ROBOTIC MODELS OF NEUROMECHANICAL STEP GENERATION IN INSECTS

by

BRANDON LEWIS RUTTER

Submitted in partial fulfillment of the requirements

For the degree of Doctor of Philosophy

Dissertation Adviser: Dr. Roger Quinn

Department of Mechanical and Aerospace Engineering

CASE WESTERN RESERVE UNIVERSITY

May, 2010

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis/dissertation of

Brandon Lewis Rutter

candidate for the **Doctor of Philosophy** degree *.

(signed) **Roger D. Quinn**
(chair of the committee)

Roy E. Ritzmann

Kiju Lee

Mark Willis

Robert Kirsch

(date) **July 24, 2009**

*We also certify that written approval has been obtained for any proprietary material contained therein.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I acknowledge my fallability.

Also, everybody in the broadly defined Biorobotics group for the past five years at least. The collegial and social support of these people is outstanding, heavy use of sarcasm notwithstanding. In particular, my primary advisor Dr. Roger Quinn, biological advisor Dr. Roy Ritzmann, close colleague and friend Brian Taylor, Marcus Blümel, Laiyong Mu, William Lewinger, Dr. John Bender and Nicole Doorly have all provided substantial material contributions to this work, and those underlined deserve a cookie and/or shot of whiskey for being exposed to the entire first draft.

Doctors Roger Quinn and Roy Ritzmann both deserve additional thanks for the support, advice and guidance provided in the development both of myself and this work through the past several years. They really care, both about the teaching and the science, and it shows.

I would like to thank Dr. Mark Willis, Dr. Robert Kirsch, and Dr. Kiju Lee for providing valuable revision suggestions and putting up with my less-than-perfect scheduling as members of my dissertation committee.

I thank Ansgar Büschges and his entire lab at the Institute for Animal Physiology at the University of Cologne for training, intellectual access, and outstanding hospitality. This is also where Marcus Blümel actually works.

I would additionally like to express gratitude to: my family for unconditional support. The faculty and staff of the Department of Mechanical and Aerospace Engineering, who

do an excellent job of shielding students from the general insanity of university administration. The organizers of the NSF Integrative Graduate Education and Research Training grant in Neuromechanics at Case Western Reserve, who developed a truly outstanding incubator for novel interdisciplinary research.

And, of course, the financial support provided by the National Science Foundation for that program through award DGE 9972747, the Case Prime fellowship, Elgin Air Force Base grant F08630-03-1-0003, and particularly the U.S. Air Force Office of Science and Research grant FA9550-07-1-0149, have made it feasible to do all of this very interesting work.

Many of those mentioned above, and more to whom I apologize for the omission, have made substantial contributions to the content and clarity of this document. None of them can be held responsible, however, for the inevitable errors it contains. Those are all mine.

List of Abbreviations

CPG central pattern generator, *CTr* coxa-trochanter, *DH* Denavit-Hartenberg, *DOF* degree of freedom, *EMG* electromyogram, *FCO* femoral chordotonal organ, *FTi* femur-tibia, *GUI* graphical user interface, *HDSO* handy data storage object, *MN* motor neuron, *PG* pattern generator, *RT* real-time, *SCASM* Sensory Coupled Action Switching Modules, *SCCILS* SCASM Command and Configuration Interface Language Specification, *T1* prothoracic, *T2* mesothoracic, *T3* metathoracic, *ThC* thorax-coxa, *ThC1* thorax-coxa first degree of freedom, *TrF* trochanter-femur, *UI* user interface, *XML* extensible markup language

Robotic Models of Neuromechanical Step Generation in Insects

Abstract

by

BRANDON LEWIS RUTTER

Walking is a means of locomotion that is ubiquitous among terrestrial animals and the matter of considerable technical inquiry; both for biological understanding and description, and engineering construction and control. Although wheels and treads have numerous advantages over legs for low-complexity terrain, the promise of adept legged locomotion in a much broader range of rugged environments is eloquently demonstrated in the animal kingdom. Of primary interest in the understanding of such agility is the ability of animals to smoothly transition between behaviors requiring substantially different local behavior of locomotor appendages.

Recent developments in our understanding of insect walking systems, encapsulated in the study of the neural mechanisms of stick insect leg coordination by (Ekeberg, Blümel, & Büschges, 2004), have made it possible to construct models of local leg control based on known properties of biological systems. Such models can provide the appropriate "ports" to investigate and predict the effects of descending commands in the transition between and generation of different local behaviors.

This dissertation describes the development and use of robotic models of step generation to address questions about descending control. Robotic models were desired both for the ease of experimental interaction and the fidelity of physical modeling they can provide. The NeuRoMod software suite was developed, and provides interactive operation and experimental scripting for the robotic models.

The local control methods of the stick insect described by Ekeberg et al. are standardized as Sensory Coupled Action Switching Modules (SCASM), and tools for the use of this concept in modeling are developed and demonstrated. The apparent generality of SCASM as a computationally simple control concept is also addressed.

Experiments were conducted to demonstrate model usage, in the testing and generation of biologically relevant hypotheses. The basic function, neuromechanical nature, and resilience of SCASM-controlled steppers is demonstrated. Simple muscle models are found to provide significantly improved, reliable stepping. Control methods for forward walking and inside turning in the cockroach model are presented, and experiments are conducted investigating how descending influences can cause transitions between these behaviors. The results support the general reflex cascade hypothesis presented by (Mu & Ritzmann, 2008a).

1 Introduction

I started my graduate study with a keen interest in working on the basic problems of how to get around in the world. In my undergraduate experience I had spent time working with a few of the standard robots used for AI research, most of which at that time looked like a garbage can on wheels (Maxwell, Meeden, Addo, et al., 2001). A glorified garbage can on wheels, sure- but still completely incapable of taking the stairs, "navigating a curb", or in most cases just plain going outside.

I think I found this lack offensive for a couple of reasons. Quite possibly my own farm upbringing and affinity for the outdoors had something to do with it. More to the point for my academic goals, I'd been interested in exploration robotics for quite some time, mostly related to my interest in space exploration. Though we'd had some good examples of exploration robots used in an essentially tele-operated fashion such as Sojourner (Shirley & Matijevic, 1995), the ideal explorer needs to be able to survive on its own. The broader issue of survival has many facets, of course, including appropriate response to things and events both "good" and "bad" for the agent, but at that point I had come to believe that the "low-level" task of just physically getting from one place to another along a known route was among the most severe limitations on exploratory robots. So I found myself with the choice of going to Carnegie Mellon University and working on some of the most advanced, mostly wheeled, field-operating exploratory robots then available, or coming here to the Biologically Inspired Robotics laboratory at Case Western Reserve to work on basic problems in rugged locomotion; something really interesting.

It looks like I chose well. The purpose of my doctoral work has been to develop a neruomechanically-based robotic model system capable of behavioral changes based on descending commands, and to demonstrate its use in the testing and generation of biologically relevant hypotheses. The testing of the reflex cascade hypothesis (Mu & Ritzmann, 2008a) regarding mechanisms of descending control was a specific aim. The influence of these descending commands is the point between the most basic parts of walking and the generation of the higher-level behaviors that make legs really useful.

This dissertation describes the development and use of robotic models of step generation in stick insects and cockroaches to address questions about descending control. Robotic models were desired both for the ease of experimental interaction and the fidelity of physical modeling they can provide. The NeuRoMod suite of real-time model control software was developed, and provides highly configurable interactive operation and experimental scripting capabilities for the robotic models.

The control methods of the stick insect described by Ekeberg et al. are standardized as Sensory Coupled Action Switching Modules (SCASM), and a set of tools for the use of this concept in modeling are developed and demonstrated. The apparent generality of SCASM as a computationally simple control concept is also addressed. The SCASM Command and Configuration Interface Language Specification (SCCILS) for controlling and describing SCASM systems is developed and presented. SCCILS provides a means for accessing the ports for descending commands in a SCASM system, and is used in the modeling software.

A functional kinematic description of the cockroach middle (T2) leg was developed, in order to support accurate modeling of the thorax-coxa (ThC) joint in that animal. This

new description is fostering a better understanding of leg function in other locomotion studies as well (Bender, Simpson, & Ritzmann, 2009 (In Prep); Mu, 2007).

A series of experiments was conducted in order to demonstrate model function and usage, for both the testing and generation of biologically relevant hypotheses. The basic function, neuromechanical nature, and resilience of SCASM-controlled stepping systems are first demonstrated. The impact of simple muscle modeling is investigated, and found to provide significantly improved, reliable stepping. The control of stepping speed using tonic descending commands was found to be sufficient for generating a range of speeds in both model legs. Control methods for forward walking and two types of inside turning in the cockroach model are presented, and a series of experiments is conducted to investigate various aspects of transitions between these behaviors as commanded by descending influences. The results support the general reflex cascade hypothesis presented by (Mu & Ritzmann, 2008a).

The primary goal of this work has been modeling of the biological systems involved, but as a roboticist the work was always done with an eye towards application in legged locomotion. The possible application of these lessons to an even broader understanding of environmentally appropriate system behavior has presented itself as well.

1.1 The Investigation of Walking

Take a hike. If you want maximum agility in the broadest range of terrestrial settings, you most likely need to use legs. For low-complexity and continuous terrain, wheels and treads are likely to be faster, more efficient, more robust and simpler to implement than legged systems. But, there is a broad range of terrains and situations where, if you

wanted to do something or get somewhere, you would have to get off the bicycle, or get out of the car, or even leave the treaded tank, and use your legs.

Though this line of thought has been clear for some time, even today our walking *machines* are generally much more limited in their capabilities than the available wheeled and tracked vehicles. This was even more obviously the case in the earlier years of this decade; we simply did not understand walking well enough to build something even remotely capable of the adept legged locomotion demonstrated in the animal kingdom. There has been considerable improvement (see BigDog: Raibert, Blankespoor, Nelson, et al., 2008), but the biological counterpart of a robot using biologically based locomotion methods can still literally run, climb, swim or fly circles around it.

During my time in high school and college, my thoughts had turned more than once to biological systems for help in solving problems of dealing intelligently with the world. I'd talk with friends about how the brain of the fly I was trying (and often failing) to swat couldn't possibly have more raw computational power than the computers of the day, yet it was far more capable of behaving quickly and appropriately in the world than anything attempting to do so under computer control. The fly must be put together really well for performing its particular tasks. Years later, I discovered that Rodney Brooks, at least, was having very similar thoughts at a slightly earlier time (Waibel, Brooks, Hauert, et al., 2007).

My naïveté regarding the fly's computational power is now somewhat mitigated; the insect nervous system is capable of some rather complex computation at a preposterously high bandwidth. Nevertheless, the idea that a fly is very well constituted for doing the things a fly does is apparently still a good one. And indeed, in many cases we can find

examples of demanding tasks being accomplished by biological systems which are constructed *just so*. So I've come to believe even more strongly that if we want to build machines that are adept at exploration and survival, it will serve us well to consult the playbook of biology. Life has, after all, been constructed and selected by the demands of survival.

To the blissfully ignorant, "consulting the playbook of biology" might sound like a clever and straightforward thing to do. As it turns out, though, reverse-engineering the *full* function of something as "simple" as a single neuron is the matter of man-millennia of effort; a task that is far from complete. Additionally, upon attempts to implement biological knowledge in engineering systems one often finds critically important holes in the understanding of what is sometimes considered a well-understood biological system. To the initiated, then, it will come as no surprise that I found myself working largely on the problem of consulting this playbook; towards the application of engineering tools and methods to the greater understanding of walking in biological systems. Through this experience I have tried more and more to think about these systems in terms of the flow of information, action and function, rather than the firing of neurons or electrical currents in servomotors which carry these flows in any particular instantiation. In this respect, I have certainly become a cyberneticist, at least in the older sense of the term (Wiener, 1948).

In addressing questions such as these, there is the potential to develop a relationship between biology & engineering which benefits both; it is my hope that this work is an example of the mutual benefits possible. The engineer of legged systems and the biological modeler have similar motivations. The primary question for the engineer is:

how do we best make a machine walk? We want suitably agile leg control with minimal computational requirements. For the biological modeler, the primary question is: how do animals walk, and how do we find and approximate the animal's neural and mechanical systems to get a model that walks that way? We want a system that can test and generate biologically relevant hypotheses. Ideally the model is an investigative and pedagogical tool, useful in experimentation and system description. These desires are similar, and can often be simultaneously addressed by the same effort.

Merely the process of "consulting the playbook of biology" can result in the unveiling of questions critical to system function which were not previously a matter of investigation. The joint kinematics work in Chapter 5 is an example of this kind of interaction. Engineering techniques can also be used for investigations and modeling "purely for biology's sake", as was done for the models developed in this dissertation. Other examples of this may be found in (Webb, 2001) for instance, and are described below in section 3.7. Particularly in these cases, however, it can become difficult to separate the parts where one field is solely benefitting the other, if the synergy of investigation is strong enough. Such cases are probably the most interesting, since they represent the discovery of knowledge which may be new to all of the fields involved.

1.2 Goals of the Broader Project

There is a focus of investigation in the Ritzmann and Quinn laboratories on the effects of descending commands on the lower-level control of basic walking behavior. Within this context, we wanted a physically manipulable insect leg model, with local neural and mechanical systems based on known biological properties, capable of transitions between behaviors such as walking and turning while allowing examination of the progression of

sensory and motor events. Robotic models were desired both for the ease of experimental interaction and the fidelity of physical modeling they can provide.

The short-term specific aim was to use the system to test, find, and refine hypotheses about how the system transitions between behaviors. Specifically, a test of the "reflex cascade hypothesis" (Mu & Ritzmann, 2008a) described in section 0 was desired. A model with this neural basis presents specific ports for descending activity. We can establish where the descending commands would have a described effect/change in behavior. It also allows us the possibility to counteract or mimic some of these changes via physical manipulation; something that is much more difficult in the animal. With sufficient accuracy of the neural basis, such a model can act through intentional tinkering or serendipity as a hypothesis generator, as well as a tester of hypothesis feasibility.

A model capable of doing these things gives us the ability to do experiments and get data probably not possible in the animal; full state information allows us to read sensor and motor changes in the evolution of the system from one state to another. We could then see the interaction between events and subsystems during behavior. This could allow the observation of a particular reflex or sensory influence within the context of behavior, as well as the observation of interplay between such system elements, some of which is extremely difficult or even impossible to observe either in the behaving animal or in models of subsystems. For example, simultaneously reading the state of the central nervous system and complete sensory information is very useful in the description of the progression of leg behavior; subsets of these data are available in biological experimentation but are rarely, if ever, able to completely describe the state of the animal.

