*der(w);
  Cf = (if (abs(R)< 1) then R/(abs(R) + 1e-5)*Cf0*(1 - abs(R)) else 0);
  rh =  (if (abs(R)> eps)then max(b1*Qv^2/R^2 + b2*Qv/R + b3,rhmin) else rhmin);/*Rnd hyd*/
  Cht=  (if abs(w) > eps then Qv*deltaP/w else  Qv*deltaP/eps); /* couple hydrolique total*/
  Ch = (if Ch*w >= 0 then Cht/rh else rh*Cht);
  pompe =if Ch*w >= 0 then 1.0 else 0.0; /*if Ch*w <= 0  Fonctionnement en turbine */
  Pm = (C1.P + C2.P)/2;
  Hm = (C1.H + C2.H)/2;
  Tm = Eau_PH_T(Pm, Hm, mode);
  region = Eau_PH_region(Pm, Hm, mode);
  rho = if (p_rho > 0) then    p_rho
                       else    Eau_PH_d(Pm, Hm, mode);
end PompeCentrifugeDyn;
```

B.2 A Heat Exchanger

```
class Echg5
  parameter Real ETAMIN=1.0e-9;
  parameter Real lambda=15.0     "Conductivite thermique du metal";
  parameter Real p_hc=6000 "Coefficient  convectif du fluide chaud si non calculé par les corrélations";
  parameter Real p_hf=3000 "Coefficient  convectif du fluide froid si non calculé par les corrélations";
  parameter Real p_Kc=100  "Coefficient d pertes de charges ne nont pas calculées par les corrélations (m-4)";
  parameter Real p_Kf=100  "Coefficient  de charges ne nont pas calculées par les corrélations (m-4)";
  parameter Real Vc=1 "Volume de la branche chaude";
  parameter Real Vf=1 "Volume de la branche froide";
  parameter Real emetal=0.0006 "Epaisseur du metal";
  parameter Real Sp=2 "Surface d'une plaque";
  parameter Real nbp=499 "Nombre de plaques (s.u.)";
  parameter Real c1=1.12647 "Facteur de correctif de perte de charge pour les corrélations SRI (s.u.)";
  parameter Real N=5.0 "Nombre de segments";
  parameter Real permanent=1 "Calcul du permanent";
  parameter Real p_rhoc=0 "Si > 0, masse volumique imposée du fluide chaud";
  parameter Real p_rhof=0 "Si > 0, masse volumique imposée du fluide froid";
  parameter Real modec=1  "Région IF97 . 1:liquide - 2:vapeur - 4:saturation - 0:calcul automatique";
  parameter Real modef=1  "Région IF97 . 1:liquide - 2:vapeur - 4:saturation - 0:calcul automatique";
  parameter Real Correlation_Thermique=1 ". d'échange. 0: pas de corrélation. 1: corrélations SRI (Entier)";
  parameter Real Correlation_Hydraulique=1 " charges. 0: pas de corrélation. 1: corrélations SRI (Entier)";