As a tool for guiding biological experimentation, the full state information of the model might be used to identify the most easily measurable results of some change in the controller. Using the cockroach model in the more standard capacity as a hypothesis sufficiency tester, in which the complex interactions of known subsystems is investigated as was done in (Ekeberg, et al., 2004), is also a definite capability. It is sometimes possible to predict these interactions analytically or mathematically, but it is very difficult to convincingly demonstrate physical sufficiency without building a system that interacts with the real world and is therefore forced to deal with some of the same physical environmental interactions as the target system.

The ability for experimental manipulation in the model could be beyond anything possible in the animal. An appropriately constructed model could allow the strengthening, weakening, reversal, disconnection and reconnection of any single modeled neural signal or set thereof, at any time, without disturbing other neural correlates or the physical system.

The roboticists in our group also wanted to develop the ideas from the work of (Ekeberg, et al., 2004) into a more general understanding of how systems such as these generate coordinated behavior. Compared to the then-standard methods of generating walking movement in robots, this had the promise of getting the job done with vastly lower computational and informational demands. As described above, these desires are not at odds with one another; one could argue that they are best served when done together.

1.3 The state of the art in 2005

This section describes the tools and knowledge available at the beginning of the work in this dissertation. This is covered in more detail in the background section.

Then as now, the stick insect and cockroach were among the most thoroughly analyzed walking systems. The stick insect was studied most in terms of local sensorimotor pathways and behavior (Cruse, 1990; Ekeberg, et al., 2004), while the cockroach was among the most analyzed in terms of mechanical dynamic properties(Jindrich & Full, 1999), with significant knowledge of mechanosensory systems (Zill, Schmitz, & Büschges, 2004), and agile transitional behaviors (Watson, Ritzmann, Zill, et al., 2002). The existence of local joint-level central pattern generators was known in the stick insect, as measured through motor neurons (Büschges, Ludwar, Bucher, et al., 2004).

Here at Case Western Reserve in the Ritzmann lab, experiments were beginning to address the question: how do higher-level influences interact with local control to generate the range of behaviors? Specifically, what differs between and what initiates transition between walking and turning? The specific descriptions of mechanisms that might be involved in transitions between walking and turning now published in (Mu & Ritzmann, 2005, 2008a, 2008b) were then being investigated. That work involved the identification of effects of descending influences on local magnitude reflex reversal, and the presentation of the "reflex cascade hypothesis" as a method by which a few descending influences might dramatically alter leg behavior. The general idea of this hypothesis is that descending commands might alter a few critical local reflexes that start a cascade of physical changes in leg movement or posture, leading to further alterations

in the overall behavior of the leg. Further description of this is included in the background; see section 0.

Robotic models of biological walking systems and robots using biological inspiration were at this point mostly behaviorally based. There were a few which used neurally-inspired control mechanisms to generate this behavior, e.g.(Dürr, Krause, Schmitz, et al., 2003). As described in (Webb, 2001), the behavioral approach has drawbacks in terms of usefulness to biology. Examining a black box doesn't generally help understanding of the mechanism, or *how* the behavior is produced. This understanding of the mechanism contains the principles of design and control that the engineer would really like to know.

The work which most directly instigated the development of the models in this dissertation was the synthesis of (Ekeberg, et al., 2004). In the middle leg of the stick insect *Carausius morosus*, a number of neural pathways were known which defined sensory coordinating influences capable of switching the activity in the joint central pattern generators (CPGs). During stick insect stepping behavior, these CPGs behave as bi-stable circuits, specifying either flexion or extension in the associated joint. Central coupling, if present, is weak enough to be ignored under these circumstances. It was hypothesized that these influences were sufficient to generate stepping. They constructed a set of state machines whose transition rules implemented these influences, and used that controller in a dynamic computational model of the stick insect middle leg. The resulting model generated stepping motion, thus successfully demonstrating the sufficiency of these combined mechanisms to explain the basic overall behavior of forward stepping in the biological leg. They modified the controller to successfully generate stepping in models of the front and rear legs as well. Since the controller was explicitly and

structurally derived from known neural pathways, the changes for front and back legs represented specific hypotheses regarding the neural function of the animal in those legs. Their work and that supporting it are covered in more detail in section 3.4.1 of the background.

1.4 Project Goals

The purpose of this work is to develop a neruomechanically-based robotic model system capable of behavioral changes based on descending commands, and to demonstrate its use in the testing and generation of biologically relevant hypotheses. There are several more specific goals that were involved in the pursuit of this overall purpose:

1) Implementation of the control methods used in the model of (Ekeberg, et al., 2004) in a robotic model system. This includes the development of the hardware and software necessary. One goal was to make the physical robots small, both for portability and to reduce problems with dynamic scaling. Extensibility was a goal for the software, in order to allow the addition of elements necessary for this and later work.

2) Make it useable by and useful to biologists. This includes developing methods for using these models in experimentation, and understanding their function (see section 2.1). Interactive modification of the controller was desired, as well as experimental scripting capability. These requirements dictate the development of an at least minimally useful user interface.

3) Test and generate biological hypotheses, specifically including the reflex cascade hypothesis of (Mu & Ritzmann, 2008a), by examining the transitions between forward and turning behaviors. This includes the generation of these behaviors in the first place, which embody their own sets of hypotheses. We ultimately wanted to clarify the resulting hypotheses to arrive at feasibly testable biological hypotheses. We felt that a cockroach model would best serve this goal.

1.5 Description of Contents

The rest of this document addresses the pursuit and achievement of the goals above. Chapter 2 is a brief look at essential concepts in robotic model development. A thorough understanding of the material in that chapter will be most useful to those either building models or wishing to use them to their fullest extent. A background focused on the walking task and robotic modeling and control methods is presented in Chapter 3.

In Chapter 4, the control methods of the stick insect described by (Ekeberg, et al., 2004) are standardized as Sensory Coupled Action Switching Modules (SCASM), and a set of methods for the use of this concept in modeling are developed and demonstrated. The apparent generality of SCASM as a computationally simple control concept is also addressed. The SCASM Command and Configuration Interface Language Specification (SCCILS) for controlling and describing SCASM systems is developed and presented. This language provides a means for accessing the ports for descending commands in a SCASM system.

Chapter 6 describes the robotic models, including The NeuRoMod suite of real-time model control software, which provides highly configurable interactive operation and experimental scripting capabilities for these models, using SCASM and SCCILS. Chapter

5 presents a functional kinematic description of the cockroach T2 leg, developed in order to accurately model the ThC joint. This new description is fostering a better understanding of leg function in other biological work.

In Chapter 7, a series of experiments is described in order to demonstrate model function and usage, for both the testing and generation of biologically relevant hypotheses. The basic function, neuromechanical nature, and resilience of SCASM-controlled steppers are first demonstrated in the robotic stick insect model. The impact of simple muscle modeling is investigated, and found to provide significantly improved stepping. A hypothetical method for the control of stepping speed is shown to be sufficient. Control methods for forward walking and two types of inside turning in the cockroach model are presented, and a series of experiments is conducted to investigate various aspects of transitions between these behaviors as commanded by descending modification of a few critical local sensory influences. The results support the general reflex cascade hypothesis presented by (Mu & Ritzmann, 2008a).

This is followed by a summary of formal conclusions and a discussion of future work in Chapter 8. The future work section provides a more complete, informal treatment of where further investigation is recommended and expected to proceed from the current realization of the broader project goals.

2 Essentials in Robotic Model Development

Since the primary goal of my work was the development of models useful to biologists, it was essential to examine how such models can be used, and what helps make them useful. This chapter addresses these issues in the manner of (Webb, 2001), and sets specific goals for model development.

2.1 Elements of a Useful Model

As discussed in detail by (Webb, 2001), there is some confusion and even more disagreement about the definition of the term "model". There are a number of common threads presented there, however, and I will use her work to frame the goals and work described here.

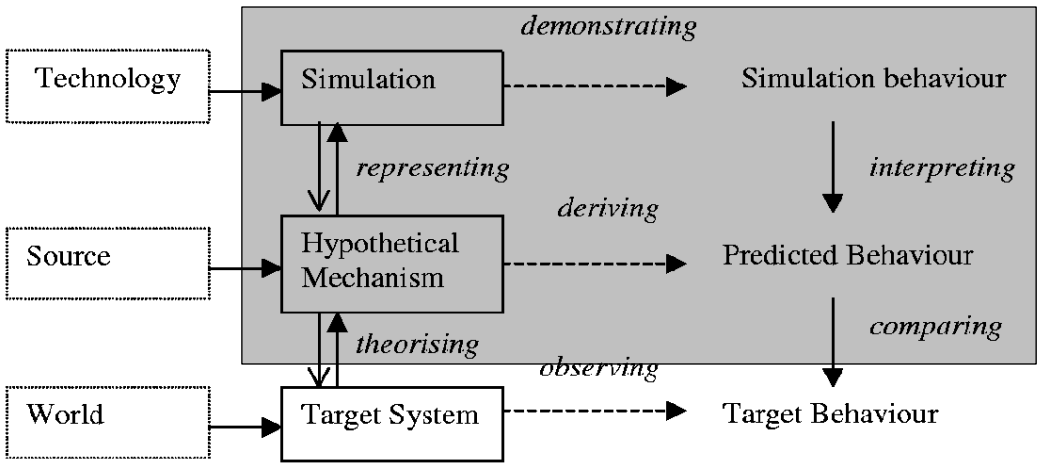


Figure 2-1 "Models and the process of explanation", Figure 1 from (Webb, 2001). There, "model" refers to the "simulation" block above; I attempt to use the same convention. This figure more or less represents the usage of a simulation model to test hypothesis sufficiency in reproducing behavior. The grey box includes many of the elements necessary to actually make use of the simulation, several of which were explicitly addressed in this work along with the development of the simulation model itself.

In order for a model to be useful, i.e. to help the process of explanation, it needs to be appropriately connected to the other elements of explanation in Figure 2-1. Webb's description of this figure includes examples of these elements and is included below:

> [Figure 2-1] can be regarded as an elaboration of standard textbook illustrations of either the "hypothetico-deductive" approach or the "semantic" approach to science (see below). To make each part of the diagram clear, consider an example. Our target –selected from the world–might be the human cochlea and the human behaviour of pitch perception. Our hypothesis might be that particular physical properties of the basilar membrane enable differently positioned hair cells to respond to different sound frequencies. One source of this idea may be the Fourier transform, and associated notion of a bank of frequency filters as a way of processing sound. To see what is predicted by the physical properties of the basal membrane we might build a symbolic simulation of the physical properties we think perform the function, and run it using computer technology, with different simulated sounds to see if it produces the same output frequencies as the cochlea (in fact [(Von Békésy, 1960)] first investigated this problem using rubber as the technology to represent the basilar membrane). We could interpret the dominant output frequency value as a "pitch percept" and compare it to human pitch perception for the same waveforms: insofar as it fails to match we might conclude our hypothesis is not sufficient to explain human pitch perception. Or, as [(Chan & Tidwell, 1993)] concisely summarise this process, we theorise that a system is of type T, and construct an analogous system to T, to see if it behaves like the target system.

It should be noted that the expression of a "Hypothetical Mechanism" might also be evaluated in several of the same dimensions that Webb describes models, as described in section 3.7 of the background; e.g. level, generality, abstraction, and perhaps structural accuracy.

Construction of a model, then, in order to be useful, needs to include the process of ensuring that tools supporting the processes represented by the labeled arrows in Figure 2-1 are present. Ideally, they should also be clear and correct. With this in mind, the attempt has been made to include the following elements when considering model usage and construction. Building each of these elements separately can be highly instructive, but together they can more effectively facilitate the process of explanation.

1) The simulation model itself; in this work, the physical robots and control programs. Building and refining of this element is covered in Chapter 6 "Neuromechanical Robotic Models" as well as Chapter 5 "Functional Cockroach Leg Kinematics".

2) Tools for model operation; supporting ways to *demonstrate* and generate simulated behavior. This includes parameter modification, user interface, experiment scripting and data recording. The development of this element consists primarily of the development of the SCASM Command and Configuration Interface Language Specification (SCCILS) described in 4.3, support for SCCILS in the NeuRoMod engine described in Chapter 6, and the user interface written by John Bender and described in the model operation manual in Appendix 10.4.

3) Tools for data analysis and representation; facilitating the *interpreting, comparing* and possibly even *observing* processes. Some of these are documented in the Experiments chapter (7), as well in the description of the MATLAB SCCILStoolbox package in Appendix 10.3.

4) Tools for system representation; facilitating the *theorizing* and *representing* processes, and possibly also the representation of the "Hypothetical Mechanism".

This is addressed in the development of Sensory Coupled Action Switching Modules (SCASM) as a concept, in Chapter 4.

In the building of some models, it is possible that data and system representation methods are already sufficient. This is sometimes not the case, however, because either (a) data are available in the model that has not been available from the target (i.e., full state information), or (b) there are a lot more data available from the model, and methods developed for analyzing and representing the target cannot cope with this data volume.

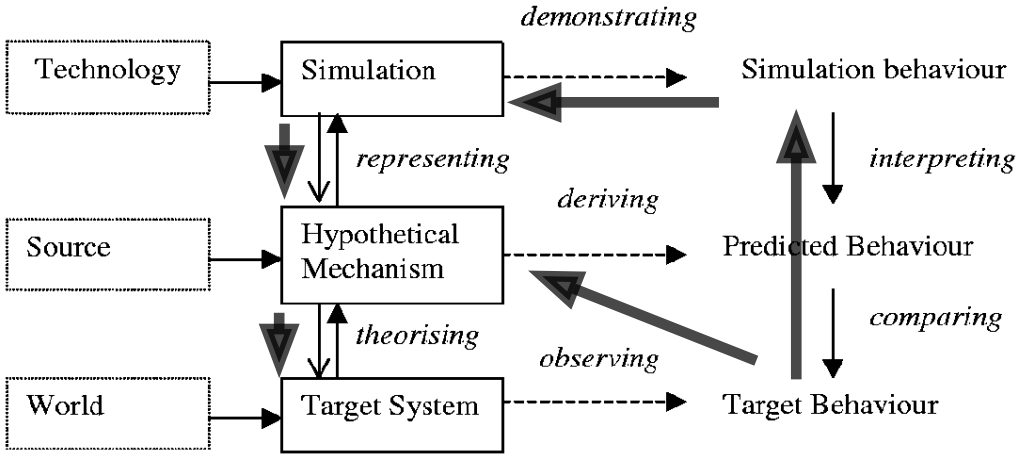


Figure 2-2 Reverse-pathway model usage; adapted from (Webb, 2001). This represents the use of a model and associated tools to generate hypotheses that are testable in the target system.

Of significant interest in this dissertation is the use of a model in a partially-reverse explanation pathway, shown in Figure 2-2. Once a simulation model has been produced whose behavior matches well enough with the predicted and target behaviors and where the structure represents the target accurately enough, one might be able to modify the model in order to reproduce other system behaviors, for which no explicit hypothetical method might exist. This can only be done if the methods for model operation and data representation are sufficient for this task. If the methods for data analysis and

representation and system representation are also sufficiently strong, it should then be possible to reverse-represent and reverse-derive the model and its behavior, arriving at new hypothetical mechanisms which might then be tested through experimentation in the target system.

This reverse pathway of model use is to some degree implicit in the process often referred to as "model refinement", however the explicit use of this pathway in order to guide experimentation is rare (as evidenced, partly, by the fact that it doesn't show up in Webb's presentation shown in Figure 2-1). This can only work if there is sufficient *structural accuracy* in the simulation model; to the degree that the simulation is a "black box" model, the tweaks in reverse-demonstration necessary to produce the new behavior will not have any clear use in reverse-representation, even if the behavioral match is perfect. Such a system will not be useful in making specific hypotheses about the function of the target system. Given the hypothesis generation as a specific goal of this work, all of the elements of model usage listed above need to be present at least to some extent.

2.2 Model Scope

When building a model of a biological system it is easy to get carried away with the details. Model builders can spend vast amounts of time in the process of model refinement, building greater complexity into the model in order to get a better match with the target system in one way or another. Model builders are sometimes even diverted into studying the properties of the model itself, never actually using it in the originally intended role as a tool for explaining function in the target system.

Given this potentially endless process of making a "good" model of a biological system, my goal was to satisfy each of the elements of model usage described above *just well enough* to make a reasonable attempt at the reverse-path hypothesis generation, with respect to the mechanisms of descending influence on changes in leg-level locomotor behavior. This introduces the problem at the other end of the scale where simplifying assumptions are too aggressive, making the structural accuracy or behavioral match of the model too weak to be useful. These concerns dictated a prototyping process whereby the necessary tools were built as simply as possible and improved as necessary.

3 Background

The work presented here is, of course, founded on a mountain of previous investigation. Although it does not go back to first references for all of the work involved, this chapter should provide a reasonable starting point for investigating the literature related to questions directly addressed in this dissertation, as well as many related or derivative questions.

3.1 The Walking Task

Walking is a means of locomotion which is ubiquitous among terrestrial animals, and the matter of considerable technical inquiry; see for example the advanced class coordinated by (Pfeiffer & Zielinska, 2003). It is a process which has evolved at least twice, and which shows a number of task restrictions, at least in the animal world, that are revealed through convergent evolution (Koditschek, Full, & Buehler, 2004; Ritzmann, Gorb, & Quinn, 2004)

The word "walking" is also applied to non-biological systems, including some very simple passive dynamic walkers (Collins, Wisse, & Ruina, 2001; Garcia, Chatterjee, Ruina, et al., 1998). In fact a considerable amount of robotics research has been devoted to the development of legged machines (see section 3.5). It has long been recognized that for low-complexity and non-sparse terrain, wheels and treads are likely to be faster, more efficient, more robust and simpler to implement than legged systems (Raibert, 1986). However, in complex terrain such as in rocky regions or in rubble, legs offer superior capabilities as described in (Ritzmann & Quinn, 2003).

At a higher level of abstraction is the review of coordination of "multisegmental organs" including legs for walkers and spinal cords for lamprey (Büschges, 2005). Büschges presents evidence that central pattern generators (CPGs) often can be or are broken down by physical segment, and mentions that we need to know the internal organization of these CPGs so that we can target and analyze the neural mechanisms for task-specific flexibility. A CPG is a network function that generates rhythmic motor output, even without any peripheral input (Grillner, 1975).

The body and various legs of an agile walking system must coordinate with one another to produce appropriate effective behavior. Equally important is the coordinated movement of the individual leg segments to carry and propel the body during stance phase and return to a desired touch-down position in swing phase.

3.2 *Insect walking behavior*

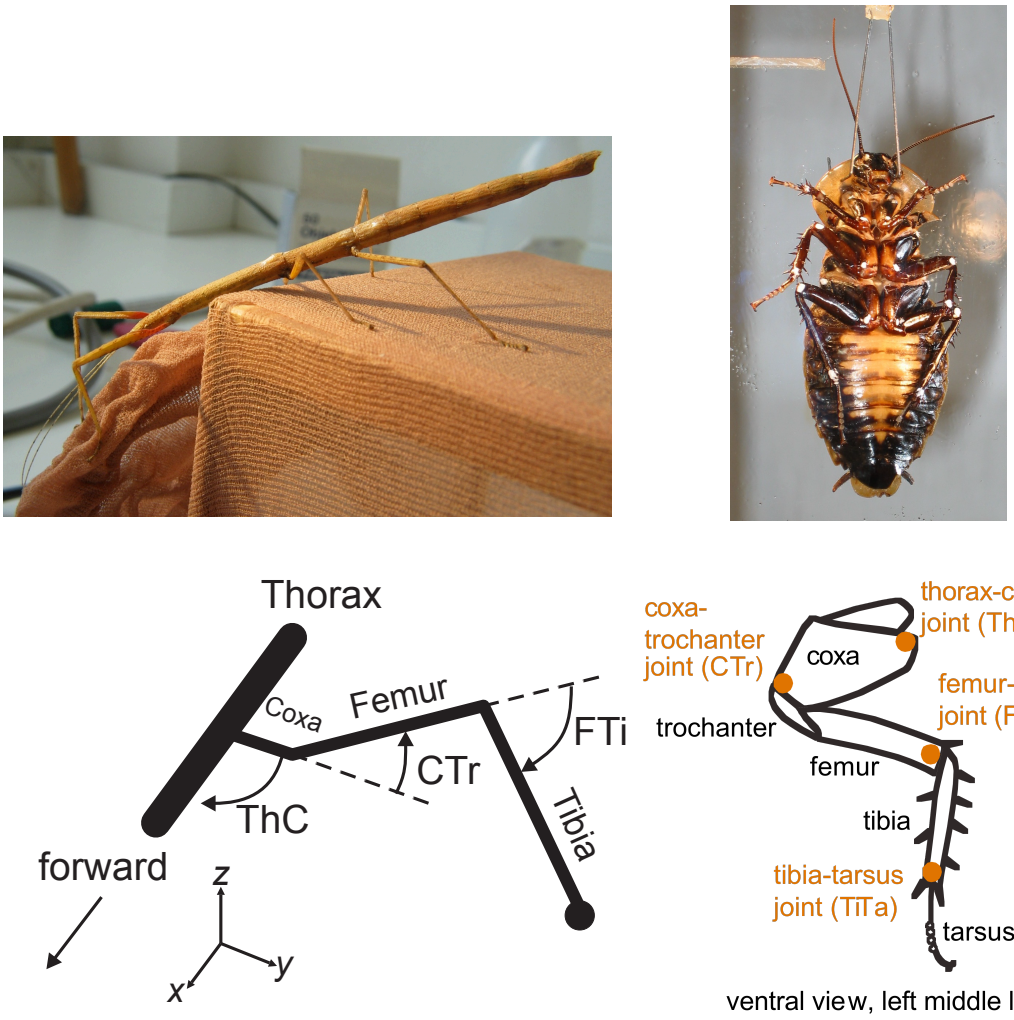


Figure 3-1 The insects and legs modeled. Top left: the stick insect *Carausius morosus*; Top right: the cockroach *Blaberus discoidalis*, ventral view. Bottom left: joints of the stick insect leg: Thorax-Coxa (ThC), Coxa-Trochanter (CTr) and Femur-Tibia (FTi). Bottom right: segments and joints of the cockroach left middle leg. Not labeled in either figure is the Trochanter-Femur (TrF) joint. In the stick insect this is fused but provides an autotomization (leg disconnect) point; in the cockroach it is a functioning joint only actuated in one direction (James T. Watson, et al., 2002).

The stick insects *Carausius sp.* and *Culiculina sp.* and the cockroaches *Blaberus sp.* and *Periplaneta sp.* are model systems for the investigation of insect walking. *Carausius morosus* and *B. discoidalis* are the species most used for the work in this dissertation. The animals and some anatomical terms are shown in Figure 3-1. The six legs of insects are

attached at the thorax, the central element of the insect body consisting of three fused segments. These segments are sometimes referred to as T1 through T3, T1 meaning thorax-1, the front (prothoracic) segment, to which the front legs are attached. T2 is the middle, mesothoracic segment, and T3 is the hind, metathoracic segment.

During walking, each leg goes through a cyclic motion consisting of "stance phase" or the "power stroke" while the foot is on the ground, and "swing phase" or the "return stroke" while the foot is in the air. In forward walking, the swing-stance transition happens at the anterior extreme position (AEP) and stance-swing happens at the posterior extreme position (PEP).

An early thorough description of behavioral kinematics and ground interaction forces of the stick insect *C. morosus* for walking on a horizontal path, horizontal plane, vertical path, and hanging from a beam, was presented in (Cruse, 1976). An interesting analysis of the function of the various legs under different situations can be found there. Later, (Cruse & Bartling, 1995) described more complete kinematics for *C. morosus*, on a treadwheel, oiled glass, and free walking on glass.

Behavioral mechanisms of gait coordination in arthropods were described in (Cruse, 1990), and later even distilled for use in robot control design (Cruse, Dean, Muller, et al., 1991). These detailed behavioral observations have been the basis for gait coordination in many walking robots, as described in section 3.5.

There has been considerable analysis of the dynamics of fast running and turning in *B. discoidalis*; e.g. (Full & Tu, 1990; Jindrich & Full, 1999). Part of this work has shown that cockroaches apparently use similar whole-body mechanics for running when compared to other animals of various sizes and numbers of legs (Full & Tu, 1990).

Associated with this work is coverage of stabilization effects in the mechanics of the animal (Jindrich & Full, 2002; Koditschek, et al., 2004).

Although the whole-body dynamics of running animals appears to be similar over a very broad range of sizes, the way the motion is generated may differ between scales. Scaling effects on locomotion and limb control were described in (Hooper, Guschlbauer, Blümel, et al., 2009); this is a verification of what the mechanical engineer expects about damping and passive stiffness becoming more important than gravity and inertia as the system scale decreases.

There has been a buildup of description of fine locomotor behavior and its control in the cockroach:

Watson and Ritzmann published descriptions of basic kinematics and muscle activity for primary muscles active during stance in the T2 and T3 legs for slow and fast running (Watson & Ritzmann, 1998a, 1998b). The cycle-to-cycle variation of slow motor neuron frequency and the recruitment of fast motor neurons were shown to affect running speed and (for the slow motor neurons) directional control.

Following investigation began to address "transitional" behaviors necessary for navigation in the natural world; in particular climbing and turning. In climbing it was found that below some threshold of obstacle height, no kinematic change in leg motion is necessary; above that the T2 leg is redirected by changes in kinematics both at the "alpha" degree of freedom (DOF) at the Thorax-Coxa joint (this is similar to the ThC-1 DOF described in Chapter 5), and Trochanter-Femur rotation (James T. Watson, et al., 2002). Apparent reflex adjustments are made in other joints in T2 leg, due to the different

physical environment this change in leg posture provides (Watson, Ritzmann, & Pollack, 2002).

Differing motor patterns between walking and turning were investigated in (Mu & Ritzmann, 2005). Distinct changes in coordination were observed in the T2 leg. In walking, the CTr and FTi joints extend during stance and flex in swing; in inside turning (where the leg in question is on the side being turned towards; e.g. the left leg during a left turn) this relationship is reversed; both CTr and FTi flex during stance and extend during swing. There is also a change in the order of extension (CTr depression before FTi extension in walking; reversed in turning). In (Mu, 2007; Rutter, Taylor, Mu, et al., 2007) a kinematic model (based on work in Chapter 5) was used to find that this reversal of order and the pose of the ThC joint are both critical in defining the differences in foot motion between walking and turning.

Force and contact sensing information carry valuable information about the animal's interaction with the environment. A review on force influences in control of posture and walking for both cockroach and stick insect is presented in (Zill, et al., 2004).

3.3 The descending control problem

Of primary interest in the understanding of agile navigation of varying terrain is the ability of animals to smoothly transition between behaviors requiring substantially different local behavior of locomotor appendages.

Local control circuits are occasionally considered "reflex" and the descending control is "voluntary", however where to draw this line is not always clear, and some would argue not a proper thing to do in the first place (Prochazka, Clarac, Loeb, et al., 2000).

Nevertheless, “reflex” is a very common term and is used in this document. To help clarity, I attempt to adhere to the convention used in (Ekeberg, et al., 2004), shown in Figure 3-3, which differentiates between “timing” and “magnitude” influences. The term “coordinating influences” is sometimes used here to mean the same thing as “timing influences” as defined by Ekeberg et al.; this is because time is not actually explicitly involved in these influences, just the relative phase or coordination of activity.

Also, there is a range of the complexity of neural involvement in system behavior in locomotion control, from the physical-only dynamic properties of muscles, to low-level reflexes, to reactive behaviors, to high-level predictive behavior (Prochazka & Yakovenko, 2002).

The ability of descending influences to change the gains of and even reverse some local magnitude influences was shown in (Mu & Ritzmann, 2008a), for both inter- and intra-joint influences. There was a reversal of CTr slow depressor response to stimulation of the femoral chordotonal organ in a quiescent state. These experiments were done via gross removal of descending influences; severing all of the connections between the local thoracic ganglia and the brain. This paper also posed the general reflex cascade hypothesis, the testing of which was a specific aim of this work. The idea is that the alteration of a few local reflexes can lead to a different physical state of the leg, thereby triggering further changes in local influences, resulting in an overall different leg behavior.