  Real dW[5] "Puissance echangée";
  Real DPc[5] "Perte de charge du fluide chaud";
  Real DPf[5] "Perte de charge du fluide froid";
  Real hc[5]    "Coefficient d'échange convectif du fluide chaud";
  Real hf[5]    "Coefficient d'échange convectif du fluide froid";
  Real K[5] "Coefficient d'échange global";
  Real dS "Surface d'échange";
  Real Tec "Température d'entrée du fluide chaud";
  Real Tsc "Température de sortie du fluide chaud";
  Real Tef "Température d'entrée du fluide froid";
  Real Tsf "Température de sortie du fluide froid";
  Real Pcc[5 + 1]    "Pression du fluide chaud aux frontières de la section i";
  Real Qcc[5 + 1] (start={368.7,368.7,368.7,368.7,368.7,368.7})    "Déb frontières de la section i";
  Real Hcc[5 + 1]  "Enthalpie spécifique du fluide chaud aux frontières de la section i";
  Real Pcf[5 + 1]        "Pression du fluide froid aux frontières de la section i";
  Real Qcf[5 + 1](start={1062.6,1062.6,1062.6,1062.6,1062.6,1062.6})"Débit  frontières de la section i";
  Real Hcf[5 + 1]  "Enthalpie spécifique du fluide froid aux frontières de la section i";
  Real Qc[5](start={368.682,368.682,368.682,368.682,368.682})    "Débit massique du fluide chaud";
  Real Qf[5](start={1062.62,1062.62,1062.62,1062.62,1062.62})    "Débit massique du fluide froid";
  Real qmc[5];
  Real qmf[5];
  Real quc[5];
  Real quf[5];
  Real M;
  Real  rhoc[5](start={998,998,998,998,998})   "Masse volumique du fluide chaud";
  Real  rhof[5](start={998,998,998,998,998})   "Masse volumique du fluide froid";
  Real  muc[5](start={1e-3,1e-3,1e-3,1e-3,1e-3})  "Viscosite dynamique du fluide chaud";
  Real  muf[5](start={1e-3,1e-3,1e-3,1e-3,1e-3})  "Viscosite dynamique du fluide froid";
  Real  lambdac[5](start={0.602698,0.602698,0.602698,0.602698,0.602698}) "Conductivite  thr du fluide chaud";
  Real  lambdaf[5](start={0.597928,0.597928,0.597928,0.597928,0.597928})  "Conductivite  thr du fluide froid";
  Real  Tmc[5](start={290,290,290,290,290})         "Température moyenne du fluide chaud";
  Real  Tmf[5](start={290,290,290,290,290})         "Température moyenne du fluide froid";
  Real  Pmc[5](start={1.e5,1.e5,1.e5,1.e5,1.e5})         "Pression moyenne du fluide chaud";
  Real  Pmf[5](start={1.e5,1.e5,1.e5,1.e5,1.e5})         "Pression moyenne du fluide froid";
  Real  Hmc[5](start={70821,70821,70821,70821,70821}) "Enthalpie spécifique moyenne du fluide chaud";
  Real  Hmf[5](start={50070,50070,50070,50070,50070}) "Enthalpie spécifique moyenne du fluide froid";

  Real regionc[5](start={1,1,1,1,1})         "Numéro de région IF97 du fluide chaud";
  Real regionf[5](start={1,1,1,1,1})         "Numéro de région IF97 du fluide froid";

  Real ac[5], af[5], bc[5], bf[5];
  Real abs_Qc[5], abs_Qf[5], abs_qmc[5], abs_qmf[5];
  Real abs_mucc[5],abs_muff[5], mucc[5], muff[5], maxc[5], maxf[5];

  Real measure;

  PortPHQ1 Ec;
  PortPHQ1 Ef;
  PortPHQ2 Sf;
  PortPHQ2 Sc;

equation

   measure = Qcc[1];
   Ec.P = Pcc[1]; Sc.P = Pcc[6]; Ec.Q = Qcc[1]; Sc.Q = Qcc[6]; Ec.H = Hcc[1];Sc.H = Hcc[6];
   Ef.P = Pcf[6]; Sf.P = Pcf[1]; Ef.Q = Qcf[6]; Sf.Q = Qcf[1]; Ef.H = Hcf[6];Sf.H = Hcf[1];

   0 = if (Ec.Q > 0) then Ec.H - Ec.Hm else Sc.H - Sc.Hm;
   0 = if (Ef.Q > 0) then Ef.H - Ef.Hm else Sf.H - Sf.Hm;

   dS = (nbp - 2)*Sp/N;
   M = (nbp - 1)/2;

   for i in 1:5 loop
     Qcc[i] = Qcc[i + 1];
     Qcf[i] = Qcf[i + 1];

     Qc[i] = Qcc[i];
```

```
Qf[i] = Qcf[i];

Pcc[i + 1] = if (Qc[i] > 0) then Pcc[i] - DPc[i]/N else Pcc[i] + DPc[i]/N;
Pcf[i + 1] = if (Qf[i] > 0) then Pcf[i] + DPf[i]/N else Pcf[i] - DPf[i]/N;