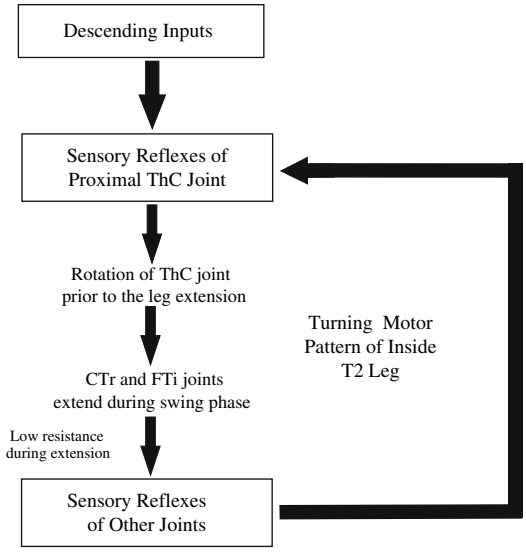


Figure 3-2 Fig. 1 from (Mu & Ritzmann, 2008b), described as "A schematic figure of the reflex cascade hypothesis on how descending inputs generate inside turning motor pattern." In that work, the "low resistance during extension" signal is physically induced by substrate removal and results in searching behavior of the leg, which is similar to inside turning.

In (Mu & Ritzmann, 2008b) it was demonstrated that physical manipulation can trigger a change in behavior in the absence of descending control- this is support for the specific reflex cascade they hypothesized for inside turning, which is shown in Figure 3-2. This specific cascade depends on magnitude influence responses to leg load, as well as other unspecified reflexes.

3.4 Stick Insect Leg Control

The stick insect has been used in substantial investigation of joint-level neural control and the characterization of specific neural pathways. Evidence in the leg muscle control system of the stick insect suggests that the neuronal control can be subdivided into several central pattern generators (CPGs). In the stick insect, each joint CPG can generate a basic alternating activity pattern in antagonistic muscles of a given leg joint. It is possible to activate these CPGs either pharmacologically or by tactile stimulation of the

animal (Büschges, et al., 2004; Büschges, Schmitz, & Bässler, 1995). Experiments with pharmacological activation of the leg control network lead to the conclusion that each leg joint (ThC, CTr, FTi; see Figure 3-1) can be associated with an individual CPG. This is based on the observation that the joint motor neurons show alternating activity (oscillations) under pharmacological activation but do not show complete cycle-to-cycle coupling between the oscillations of motor neuron pools associated with different joints. Signals of sense organs can a) change the probability of the CPG to stay in or transition to a given state (timing influences) or b) shape the amplitude of the motor output (magnitude influences) (Büschges, 2005).

To generate a stepping movement the activity of these joint control CPGs must be coordinated. This coupling is achieved primarily by means of sensory feedback (Akay, Bässler, Gerharz, et al., 2001; Akay, Haehn, Schmitz, et al., 2004; Bucher, Akay, DiCaprio, et al., 2003; Hess & Büschges, 1999), although central coupling does have some effect in isolated nervous systems (Büschges, et al., 1995). Any central coupling is too weak to fully couple the joint CPGs into activity resembling "fictive locomotion", at least in these preparations (Ekeberg, et al., 2004). Usually leg proprioceptors can be attributed to a specific leg segment and therefore be associated with the joint that moves this segment. If a sense organ influences the CPG of its associated segment, we call this an intra-joint influence, if it affects the activity in other joints it is an inter-joint influence. The femoral chordotonal organ (FCO) for example is a stretch receptor inside the femur in both stick insects and cockroaches. It is able to signal parameters of the joint geometry like joint angle or angular change (Bässler, 1993). Signals from the FCO in the stick insect do not only influence the CPG of the FTi joint itself (Bässler & Büschges, 1998)

but also the motor-activation of the adjacent (CTr) joint (Bucher, Akay, DiCaprio, et al., 2003; Hess & Büschges, 1999). These inter-joint influences are an important mechanism of segment coordination.

The role of individual leg sense organs in the control of motor output for stepping has been studied in great detail (Bässler & Büschges, 1998; Büschges, 2005). In order to cope with the complexity of this topic, many experiments have been performed using so-called ”reduced preparations”. In these preparations one tries to approach operational isolation of the behavior of interest by inactivation or removal of dispensable biological functionality. In the case of the stick insect a popular reduced experimental preparation is the single leg preparation, with only one leg remaining attached to the animal (Bässler, Rohrbacher, Karg, et al., 1991; Fischer, Schmidt, Haas, et al., 2001). Often the remaining leg is further limited in its degrees of freedom by preventing forward and backward movements, referred to as the restricted single leg preparation. This experimental setup creates a movement that is sometimes referred to as ”sideways walking”.

In order to change the coordination between the joint CPGs, and thereby the resulting behavior, it is necessary that the sensory coupling be changeable. A load influence on the timing of the ThC joint CPG has been shown by (Akay, Ludwar, Goritz, et al., 2007) to reverse between forward and backward walking behaviors.

A synergistic interaction of load and joint angle sensory signals was described in (Akay & Büschges, 2006). Also a reversal of magnitude reflex/influence between quiescent and active states; also previously known “active reaction” of (Bässler, 1988).

3.4.1 The Synthesis of Ekeberg, Blümel and Büschges

The biological results above were the basis for the development of a new model of leg movement control by (Ekeberg, et al., 2004). In the middle leg of *C. morosus*, a number of neural pathways were known which defined sensory coordinating influences capable of switching the activity in the joint CPGs. During stick insect stepping behavior, these CPGs behave as bi-stable circuits, specifying either flexion or extension in the associated joint, as generally represented in Figure 3-3. Note in that figure the qualitative difference between "timing" influences, which affect the selection of the currently active motor

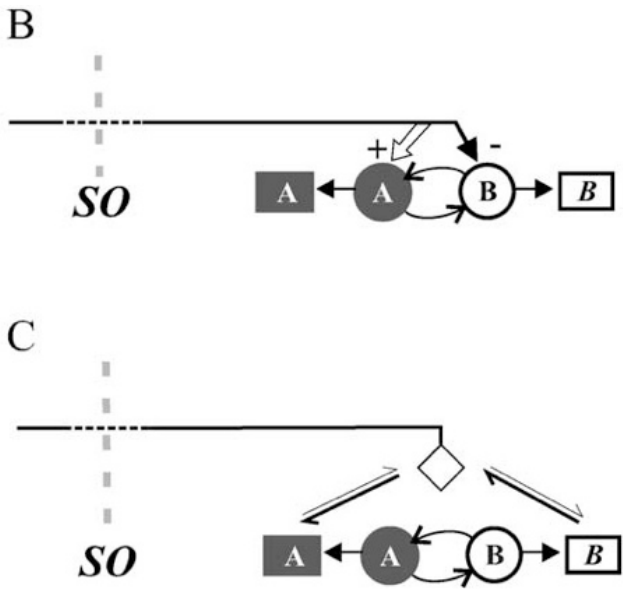


Figure 3-3 Figure 1 B and C from (Ekeberg, et al., 2004). The caption is as follows: "…The two possible states are represented by the two circles. Motoneuron pools are represented by the rectangular components. (B) The used notation scheme demonstrating the influence of a sense organ (SO) on state transitions of a neural circuit, referred to as 'timing influences'. In the example shown, signals (movement, position or strain) detected by the SO initiate or assist a fictive state A. The positive arrow means that the system will react to the signal with an (immediate) transition into state A. If necessary it will therefore leave state B. (C) This figure shows an example for the second class of influences described, here called 'magnitude influences'. Signals from the sense organ (SO) affect the output activation level of the joint controlling system in a specific way: Proprioceptive information from a sense organ (SO) e.g. movement or position data, will in this case increase the state A effector activity and decrease the magnitude of effects associated with state B."

pattern, and "magnitude" influences, which affect the strength of the associated motor neuron activation.

Central coupling is weak enough to be ignored under these circumstances. It was hypothesized that these influences were sufficient to generate stepping. They constructed a set of state machines whose transition rules implemented these influences, and used that controller in a dynamic computational model of the stick insect middle leg. The resulting model generated stepping motion by going through a repeating series of motions as shown in Figure 3-4. This successfully demonstrates the sufficiency of these combined mechanisms to explain forward stepping in the biological leg to a large degree. They modified the controller to successfully generate stepping in models of the front and rear legs as well, making use of a timing influence reversal to qualitatively change leg motion

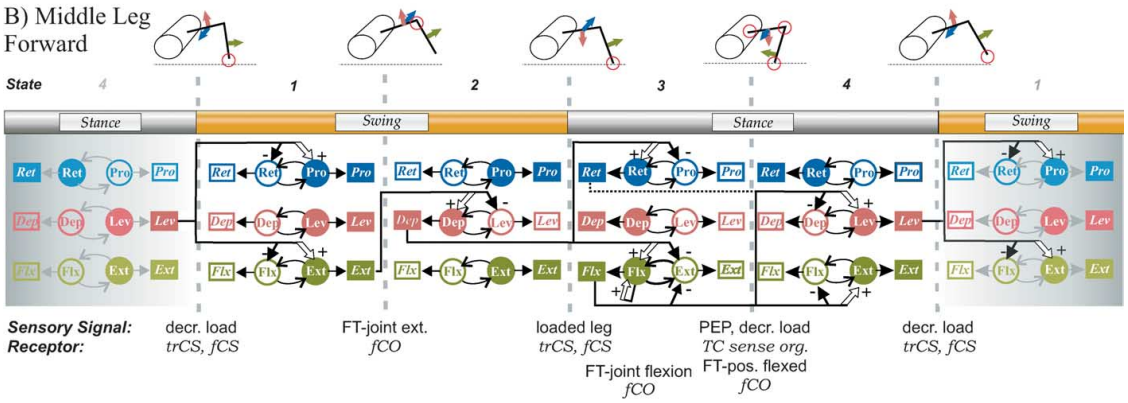


Figure 3-4 Figure 3B from (Ekeberg, et al., 2004). Their description of the figure is summarized as follows: "Diagrams showing the sequence of events … (B) Normal walking of the middle leg. … time progresses from left to right and dashed vertical lines indicate when feedback from sensors causes a state transition. The stick figure on top illustrates the mechanical situation whenever a state transition in the neural circuits occurs, with arrows indicating movement direction and a red circle indicating the sensed event. At the bottom of the dashed line the sensed signal along with the relevant receptor is named. The arrows horizontally passing these dashed vertical lines show how the bistable circuits are affected by the sensed event … Notice that these influences originate from rectangular boxes, illustrating effectors (motoneurons/muscles), and terminate on circles, illustrating the bistable circuits … Colors are used to denote joints: TC is blue, CT is red and FT is green."

in the rear leg. Since the controller was explicitly and structurally derived from known neural pathways, the changes for front and back legs represented specific hypothesis regarding the neural function of the animal in those legs.

3.5 Stepping control methods in robots

Biological systems have been a source of inspiration for robotic walking machines at varying levels of abstraction, ranging from the highly abstracted Whegs™ (Allen, Quinn, Bachmann, et al., 2003) and RHex (Altendorfer, Moore, Komsuolu, et al., 2001) vehicles, through less abstract but still highly simplified systems like MechaRoach robots (Boggess, Schroer, Quinn, et al., 2004; Wei, Quinn, & Ritzmann, 2004), to more flexible and complex systems, including Robot II (Espenschied & Quinn, 1994; Espenschied, Quinn, Beer, et al., 1996), the TUM walking machine (Pfeiffer, Weidemann, & Eltze, 1994) the Tarry series (Buschmann, 2000a, 2000b), the Lauron series (Gassmann, Scholl, & Berns, 2001), and BILL-Ant-p (Lewinger, 2005; Lewinger, Branicky, & Quinn, 2005).

These systems use methods of varying complexity for coordination between and within the legs. In Whegs™ and MechaRoach both inter- and intra-leg coordination are accomplished through mechanical coupling. Indeed, in both Whegs™ and RHex systems the intra-leg coordination problem is solved by using single-link appendages. While these systems are outstanding in their simplicity and physical robustness, there is a limit to the complexity of behavior such systems can exhibit, and a corresponding limit to the complexity of locomotion tasks they can solve.

Coordination between mechanically uncoupled legs to produce gaits has been successfully implemented (Espenschied, et al., 1996; Espenschied, Quinn, Chiel, et al.,

1993; Lewinger, et al., 2005; Pfeiffer, et al., 1994) using rules based on animal behavior (Cruse, 1990). Intra-leg coordination in such systems has most typically been done using inverse kinematics (Choi, Rutter, Kingsley, et al., 2005; Espenschied, et al., 1996; Lewinger, et al., 2005; Nelson & Quinn, 1999; Pfeiffer, et al., 1994). While this is conceptually straightforward (unless kinematically redundant as in (Choi, et al., 2005; Nelson, Quinn, Bachmann, et al., 1997)), dealing with dynamic environments and perturbations can be a matter of considerable effort; explicit desired foot paths and explicit handling of obstacles is usually required. In addition, these methods require trigonometric and other computations that are often beyond the capability of microcontrollers found in small-scale robots. As a result, such robots are either larger, to house on-board processors, or require off-board, tethered control systems, which can limit mobility and usable range of operation. Other solutions for intra-leg coordination are based on neurally-inspired pattern generators whose action can be modified by sensory input, e.g. by Wadden & Ekeberg (Wadden & Ekeberg, 1998; Wadden & Ekeberg, 1999) or more loosely in Walknet (Cruse, Kindermann, Schumm, et al., 1998; Dürr, et al., 2003). The computational complexity of such systems can also be relatively high, however, since they may require many simultaneous integrations to simulate the artificial neural dynamics. Conceptually between these methods are arrangements such as that used in Genghis (Brooks, 1989) where the computational elements are loosely based on neural function, and the network organization is arranged in a more engineering-like decomposition of events and functions.

3.6 Robotic control architectures

It may be useful for the reader to compare the control description presented in Chapter 4 with previously developed robotic control architectures, some of which are described here.

3.6.1 Three-layer architectures

Somewhere in between the "old fashioned" serial Sense-Plan-Act cycle of control (Nilsson, 1980) and the "new approach" using reactive agents and subsumption architecture (Brooks, 1986) are three-layered architectures (Gat, 1998). These typically consist of a controller/reactive layer on the "bottom" executing low-level behaviors, often like "go forward" and "avoid obstacle", a sequencing layer in the middle choosing which (optional) behavior(s) is/are active for more complex but immediate tasks like path following, and a deliberative layer for high-level things like path planning, goal selection, mapping, and so on.

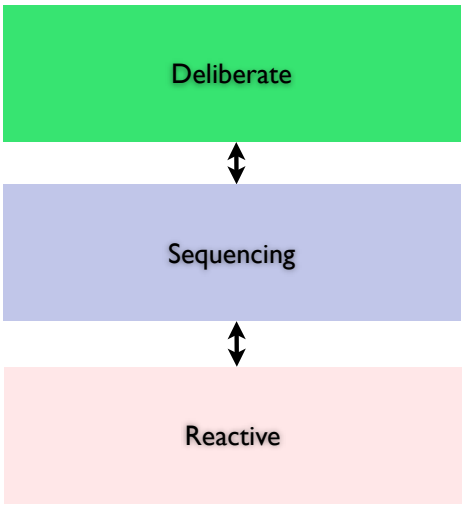


Figure 3-5 Schematic of a three-layer architecture. Sensory and control data can pass between layers, and is usually more compressed or symbolic at higher levels. Usually all three levels are processing simultaneously. Note that this is not the same as either of the control architectures shown in Figure 3-6.

3.6.2 Subsumption architecture and the use of finite state machines

A finite state machine is a model of computation, in our usage more usefully thought of as a computational element, which has a set of states, a transition function which maps current state and input to the next state, an input alphabet, and a start state (Black, 2008).

Systems which are controlled by computers but exist in the physical world are usually actually "hybrid dynamic systems" (Branicky, 1997; Gollu & Varaiya, 1989). These systems are often modeled as either a discrete event system or as a continuous system, but it can be beneficial (and is more strictly correct) to take account of the hybrid nature. There has been work developing tools for the representation and analysis of such systems (Branicky, 1995).

An overview of reactive robotics and the use of subsumption architecture is given in (Brooks, 1991). This includes a description of the concepts of *situatedness*: an agent having immediate perception only, or at least having the immediate perception be more important in choosing actions than abstract models, and *embodiment*: agent actions are "part of a dynamic with the world" and produce immediate feedback.

The initial subsumption architecture work is described in (Brooks, 1986); this began a revolution in mobile robots. The subsumption architecture is a hierarchy of increasingly complex and lower priority tasks, which can subsume (take over) the behavior of a lower-level module by suppressing its output or the input it provides to other specific lower-level modules. Brooks assembled these controllers using augmented finite state machines; state machines with instance variables and four types of states: output, side effect, conditional dispatch and event dispatch. A graphical comparison between

subsumption architecture and the more old-fashioned sense-plan-act cycle is shown in Figure 3-6.

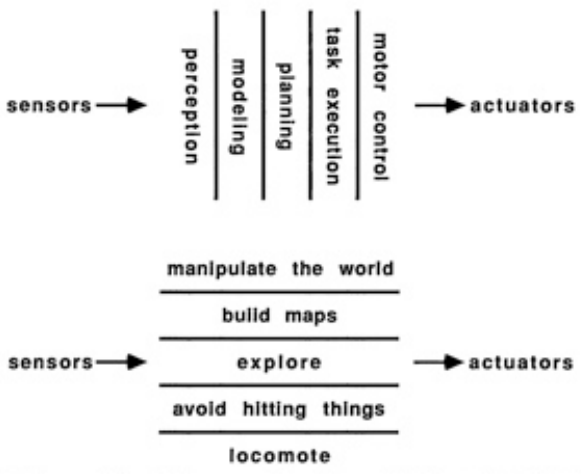


Figure 3-6 Reproduction of Fig. 2 from (Brooks, 1991). Top: a sense-plan-act architecture of functional modules; bottom: a subsumption architecture of task-achieving behaviors.

Genghis was a vaguely insect-inspired robot demonstrating the use of subsumption architecture and finite state machine control of walking in a six-legged robot (two DOF per leg) (Brooks, 1989). Alarm clocks were added to the augmented finite state machines. Position control was applied at the joints, with higher-level input from force feedback, collision sensing and tactile sensors. Genghis' arrangement of augmented finite state machines for control is shown in Figure 3-7.

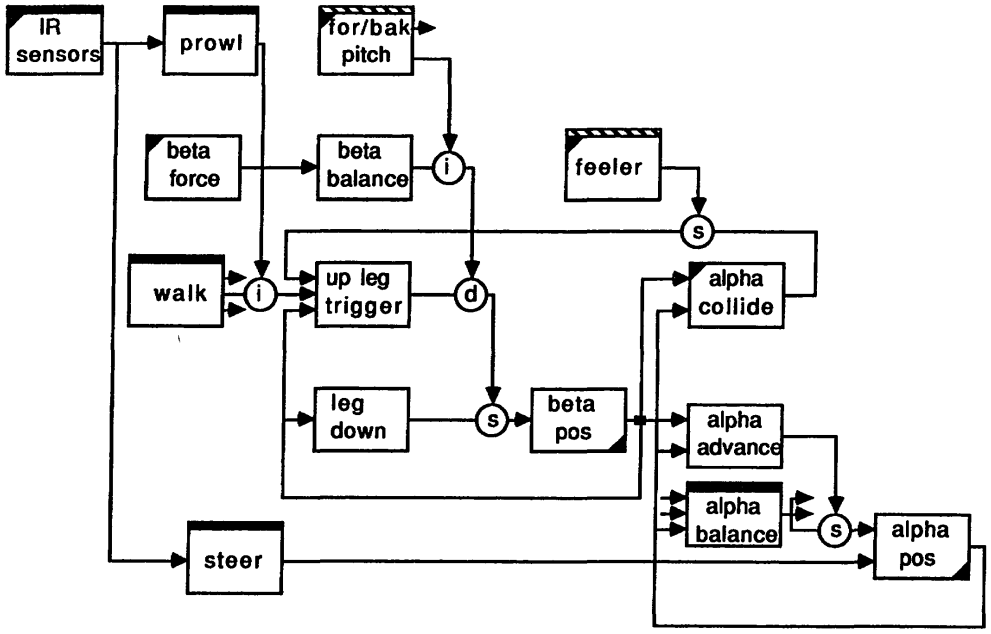


Figure 3-7 Reproduction of Fig. 4 from (Brooks, 1989). The 57 augmented finite state machines in the subsumption network for controlling Genghis. Elements with solid bands are unique; striped bands are repeated twice for specific legs, and the rest are each repeated six times; once for each leg.

3.7 Robotic Models of Biological Systems

A detailed framework for the description of models, specifically geared towards robotic models of biological behavior/systems, is provided by (Webb, 2001). The seven dimensions of model description she presents (as shown in Figure 3-8) are:

1) Biological Relevance: Is the biological target system clearly identified? Does the model generate hypotheses for biology?

2) Level: What are the base units of the model? (e.g. atoms, individuals, populations)

3) Generality: How many systems does the model target?

4) Abstraction: How many elements and processes from the target are included in the model? (more detailed models are less abstract)

5) Structural Accuracy: Is the model a true representation of the target? How well do the mechanisms in the model reflect the real mechanisms in the target?

6) Match: To what extent does the model behave like the target?

7) Medium: What is the simulation built from? (e.g. symbolic, computer simulation, physical implementation)

The interested reader should consult this reference for a complete treatment. The definition of "model" used here has already been addressed in section 2.1.

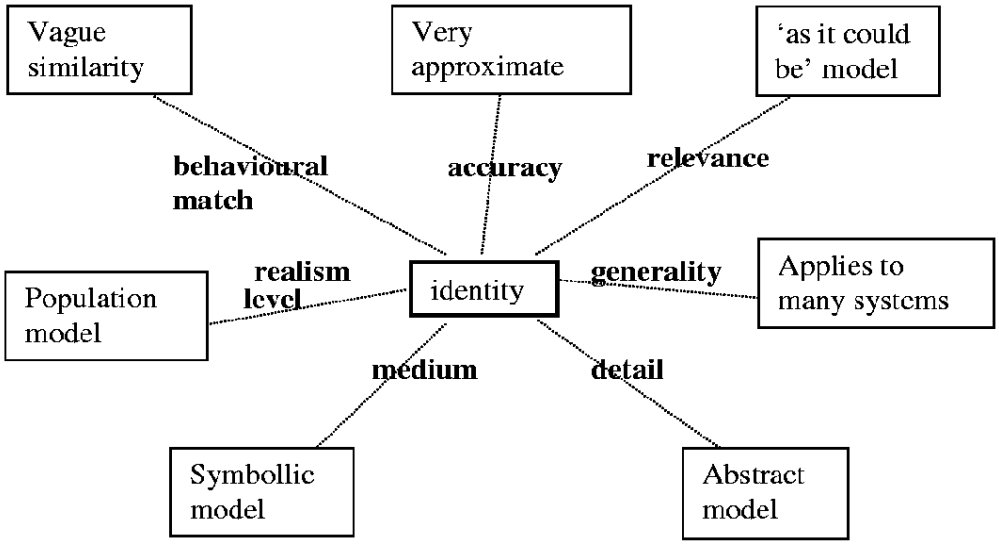


Figure 3-8 Dimensions of model description, each expressed from identity to loose association with the target system. Reproduction of Figure 2 from (Webb, 2001).

A compendium of biorobotics models is presented in (Webb, 2001), but a few specific examples are provided here:

Robot I (Espenschied, et al., 1993) was the first robot behavioral model of gait coordination, based on the behavioral model of stick insect behavior from (Cruse, 1990). This was followed by the TUM robot (Pfeiffer, et al., 1994) and the Tarry series

(Buschmann, 2000a, 2000b) which uses the more fine-grained behavioral basis of Walknet (Dürr, Schmitz, & Cruse, 2004).

Robot III (Bachmann, Nelson, Flannigan, et al., 1997; Nelson, 2002; Nelson & Quinn, 1999) was kinematically modeled on cockroach behavior, and gait coordination was performed using a generalized version of the rules from (Cruse, 1990). Robot V (Choi, et al., 2005; Kingsley, Quinn, & Ritzmann, 2003) was similar in intent to Robot III, but powered with braided pneumatic actuators. Robot V was used in some attempts at biological modeling of cockroach joint/muscle dynamics in (Rutter, Mu, Ritzmann, et al., 2007).

A fairly close implementation of cricket phonotaxis sensory circuits has been used as input for gross behavior in robots (Webb & Scutt, 2000). This was later extended to more abstract models for outdoor embodiment (Horchler, Reeve, Webb, et al., 2004; Reeve, Webb, Horchler, et al., 2005). The reproduction of animal motion in dynamically scaled models of wings (Dickinson, Lehmann, & Sane, 1999) was done in order to measure the physical processes involved; no neural basis or behavioral modeling was involved.

Probably the best prior example of a neuromechanically based robot is by (Ijspeert, Crespi, Ryczko, et al., 2007). This used biologically based and/or hypothesizing CPG control circuits with an abstracted but fairly accurate salamander body. They were able to address their specific questions regarding the evolution of walking control using position control and no modeling of neural sensory feedback.

The generation of lamprey swimming by central pattern generators was examined by (Ekeberg & Grillner, 1999) in a computational neuromechanical model. This and similar work was a precursor to the above work and the AmphiBot lamprey/snake robot (Crespi,

Badertscher, Guignard, et al., 2004). The computational model showed distinct improvement of behavior when perturbed when segment-level sensory feedback was added. This work also investigated mechanisms for generating different behaviors using brainstem/descending input (though more of a continuum here than for stepping).

4 Sensory Coupled Action Switching Modules

4.1 The basis in biology

The implementation of controllers capable of generating insect stepping behavior in the same manner as described by (Ekeberg, et al., 2004), namely through a set of sensory-coupled pattern generators as described in section 3.4.1, led to the development of Sensory Coupled Action Switching Modules (SCASM) as a control concept in its own right. This chapter first presents the development of the concept from its biological basis. A number of examples of the application of these tools for the description and use of this concept in modeling and control, particularly with respect to the elements of usage described in section 2.1, are then presented. These tools and examples will be used and referred to in the later chapters on model construction and experimentation.

4.1.1 Stick Insect Controller Abstraction

The model controller from (Ekeberg, et al., 2004) was the basis for developing a new system of leg movement control which consists of independent joint control systems. In this system each joint controller is a bi-stable pattern generator, and the coordination task of each joint pattern generator is to determine whether to be in the joint's flexion or extension state. To make this decision the joint controller has access to specific sensor data as in the animal, but it has no direct central information about the state of other joint controllers. This reflects the effective lack of central coupling in the biological archetype.

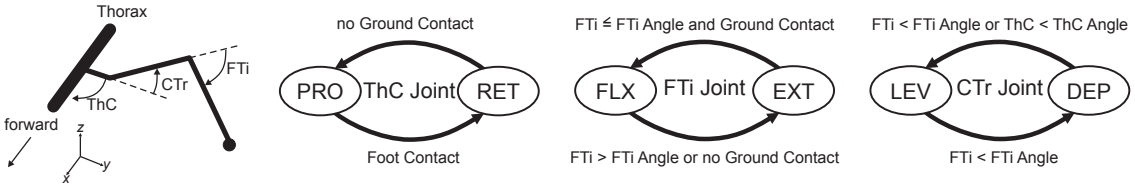


Figure 4-1 Diagram of geometry and control of the simulated stick insect middle leg, showing ThC (Thoraco-Coxal) protraction and retraction, CTr (Coxa-Trochanter) levation and depression, and FTi (Femur-Tibia) flexion and extension. The conditions used by Ekeberg et al. for transitions between states are shown adjacent to each transition arrow. The leg segments, from the body outward, are the coxa, femur and tibia. Taken from (Lewinger, Rutter, Blümel, et al., 2006)

If the joint pattern generators are implemented as finite state machines, as was done in (Ekeberg, et al., 2004), it is possible to express each pattern generator's behavior as a set of Boolean decision rules in the following form:

```
if (sensor-data_x > threshold_x) then
   state = flexion
else if (sensor-data_y > threshold_y) then
   state = extension
else
   state = previous_state
```

A graphical representation of these transition rules for each of the joint pattern generators in the stick insect model is shown in Figure 4-1. In the case of the stick insect the appropriate rules for each joint can be derived almost completely from the literature; only the threshold values have to be adjusted to function with a particular leg geometry.

This description of the stick insect model controller has been variously presented in (Lewinger, et al., 2006; Rutter, Lewinger, Blümel, et al., 2007; Rutter, Lewinger, Taylor, et al., 2006), and applied to the several robotic legs there in addition to the dynamic simulation of (Ekeberg, et al., 2004). The extension of these concepts has led to the current understanding of SCASM presented below.

4.2 General system concept

A SCASM control system consists of a set of action switching modules, each of which has some direct influence on a subset of the entire system to be controlled. In the example of the stick insect leg in Figure 4-2 A, the module shown controls the action of the Femur-Tibia joint; a subset of the entire leg. Each action switching module has associated with it a set of actions. An action describes some effect on the dynamics of the system subset associated with its module. In this example, the FTi joint has a single degree of freedom, and it is natural to consider one action which flexes the joint (FLX) and another which extends it (EXT). The module is then a control unit which switches between these actions, making it an Action Switching Module. In order to avoid confusion with the term Algorithmic State Machine, action switching modules will be referred to simply as "Modules", rather than "ASMs".