K[i] = hc[i]*hf[i]/(hc[i] + hf[i] + hc[i]*hf[i]*emetal/lambda);
dW[i] = K[i]*dS*(Tmc[i] - Tmf[i]);

Vc/N*rhoc[i]*der(Hmc[i]) = Qcc[i]*Hcc[i] - Qcc[i + 1]*Hcc[i + 1] - dW[i];
Vf/N*rhof[i]*der(Hmf[i]) = -Qcf[i]*Hcf[i] + Qcf[i + 1]*Hcf[i + 1] + dW[i];

abs_Qc[i] = if (Qc[i] > 0) then Qc[i] else -Qc[i];
abs_Qf[i] = if (Qf[i] > 0) then Qf[i] else -Qf[i];
maxc[i]=if (ETAMIN>muc[i]) then  ETAMIN else muc[i];
maxf[i]=if (ETAMIN>muf[i]) then  ETAMIN else muf[i];

qmc[i] = abs_Qc[i]/(maxc[i]* M);
qmf[i] = abs_Qf[i]/(maxf[i]* M);

abs_qmc[i] = if (qmc[i] > 0.0) then qmc[i] else -qmc[i];
abs_qmf[i] = if (qmf[i] > 0.0) then qmf[i] else -qmf[i];
mucc[i]=muc[i]*Eau_PH_cp(Pmc[i], Hmc[i], modec)/lambdac[i];
muff[i]=muf[i]*Eau_PH_cp(Pmf[i], Hmf[i], modef)/lambdaf[i];
abs_mucc[i]= if (mucc[i] > 0.0) then mucc[i] else -mucc[i];
abs_muff[i]= if (muff[i] > 0.0) then muff[i] else -muff[i];

ac[i]= if (qmc[i] < 1.e-3) then 0 else 11.245*abs_qmc[i]^0.8* abs_mucc[i]^0.4*lambdac[i];
af[i]= if (qmf[i] < 1.e-3) then 0 else 11.245*abs_qmf[i]^0.8* abs_muff[i]^0.4*lambdaf[i];

hc[i]= if (Correlation_Thermique < 0.5) then p_hc
       else if (Correlation_Thermique > 0.5) then ac[i]
             else     0.0;
hf[i]= if (Correlation_Thermique < 0.5) then p_hf
       else if (Correlation_Thermique > 0.5) then af[i]
            else     0.0;
quc[i] = abs_Qc[i]/M;
quf[i] = abs_Qf[i]/M;

bc[i]= if (qmc[i] < 1.e-3) then
          0
       else
         c1*14423.2/rhoc[i]*abs_qmc[i]^(-0.097)*quc[i]^2*(1472.47 + 1.54*(M - 1)/2 + 104.97*abs_qmc[i]^(-0.25));

bf[i]= if (qmf[i] < 1.e-3) then
          0
       else
         14423.2/rhof[i]*abs_qmf[i]^(-0.097)*quf[i]^2*(1472.47 + 1.54*(M - 1)/2 + 104.97*abs_qmf[i]^(-0.25));

DPc[i] =if (Correlation_Hydraulique < 0.5) then  p_Kc*Qc[i]^2/(2*rhoc[i])
        else if (Correlation_Hydraulique > 0.5) then bc[i]
             else 0.0;
DPf[i] =if (Correlation_Hydraulique < 0.5) then  p_Kf*Qf[i]^2/(2*rhof[i])
        else if (Correlation_Hydraulique > 0.5) then bf[i]
             else 0.0;
Pmc[i] = (Pcc[i] + Pcc[i + 1])/2;
Pmf[i] = (Pcf[i] + Pcf[i + 1])/2;
Hmc[i] = (Hcc[i] + Hcc[i + 1])/2;
Hmf[i] = (Hcf[i] + Hcf[i + 1])/2;

regionc[i] = Eau_PH_region(Pmc[i], Hmc[i], modec);
regionf[i] = Eau_PH_region(Pmf[i], Hmf[i], modef);