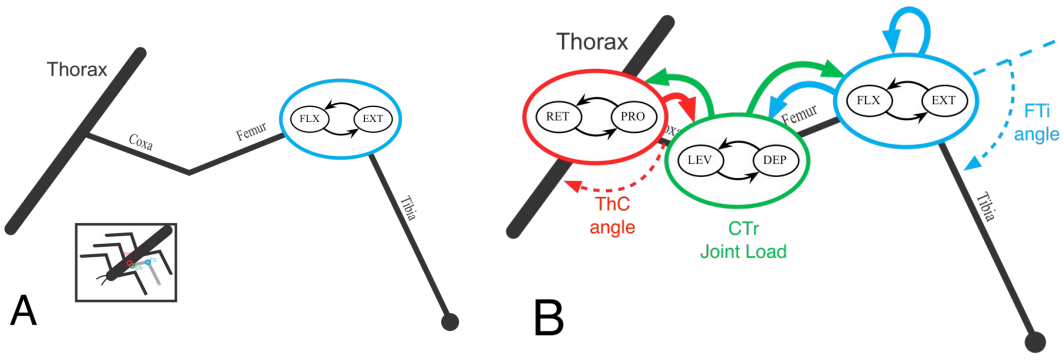


Figure 4-2 A) The Femur-Tibia Action Switching Module on the stick insect middle leg (entire animal shown in inset). B) Sensory-flow diagram of the entire leg (as implemented in this work). Though the control actually takes place in the thoracic ganglion or computer, conceptually the action switching modules are considered to be located at their associated joints.

The dynamics of an action can be quite complex. In the FTi joint example, the relevant action output is the torque and/or movement that results at that joint. One can consider the dynamics of an action to consist both of the "neural-level" control system defined

locally by the action, in concert with the "mechanical-level" dynamic response to the neural-level system outputs, as represented in Figure 4-3. The neural-level dynamics of the action can include any amount of signal processing and complexity we might associate with the term "control system", including sensory feedback and multiple control inputs. The mechanical-level dynamics can be similarly complex, but may not change

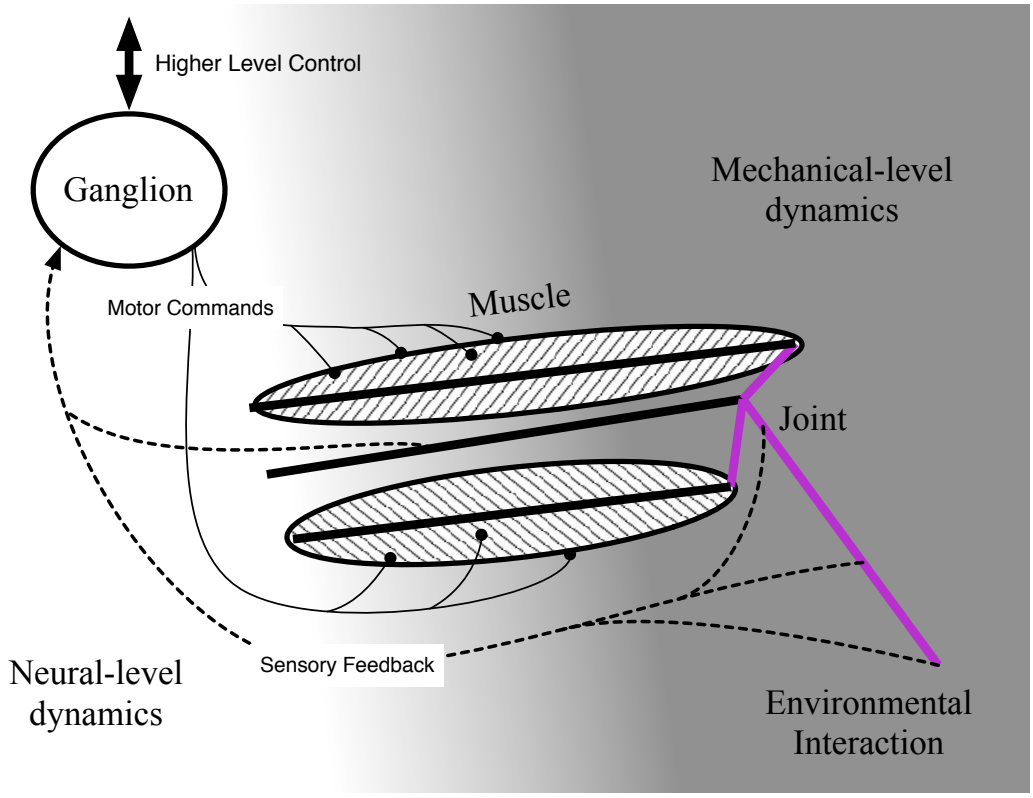


Figure 4-3 Interaction of neural-level and mechanical-level dynamics at a single joint. The neuromuscular transform (Brezina, Orekhova, & Weiss, 2000; Hooper, Brezina, Cropper, et al., 1999) and sensory transduction are where the two regimes meet. Neural-level dynamics can be modified by, e.g., interneuron function. Mechanical-level dynamics can usually be considered constant; but are modified by environmental interaction.

when the action switches at a module. Mechanical-level dynamics can include simulated mechanics as well as a real physical plant. In the FTi joint example, we might expect each action to define a set of muscle activation rules for each of the muscles at the joint. These rules could then use both sensory inputs and neural-level state information from

other parts of the animal in order to generate the inputs to the muscles. Meanwhile, the joint geometry, gross muscle properties, leg mass and damping do not change.

In a SCASM system, the switching between actions at a module is influenced by sensory signals, which can be affected by the actions at other modules. These inter-module influences couple the modules via sensory feedback, making the system a set of Sensory Coupled Action Switching Modules. The behavior of the entire system depends on the strength and sense of the inter-module connections, together with the neural-level dynamics of each action, the mechanical-level dynamics of the entire system (including mechanical coupling between modules, which is not explicitly represented in the tools developed here but might be addressed by formal representation as a hybrid dynamic system), and the environment. In Figure 4-2 B, the arrows between modules represent this sensory coupling. The FTi angle sensor signal, for instance, influences the switching between actions at both the FTi module and the CTr module.

4.2.1 Introduction to the Event-space diagram

The sensory-flow representation introduced above conveys the basic neural-level architecture of the system, but it does not provide enough information to fully describe behavior or specify an implementation of control. The sensory arrows represent the direct sensory influences between modules, but not which actions are promoted, or what level of sensory activity is necessary to do so. We can generate a more explicit representation of the sensory connections in a SCASM system using an event-space diagram, shown in Figure 4-4.

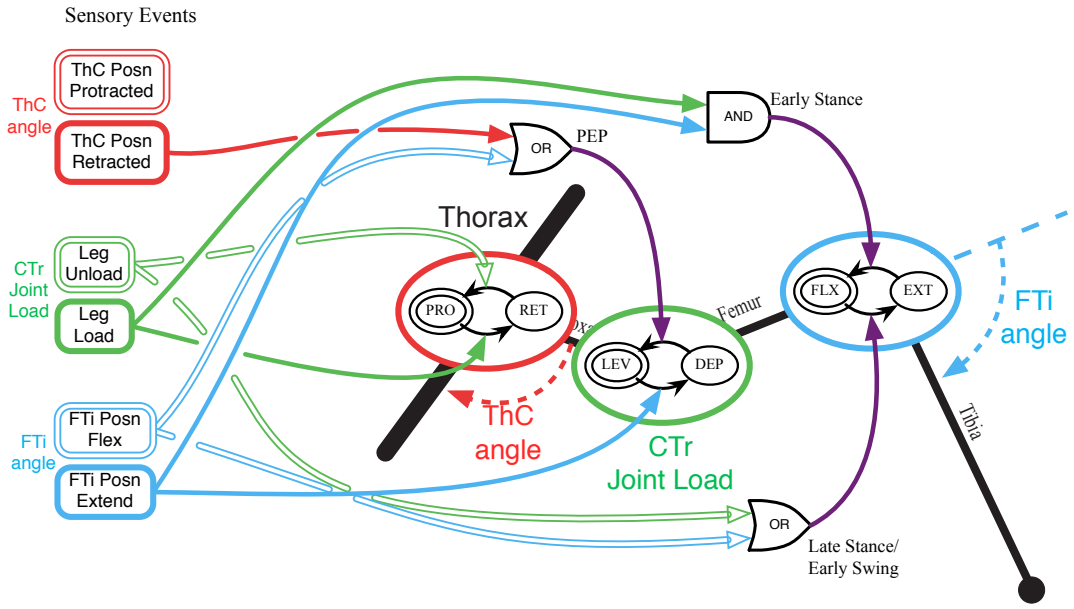


Figure 4-4 Event space diagram for forward stepping, stick insect left middle leg. The double-line actions specify system dynamics which lead to the double-line sensory events, and solid-line actions likewise lead to solid-line events. Color encodes the module most directly involved in generating each sensory event; the sensory events are also labeled by the sensors involved. For example, FTi FLX (blue module, FTi angle, double-line action) leads to the "FTi Posn Flex" event, whereas EXT (blue, FTi angle, solid-line) leads to "FTi Posn Extend". The compounded events are labeled with the complex sensory event or state they represent; e.g. retracted ThC or flexed/ing FTi represent an extreme of motion towards the end of stance, combining into the Posterior Extreme Position (PEP) event.

The action switching modules are shown as before; a set of modules, each of which contains a set of actions. Transitions that may occur between actions are shown by the black arrows connecting the actions within a module. Rather than explicit sensory signals, the influences on action switching are represented here as sensory events. An example from the FTi joint of the stick insect leg would be "FTi joint is extended/extending", which promotes both CTr depression and, in combination with leg load, FTi flexion.

The simplest example of sensory influences in the stick insect leg is the load influences at the ThC module. Leg load promotes retraction; meaning that if the leg is loaded or loading, i.e. touching the ground, the ThC will rotate the leg backward. Leg unload promotes protraction, meaning that if the leg is unloaded/unloading/not touching the ground, the ThC joint will push forward. Retraction while on the ground and protraction while in the air produce forward stepping.

Note: different lines from a single "event" may have different gains or real sensory sources in an implementation; in this case these lines don't represent *exactly* the same sensory event, but similar/related ones, as described in section 4.5.2 and shown in Figure 7-5.

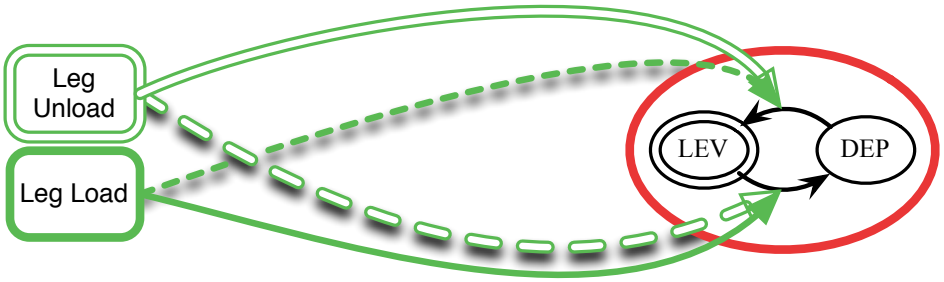


Figure 4-5 In biological systems there can be multiple competing signals as shown here, whose strength can be modulated. The dashed lines represent a load reflex reversal at this module, when compared to the solid lines. In the event-space representation of circuits and behavior, we have only drawn the influences dominant in a behavior. Those can then be translated into Boolean transition rules.

The event space diagram is the most general representation of a SCASM system behavior yet developed. Note, however, that the actual underlying biological circuitry usually has multiple continuously variable elements as shown in Figure 4-5. There is a range of systems that can be described or designed in this manner; a generic case is shown in Figure 4-6. In the general case, note that there may be no way to directly switch between some of the actions. Thus in the generic example Module I can switch into and out of Action 1 from any of the other actions, but it cannot switch directly between Actions 2 and 3.

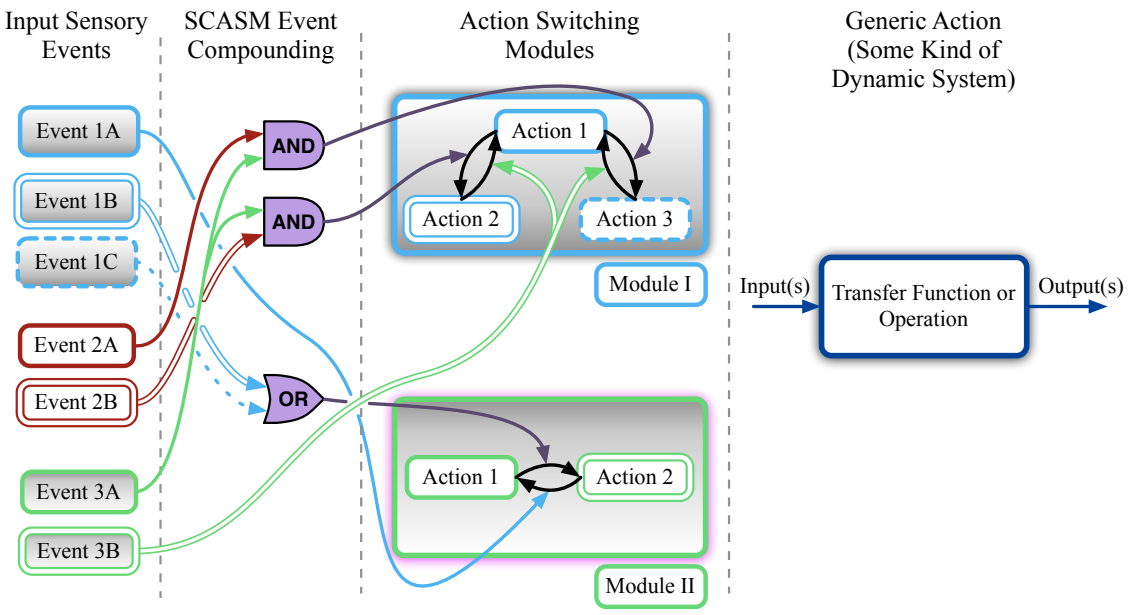


Figure 4-6 A generic event-space diagram. Sensory events, which may be compounded, influence the transition between actions in the action switching modules. Sensory coupling occurs when the action at one module affects the sensory events which influence transitions in a different module. Events 1 are results of the actions in Module 1; all lined in blue and shaded dark at the bottom. The line style reflects which action tends to lead to which sensory event: action 3 tends to produce sensory event 1C. Likewise events 3 are generally the result of actions in Module II. Sensory events 2 are more "exteroceptive" in nature, a sensation of the environment which is not necessarily directly dependent on current module actions. For this diagram to function, events within a particular class (1, 2, or 3) are mutually exclusive.

At the first level, on the left of Figure 4-6, are "input sensory events". These are events that can be sensed by the system; generally it may be useful to include higher-order events that are sensed through multiple actual sensors here. They might be associated with signals mostly internal to the system, such as joint angle. They might also be events signifying some interaction with the environment, such as foot contact, or even a non-interactive environmental event such as visual detection of an obstacle. In order to represent more complex sensory events, these events may be combined in the event compounding stage. So far we have only used Boolean combinations at this level, but one might imagine a more general weighted average or neural network implementation of this stage. Any sensory event which promotes switching between actions at a module is a

"switching event", whether raw "input" or compounded. In Figure 4-6, all the arrows that cross the line between SCASM event compounding and the modules are switching events. For example, input events 2A and 3A are compounded by a Boolean AND to generate the switching event which promotes switching from Action 1 to Action 3 in Module I.

To provide an example flow of information, suppose that the system is doing actions I-1 and II-1. This would tend to produce sensory event 3A; at some point following this say we also have the external sensory event 2A. These two events combine to complete the switching event causing Module 1 to go from action 1 to 3. Action I-3 would tend to produce input event 1C, which is sufficient to trigger the action switching event from II-1 to II-2. Action II-2 generates sensory event 3B, which then promotes switching from I-3 to I-1. Action I-1 will then generate sensory event 1A, which is the switching event from II-2 to II-1. At this point we have completed a cycle of behavior, and are back in actions I-1 and II-1.

Note that without some idea of the system being controlled in this manner, we have no idea how long it will take for each action to lead to the following sensory event, or how reliably they will do so. It could take milliseconds, hours or years, and it might have a tendency to get stuck in global action [I-2, II-1] when the system fails to either produce or detect event 1B.

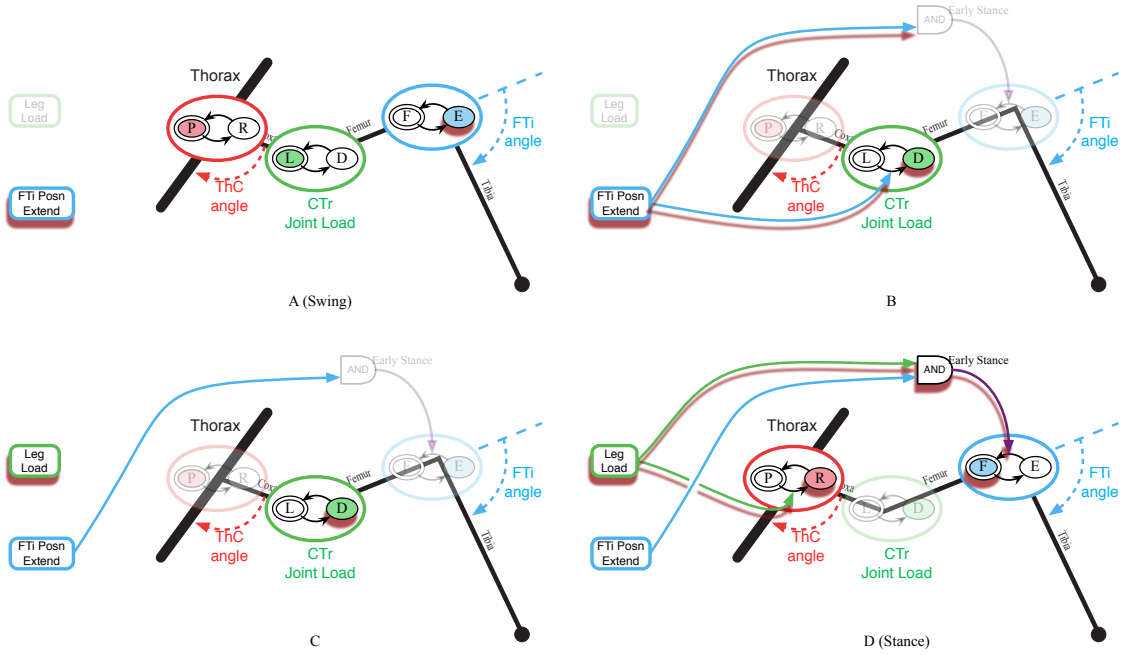


Figure 4-7 Swing-Stance events and action switching for stick insect middle leg forward stepping. A) In swing, ThC is in protraction, CTr in levation and FTi in extension. FTi extension leads to the "FTi Posn Extend" sensory event. B) "FTi Posn Extend" then directly causes transition from levation to depression at CTr, and fulfills part of the "Early Stance" compound event. C) CTr depression then leads to the "Leg Load" sensory event. D) "Leg Load" then directly causes transition from protraction to retraction at ThC, and completes the "Early Stance" compound event which causes transition from FTi extension to flexion. The leg is now in retraction, depression and flexion; this is full stance.

A graphical representation of the swing-stance transition in the stick insect system is shown in the partial event space diagrams in Figure 4-7. It can be seen that in this control arrangement this involves the interaction of a number of different signals and system components, rather than flipping a central switch from "swing" to "stance". This figure also provides an example of how the event space diagram may be used to predict basic system behavior; this same process can be done on the complete event space diagram.

4.2.2 Relationship with finite state machines

In the case where switching between actions is an instantaneous transition, the action switching module can be considered a finite state machine as described in section 3.6.2,

with each action considered a state. The transition function is implemented by the process(es) that detect switching events, with the possible addition of a switching timer internal to the module. The input alphabet depends a little on where you draw the system boundary for the module; it can consist of input sensory events or the already-compounded switching events. The start state would be one of the actions; so far the legged systems tested here are relatively insensitive to start state.

More accurately in this case, each module is a hybrid dynamic system, with the switching rules representing the finite state machine portion of the system, and the physics of the robot or animal and environment comprising the continuous dynamic portion. Depending on the level of analysis, action dynamics (e.g. muscle models) calculated in a digital computer and periodic servo control loops might be considered either part of the discrete event or continuous dynamic portions of the hybrid dynamic system.

Since we might consider the action switching as a state machine, one might ask the question: why not use the language of finite state machines to represent these systems, rather than inventing a new one? There are two answers to this question: the SCASM representation can have advantages over a traditional finite state machine representation, and the action switching may not be discrete.

4.2.2.1 Advantages of the SCASM representation

Even for cases where the switching between actions can be fully described using finite state machines, the SCASM representation can have some specific advantages. In this case SCASM is just one way of representing the collection of state machines associated with the action switching modules. It is also possible to combine these state machines

into a monolithic "global" state machine with states consisting of the combinations of the states of each module. An example of the global state machine for the stick insect is shown in Figure 4-8; the global transition events are not shown, however, since the full set cannot be concisely represented in this form. This is particularly true for systems with more degrees of freedom, since the number of possible global states (whether all are reached or not) is the product of the number of actions in each module.

There are uses for global state diagrams as well. Though the structural meaning of the transitions in the global state diagram is not easy to discern, they can be a useful way to represent the global *behavior* of a SCASM system. This diagram can be more descriptive than an event-space or other modular SCASM representation in representing the global behavior of the system, at least under standard or expected conditions. This can be particularly useful in the design of a SCASM control network, as will be demonstrated below in section 4.3.

It should be noted that unexpected sensory input might cause the global state progression to deviate from the usual one marked in Figure 4-8. This diagram might be considered a behavioral attractor, but not a full description of the behavioral state progression under all environmental inputs. It may be possible to construct such a full descriptor in such a diagram, but if so it would pertain only to a specific range of transition rules and sensory event thresholds. Also, for cases when certain sensory events are detected at different thresholds by different modules, or sensed by different sensors, the state progression will have more steps in it and may have several alternative routes through global state space.

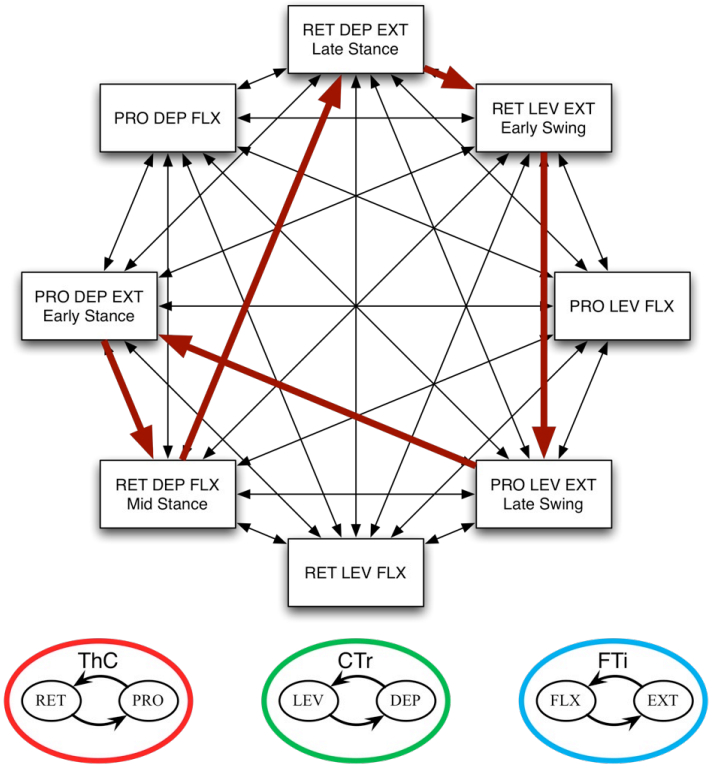


Figure 4-8 Global state diagram for stick insect middle leg forward stepping. The three module state machines are shown below to aid in making sense of the global states. The global state progression shown is the nominal progression for forward stepping.

The primary advantages of a SCASM representation when compared to a global state representation are as follows:

1) Biological hypothesis representation.

Since this dissertation is primarily about modeling: Though the representation is somewhat abstract and general (in the (Webb, 2001) senses), the mapping from elements in the SCASM system representation to the associated elements in the animal is straightforward. Each joint action switching module represents a joint pattern generator, and each switching event represents a combination of thresholds of sensory signals which is related to that event.

In the global state diagram, the mapping to joint pattern generators is not explicit, and the transitions between global states may be associated with a number of different possible sensory states. There are no lines indicating the flow of sensory information within the system.

2) Structural relevance & modularity

The association of each action switching module with some DOF or subsystem leads to a clear and explicit conceptual relationship between the controller and the physical plant. This also lends itself well to physical modularity of control.

3) Plasticity

Since the flow of control information is explicit in SCASM and associated with the transition influences on the modules, this provides a clear place for exterior processes to modify system behavior by changing these influences. Such exterior influences could, for example, be from higher-level influences or coordinating systems. The global state diagram makes changes in the progression of overall system behavior easy to see, but there is no expression of underlying mechanisms (unless it were actually a global state change control mechanism). Additionally, in SCASM the placement of parameters that could change the dynamics of any particular action is easier to see, and need not be changed or explicitly linked in the multiple global states (e.g. changing muscle activations in FTi Flexion only affects one action in SCASM, but has effects on global system dynamics in half of all global system states)

4) It's in the name.

The phrase "Finite State Machine" in connection with biological modeling will automatically mislead some people, partly due to historical considerations. Language and communication is important, particularly in interdisciplinary work such as this. There is something to be said for choosing terminology that doesn't carry incorrect historical connotations.

4.2.2.2 Module actions may not be discrete

There may be cases (particularly in biological systems) where the switching between actions is not actually discrete, and therefore the module is not really a finite state machine. The behavior of such cases might nevertheless be described and understood using the SCASM system representation.

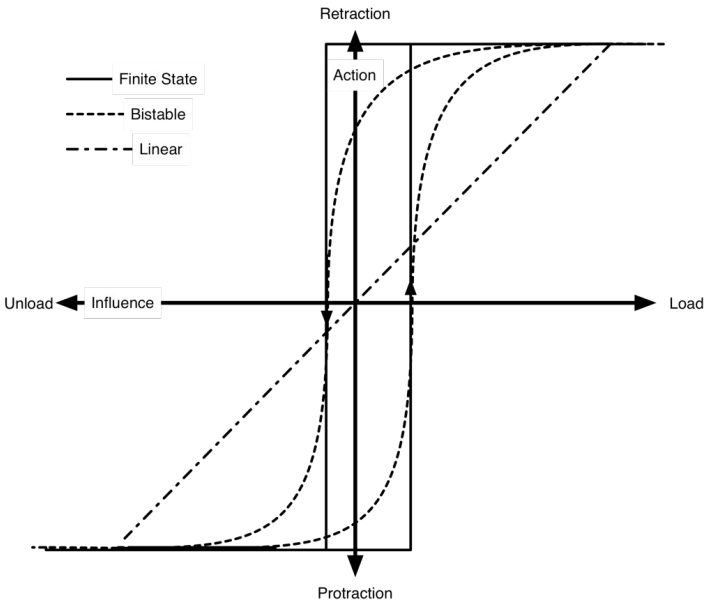


Figure 4-9 Examples of action switching rules which might be used at a module with two actions. A finite state switcher goes directly from one state to the other at thresholds of influence, which may be different. A more general bi-stable switcher might look similar to the finite-state case, or it may differ significantly within the switching region(s). A linear "switcher" would have behavior quite different from the others within the "switching region", but could still be treated as switching actions if the switching influence goes far enough towards its extremes.

One could make the argument that if a system is operating in a way that could be described as SCASM, it can then be modeled as finite state machines. This may be; however a SCASM system is arranged so that smooth transitions between actions can in some cases have the same effect. This will at least not be as easy in a global state representation/implementation, and might not work at all. For a three-DOF leg with two actions per module, the neural-level dynamics might be represented by three graphs similar to Figure 4-9. A higher input dimension might be necessary if more sensory signals are used for input; e.g. the stick insect FTi joint would have leg load (measured as CTr load in this example) and FTi position input influence dimensions, as shown in Figure 4-10. In a monolithic machine the single graph would need to be represented in at least three input dimensions and one output dimension; however the meaning of a single output dimension for all eight global states is not easily made sensible. Which global states should be made adjacent, and which should be at the extremes? One might then spend effort decomposing that space into lower-dimension manifolds associated with particular inputs– SCASM provides one such representation that is directly related to the system's mechanical structure and based on mechanisms that generate insect behavior. Also, if an *implementation* is organized as SCASM rather than monolithic, it can be relatively simple to swap between discrete & continuous influence-action relationships, and this might furthermore be done on a module-by-module basis.