Tmc[i] = Eau_PH_T(Pmc[i], Hmc[i], modec);
Tmf[i] = Eau_PH_T(Pmf[i], Hmf[i], modef);

rhoc[i] = if (p_rhoc > 0) then p_rhoc  else Eau_PH_d(Pmc[i], Hmc[i], modec);
```

```
    rhof[i] = if (p_rhof > 0) then p_rhof  else Eau_PH_d(Pmf[i], Hmf[i], modef);

    muc[i] = ViscositeDynamique_rhoT_(rhoc[i], Tmc[i]);
    muf[i] = ViscositeDynamique_rhoT_(rhof[i], Tmf[i]);

    lambdac[i] = ConductiviteThermique_rhoT_(rhoc[i], Tmc[i], Pmc[i], regionc[i]+0.01);
    lambdaf[i] = ConductiviteThermique_rhoT_(rhof[i], Tmf[i], Pmf[i], regionf[i]+0.01);
  end for;

  Tec = Eau_PH_T(Ec.P, Ec.H, modec);
  Tsc = Eau_PH_T(Sc.P, Sc.H, modec);
  Tef = Eau_PH_T(Ef.P, Ef.H, modef);
  Tsf = Eau_PH_T(Sf.P, Sf.H, modef);
end Echg5;
```

Bibliography

[1] 20-SIM. http://www.20sim.com.

[2] ABACUSS. http://yoric.mit.edu/abacuss2/abacuss2.html.

[3] ALLGOWER, E. L., AND GEORG, K. Numerical continuation methods, an introduction. *Springer Ser. in Comput. Math. Springer-Verlag 13* (1990).

[4] ALLGOWER, E. L., AND K.GEORG. *Continuation and path following.* Acta Numerica., 1993.

[5] AMESIM. www.amesim.com.

[6] AREVALO, C., CAMPBELL, S. L., AND SELVA, M. Unitary partitioning in general constraint preserving dae integrators. *Matehmatical and Computer Modelling 40* (2004), 1273–1284.

[7] ASCHER, U. M., AND PETZOLD, L. R. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*, 1st ed. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1998.

[8] AZIL, A. *Modélisation et Simulation des systèmes Dynamique Hybrides: Formalisme Scicoas et Compilation.* Doctor of science thesis, Universite de Paris XI, Orsay, 2005.

[9] BENVENISTE, A. Compositional and uniform modeling of hybrid systems. *IEEE Transaction on Automatic Control 1*, AC-43 (1998).

[10] BRANICKY, M. S. Studies in hybrid systems: Modeling, analysis, and control.

[11] BRANICKY, M. S. General hybrid dynamical systems: Modeling, analysis, and control. *In R. Alur, T.A. Henzinger, and E.D. Sontag (eds.), Hybrid Systems III: Verification and Control 1066* (1996), 186–200.

[12] BRENAN, K. E., CAMPBELL, S. L., AND PETZOLD, L. R. Numerical solution of initial-value problems in differential-algebraic equations. *SIAM publication, Philadelphia* (1996).

[13] BROWN, P. N., HINDMARSH, A. C., AND PETZOLD, L. R. Consistent initial condition calculation for differential-algebraic systems. *SIAM Journal on Scientific Computing 19*, 5 (1998), 1495–1512.

[14] BUNKS, C., CHANCELIER, J. P., DELEBECQUE, F., GOMEZ, C., GOURSAT, M., NIKOUKHAH, R., AND STEER, S. *Engineering and Scientific Computing with Scilab*, 1st ed. Barkhausen, Le Chesnay, France, 1999.

[15] CALLIER, F. M., AND DESOER, C. A. *Linear System Theory*, 1st ed. Springer-Verlag, 1991.

[16] CAML. http://caml.inria.fr.

[17] CAMPBELL, S. L. Numerical methods for unstructured higher index daes. *Annals of Numerical Mathematics 1* (1994), 265–278.

[18] CAMPBELL, S. L., CHANCELIER, J., AND NIKOUKHAH, R. *Modeling and simulation in Scilab/Scicos*, 1st ed. Springer, 2005.

[19] Campbell, S. L., and Moore, E. Constraint preserving integrators for general nonlinear higher index daes. *Numerische Mathematik 69* (1995), 383–399.

[20] Campbell, S. L., Moore, E., Nakashige, R., Zhong, Y., and Zochling, R. Constraint preserving integrators for unstructured higher index daes. *Zeitschrift fuer Angewandte Mathematik und Mechanik (ZAMM) 76* (1996), 83–86.

[21] Campbell, S. L., and Zhong, Y. Jacobian reuse in explicit integrators for higher index dae. *Appl. Num. Math. 25* (1997), 391–412.

[22] Chancelier, J. P., Delebecque, F., Gomez, C., Goursat, M., Nikoukhah, R., and Steer, S. *An introduction to Scilab*, 1st ed. Springer Verlag, Le Chesnay, France, 2002.

[23] Chi. http://seweb.se.wtb.tue.nl/documentation.

[24] CSP. http://archive.comlab.ox.ac.uk/csp.html.

[25] Djenidi, R. *Formalisme de Modélisation de Systèmes Dynamiques Hybrides.* Doctor of science thesis, Universite de Paris XII, Val de Marne, 2001.

[26] Djenidi, R., Nikoukhah, R., and Steer, S. A propos du formalisme scicos. *MOSIM, Troyes, France* (April 2001).

[27] Dyess, A., Chan, E., Hofmann, H., Horia, W., and Trajkovic, L. Simple implementations of homotopy algorithms for finding dc solutions of nonlinear circuits. *Proc. IEEE Int. Symp. Circuits and Systems* (1999), 290–293.

[28] Dymola. http://www.dynasim.se.

[29] EASY5. http://www.mscsoftware.com.

[30] Eaves, B. C. A short course in solving equations with pl homotopies. In *SIAM-AMS Proceedings, IX* (1976), pp. 73–143.

[31] Fabian, G. *A Language and Simulator for Hybrid Systems.* Doctor of science thesis, Eindhoven University of Technology, 1999.

[32] Folland, G. B. *Introduction to Partial Differential Equations.* Princeton University Press, Princeton,New Jersey, 1995.

[33] Ford, L. R., and Fulkerson, D. R.

[34] Fritzson, P. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, 1st ed. Wiley-IEEE Press, 2004.

[35] Fritzson, P., and Bunus, P. Modelica a general object-oriented language for continuous and discrete-event system modeling and simulation. *Annual Simulation Symposium* (2002), 365–380.

[36] Gear, C. W. The simultaneous numerical solution of differential-algebraic equations. *IEEE Trans. Circuit Theory TC-18* (1971), 89–95.

[37] Gear, C. W. Differential-algebraic equation index transformations. *SIAM. J. Sci. Stat. Comp. 9* (1988), 3947.

[38] Gear, C. W., Leimkuhler, B., and Gupta, G. K. Automatic integration of euler-lagrange equations with constraints. *Comput. Appl. Math. 12-13* (1985), 7790.

[39] Georg, K. An introduction to pl algorithms: Computational solution of nonlinear systems of equations. In *Lectures in Applied Mathematics, American Mathematical Society* (1990), vol. 26, pp. 207–236.

[40] H., C., Lee, E. A., L., J., Liu, X., Neuendorffer, S., Xiong, Y., and (eds.), H. Z. Ptolemy ii: Heterogeneous concurrent modeling and design in java.

[41] Heemels, W. P. M. H., Schumachera, J., and Weiland, S. Linear complementarity systems. *SIAM Journal on Applied Mathematics 60* (2000), 1234–1269.

[42] Hindmarsh, A. C. Lsode and lsodi, two new initial value ordinary differential equation solvers. *ACM-Signum Newsletter 15* (1980), 10–11.

[43] Hindmarsh, A. C. Odepack, a systematized collection of ode solvers. *IMACS Transactions on Scientific Computation 1* (1983), 55–64.

[44] Koo, T. J., Hoffmann, F., Shim, H., Sinopoli, B., and Sastry, S. Hybrid control of model helicopter. In *Proc. of IFAC Workshop on Motion Control, Grenoble, France* (1999), pp. 285–290.

[45] Kroner, A., Marquardt, W., and Gilles, E. D. Computing consistent initial conditions for differential-algebraic equations. *ESCAPE-1 Comput. Chem. Engng, Elsinore, Denmark 16* (1992), S131–S138.

[46] Kunkel, P. A tree-based analysis of a family of augmented systems for the computation of singular points. *IMA J. Numer. Anal. 16* (1996), 501–527.

[47] Kunkel, P., and Mehrmann, V. Index reduction for differential-algebraic equations by minimal extension. *Z. Angew. Math. Mech. 84* (2004), 579–597.

[48] Kunkel, P., Mehrmann, V., Rath, W., and Weickert, J. A new software package for linear differential-algebraic equations. 115–138.

[49] Kunkel, P., Mehrmann, V., and Seufer, I. Genda: A software package for the numerical solution of general nonlinear differential-algebraic equations. *Institut fur Mathematik, TU Berlin 730* (2002), 579–597.

[50] LabVIEW. http://www.ni.com/labview.

[51] Law, A. M., and Kelton, W. D. *Simulation Modeling and Analysis*, 3rd ed. McGraw-Hill, Inc, New York, USA, 2000.

[52] Lawler, E.

[53] Leenaerts, D. M. W., and van Bokhoven, W. M. G. *Piecewise Linear Modelling and Analysis.* Kluwer Academic Publishers, Boston, 1998.

[54] Leimkuhler, B., Petzold, L. R., and Gear, C. W. Approximation methods for the consistent initialization of differential algebraic equations. *SIAM Journal on Numerical Analysis 28* (1991), 205–226.

[55] Lemke, C. E., and Howson, J. Equilibrium points of bimatrix games. *SIAM J. Appl. Math 12* (1964), 413–423.

[56] L.R. Ford, J., and Fulkerson, D.

[57] Mathis, W., Trajkovic, L., Koch, M., and Feldmann, U. Parameter embedding methods for finding dc operating points of transistor circuits. *Proc. NDES '95, Dublin, Ireland* (1995), 147–150.

[58] MATRIXx. http://www.ni.com/matrixx.

[59] Mattsson, S. E., Elmqvist, H., Otter, M., and Olsson, H. Initialization of hybrid differential-algebraic equations in modelica 2.0. *2nd Inter. Modelica Conference 2002, Dynasim AB, Sweden and DLR Oberpfaffenhofen, Germany* (2002), 9–15.

[60] Mosterman, P. J. An overview of hybrid simulation phenomena and their support by simulation packages. *In Vaandrager, F. W. and van Schuppen, J. H., editors, Hybrid Systems: Computation and Control, Second International Workshop, HSCC'99, volume 1569 of Lecture Notes in Computer Science* (1999), 165–177.

[61] Murty, K. G. *Linear Complementarity, Linear and Nonlinear Programming.* Helderman-Verlag, 1988.

[62] Najafi, M., Azil, A., and Nikoukhah, R. Extending scicos from system to component level simulation. *ESMC2004 international conference, Paris, France* (October 2004).

[63] Najafi, M., Nikoukhah, R., and Campbell, S. L. Computation of consistent initial conditions for multi-mode daes: Application to scicos. *IEEE International Symposium on Intelligent Control Computer Aided Control Systems Design, Taipei* (September 2004).

[64] Nikoukhah, R., and Steer, S. Scicos a dynamic system builder and simulator. In *IEEE International Conference on CACSD, Dearborn, Michigan* (1996).

[65] Nikoukhah, R., and Steer, S. Scicos: A dynamic system builder and simulator, user's guide - version 1.0. Tech. Rep. RT-0207, INRIA Technical Report, Le Chesnay, France, June 1997.

[66] Nikoukhah, R., and Steer, S. Conditioning in hybrid system formalism. *FEMSYS, Munich, Germany* (March 2000).

[67] Pantelides, C. C. "the consistent initialization of differential-algebraic systems". *SIAM. J. Sci. Stat. Comput 9(2)* (1988), 213–231.

[68] Papadimitriou, C., and Steiglitz, K.

[69] Petzold, L. R. "a description of dassl: A differential/algebraic system solver. *Proceedings of the 10th IMACS World Congress, Montreal* (1982), 8–13.

[70] Petzold, L. R. A description of dassl : A differential/algebraic system solver. Tech. Rep. SAND82–8637, Sandia National Laboratories, Livermore, CA, 1982.

[71] Petzold, L. R. Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. *SIAM J. Sci. Stat. Comput 4* (1983).

[72] Petzold, L. R. Order results for implicit runge-kutta method applied to differential algebraic systems. *SIAM. J. Numer. Anal. No. 23* (1986), 837852.

[73] Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vettering, W. T. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, Cambridge, 1992.

[74] Ptolemy. http://ptolemy.eecs.berkeley.edu.

[75] Schwarz, P. Physically oriented modeling of heterogeneous systems. In *3rd. IMACS-Symposium of Mathematical Modelling (MATHMOD)* (Wien, March 2000), pp. 309–318.

[76] Scicos. http://www.scicos.org.

[77] Scilab. http://www.scilab.org.

[78] SDX, m., and simulation software. http://www.sdynamix.com.

[79] Seila, A. F. Introduction to simulation. *Proceedings of the 27th conference on Winter simulation, Arlington, Virginia, United States* (1995), 7– 15.

[80] Shampine, L. F., and Gordon, M. K. *Computer solution of ordinary differential equations.* W. H. Freeman & Co., San Francisco, 1975.

[81] Shanno, D. F. Conjugate gradient methods with inexact searches. *Mathematics of Operations Research 3, No.3* (August 1978), 244–256.

[82] Shanno, D. F., and Phua., K. H. Algorithm 500. a variable method subroutine for unconstrained nonlinear minimization. *ACM Trans. on Mathematical Software 6* (1980), 618–622.

[83] SHIFT. http://www.path.berkeley.edu/shift.

[84] Sildex. http://www.tni-world.com/sildex.asp.

[85] SIMCREATOR. http://www.simcreator.com.

[86] SIMULINK. http://www.mathworks.com.

[87] SPICE. www.eecs.berkeley.edu.

[88] STATEFLOW. http://www.mathworks.com.

[89] STEER, S., AND NIKOUKHAH, R. Scicos: a hybrid system formalism. *ESS, Erlangen, Germany* (October 1999).

[90] SYSTEMBUILD. http://www.ni.com/matrixx/systembuild.htm.

[91] TODD, M. J. Plalgo: A fortran implementation of a piecewise-linear homotopy algorithm for solving systems of nonlinear equations. *School of Operations Research and Industrial Engineering, Cornell University* (April, 1985).

[92] WATSON, L. Golobally convergent homotopy algorithm for nonlinear systems of equations. *Nonlinear Dynamics 1* (1990), 143–191.

[93] WATSON, L. T., SOSONKINA, M., MELVILLE, R. C., MORGAN, A. P., AND WALKER, H. F. Algorithm 777: Hompack90: A suite of fortran 90 codes for globally convergent homotopy algorithms. *ACM Trans. Math. Software 23* (1997), 514–549.

[94] WEIS, P., AND LEROY, X. *Le Langage CAML*, 2nd ed. Dunod Press, 1999.

[95] ZEIGLER, B. P. *Theory of Modeling and Simulation.* John Wiley & Sons, New York, 1984.

[96] ZEIGLER, B. P., PRAEHOFER, H., AND KIM, T. G. *Theory of Modeling and Simulation*, 2nd ed. Academic Press, San Diego, 2000.