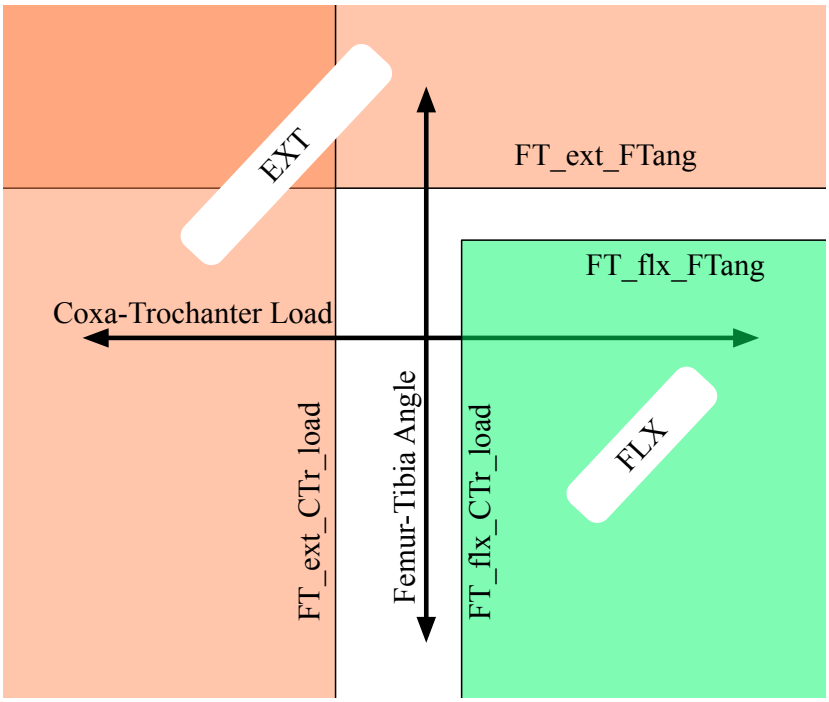


Figure 4-10 The discrete action switching rule for the femur-tibia action switching module in the stick insect for forward walking. A load greater than FT_flx_CTr_load ("leg load") AND an angle less than FT_flx_FTang ("FTi Posn Extend") will cause transition to the FLX action.

4.2.2.3 Comparison with Subsumption Architecture and Genghis

For the legged system case, each action in a SCASM module is considerably more complex than each state in the state machines of the Genghis controller (Brooks, 1989); about 50 of the state machines in that controller (see Figure 3-7) would be replaced by a set of 18 action switching modules in hierarchy; 2 at each leg for controlling the joints, and another at each leg for the gait coordination, with actions of "stance" and "swing". Though it was a leap in the capability of reactive robotics, there are still a lot of "engineering mindset" elements in the Genghis controller, in terms of the low-level control being position oriented, and the way the state machines are organized. Rather than starting in control design with the parts of a single leg, they start with the control and balance of the beta (levation/depression) DOFs of all legs simultaneously. In this kind of

organization the extent to which higher levels of control can "ignore" lower level problems is limited; the subsumption architecture has higher levels *taking over* local control in specific circumstances (through the suppression mechanism), rather than providing modifications/inputs to the local control. Though it would probably be possible to redesign that controller using a force-based output to be more fluidly adaptable to changing conditions, this method of using subsumption architecture limits the amount of abstraction possible in higher levels of control; since the higher levels "take over" lower level systems, they must also enact some of the lower level control, using some of the same inputs and outputs.

4.3 The SCASM Command and Configuration Interface Language Specification

The SCASM Command and Configuration Interface Language Specification (SCCILS) is an XML specification that formally defines a language for defining and controlling SCASM-controlled systems. While definitely still in development and pre-alpha in terms of maturity, the current implementation (released as version 0.1 in Appendix 10.2) was used for all of the experiments in chapter 7 that include a change in speed or type of behavior.

As discussed in 2.1, there are several model-related elements that can significantly increase model usefulness. In the Webb framework, SCASM itself is a concept for describing hypothetical mechanism, which can be expressed at various degrees of *level*, *generality* & *abstraction* as defined in the dimensional model description of (Webb, 2001), as demonstrated in the various system description diagrams presented in this chapter.

The SCASM representation has a number of parameters that can act as "ports" for changing the system; in modeling this could be described as changing the *hypothetical mechanism* being represented; in any usage the effect can be to change the behavior.

SCCILS makes use of these ports by developing an explicit representation language. Making a model's *representation* as explicit as possible can be a useful exercise in itself, but SCCILS was first developed with the following benefits of a unified/general standard representation of SCASM systems in mind:

a. One set of tools can be developed for operating (*demonstrating)*, configuring (*representing)*, and data representation and interpretation (*interpreting* and *deriving)* within the modeling framework of SCASM systems.

b. Unified tools for operating a model (*demonstrating)* might allow:

    i. Use of the same user interface for multiple models (each with specific benefits and weaknesses), or use of multiple interfaces (each with a specific purpose) to run a model

    ii. representational/experimental portability. one might run the exact same experiment on the target and multiple models, each of which might occupy a different location in the dimensional model description.

SCCILS was developed to address these problems, and become an integral part of the work addressing elements 2 and 3 of model usage introduced in section 2.1. The idea of using one interface to operate different models was present from the very beginning of the

wok, from my first meeting with Marcus Blümel discussing using the controller, previously used in simulation, in a robot. Initial brainstorming for the actual implementation of SCCILS was done with Marcus Blümel and Arndt von Twickel; we wanted to be able to control multiple models without rewriting the user interface. Furthermore, we wanted to be able to run exactly the same experiment in multiple models with minimal fuss.

There are a number of reasons one might want to use multiple models to address a question. In a computer simulation one can make a good model, but it can be very hard to do the physics well; this carries both the blessing and the curse of having total control and responsibility for the physics of the world. In a robotic model, you get the real world for free. In particular, modeling interaction with the environment is often the most difficult task, and such models can behave badly in situations where there is rarely a problem in the real world. Any time the model is touching something, this advantage is in action– and interacting with the ground is most of what is behaviorally relevant about walking.

For insect modeling there can be some specific advantages of computational simulation. Very advanced muscle models can work better or more directly in simulation, since this environment allows direct force output. A better match of the model system dynamics to the target is also possible, since the simulation can be made any size. In the insect gravity and inertia are much lesser influences than for humans, and can in some cases be ignored (Hooper, et al., 2009). Both of these are important in robot dynamics, particularly with the reflected inertia of electrical motors with transmissions.

So, you might want to run an experiment in both (or several) models, to see which of these things might be important for a particular behavior or experimental regime. Having same interface would be REALLY helpful to the model users.

4.3.1 The structure of a SCCILS-capable system

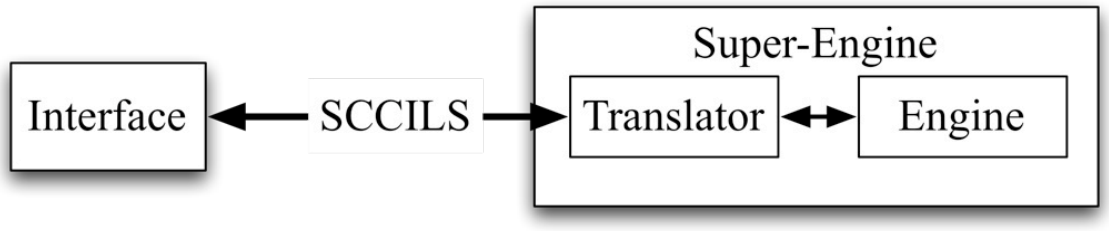


Figure 4-11 The basic layout of a system using SCCILS. "Engine" is "The model"- robot & dynamic control software, or simulation. Once configured the engine can operate by itself. Interface- something which sends configuration & command information.

The structure of a system using SCCILS online has two basic elements which communicate using SCCILS, the "interface" and the "engine", as shown in Figure 4-11. The engine is the entity which is actually *simulating* in the "simulation model" of Figure 2-1. This could for example be the physical robot and control software, or a computational simulation. SCCILS is used for configuration and control of the engine, and contains methods for data handling as well. An interface-less engine might still use SCCILS for configuration and experiment scripting.

The interface is something that sends these configuration and control commands, and might want data in return. It could be a graphical user interface (GUI) as we're doing, or a higher-level control program, or even a program that could run optimizations or higher-level experiments.

The translator is an element that takes in the SCCILS XML commands and translates them into the native representation of the engine used directly in control. It is possible that a program implementing an engine might also contain translator code, or they might be separate programs. Most of the data that can be passed between the interface and the

translator is stored in the form of Handy Data Storage Objects (HDSOs), variables that may store various types of named, multi-dimensional data.

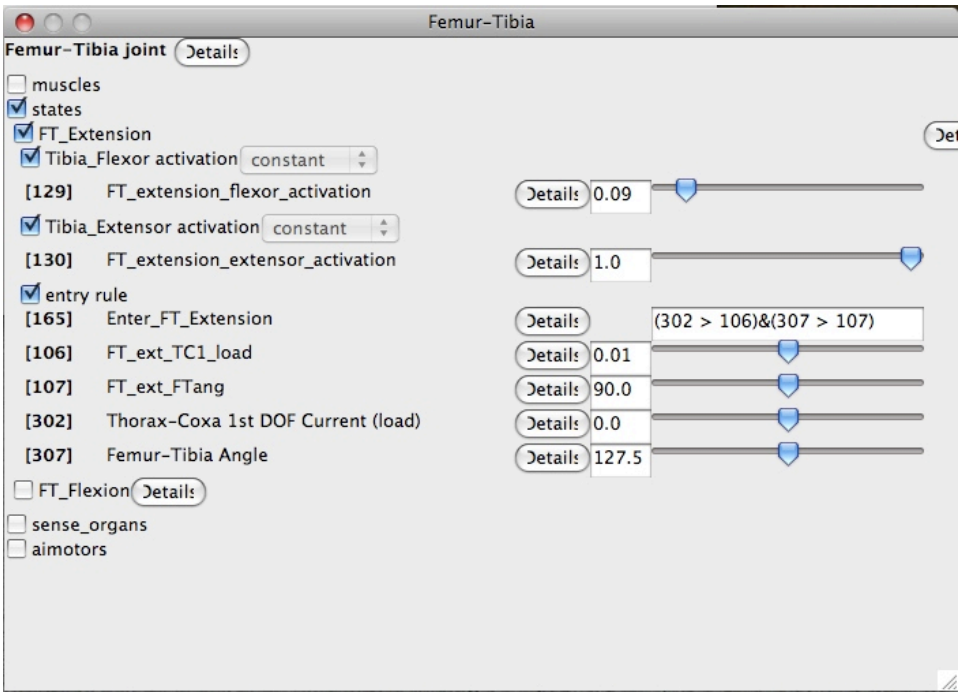


Figure 4-12 Romod_GUI representation of elements associated with the FTi Extension action. Handy Data Storage Object (HDSO) elements 129 and 130 are muscle activations; a part of the dynamics of this action, which may be modified online. 165 is the Boolean entry rule Enter_FT_Extension; its arguments are the IDs of the HDSOs shown below it. It says ((ThC1 load > FT_EXT_TC1_load) AND (FTi angle > FT_EXT_FTang)). The parameter HDSOs (106 & 107) below may be modified in order to change the strength/thresholds of those influences. Sensor HDSOs report robot state. Modification of transition parameters or activation parameters can drastically affect system performance, but the entry rule is what determines the connections in the sensory flow and event space diagrams, and defines the nominal behavioral progression in the global state diagram. The conceptual location of the HDSO parameters here are shown in Figure 4-25.

The concepts of SCASM and SCCILS are not tied to implementation in legged systems, but since the NeuRoMod control code was originally developed for such systems *before* the development of SCCILS, there are still some portions of that code that have not been generalized, and assume implementation in legs.

The NeuRoMod software (manual in Appendix 10.4) contains a SCCILS graphical user interface written by John Bender. This interface is capable of modifying any SCCILS-aware parameters in the engine it connects to, and automatically arranges its layout based

on the configuration of that engine. Examples of usage of this UI are given in the NeuRoMod manual, and Figure 4-12 shows the representation of parameters associated with a particular action of the cockroach forward walking configuration.

4.3.2 The Configuration/Data Type Sublanguage

The jointlist is the root element of a SCCILS model description file, and defines at least one leg as a chain of joints. These contain or reference all the data needed to define the interface/engine communication in SCCILS for a particular model. A jointlist describes a SCASM system to the level of an "implementation-specific sensory flow diagram", representing both the flow of sensory signals represented in an event-space diagram and the thresholds and action parameters necessary for a full specification. Numerous parameters may be modified online, but the available selection of modules, actions and sensors is constant during system operation.

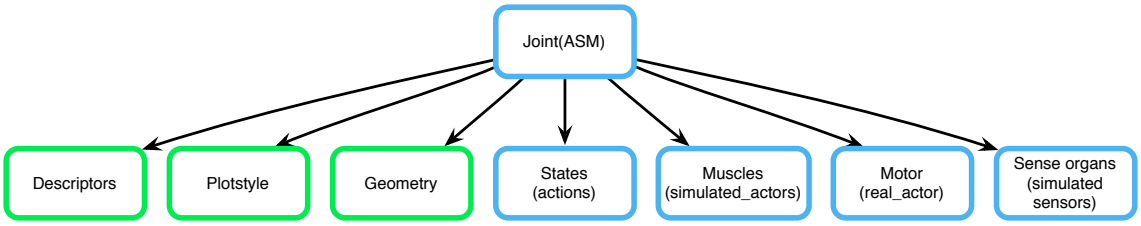


Figure 4-13 The contents of a SCCILS joint. The green elements are useful for user interfaces and data handling, but are not descriptors of the SCASM action switching module represented by the joint.

During the initial development of the SCCILS concept, we discovered that much of the data we wanted to pass between the interface and the engine was in a similar form. Handy Data Structure Objects (HDSOs) are really for data storage, but there are a number of things within them for making modification & UI representation more easily. Anything represented as an HDSO can theoretically be modified online or changed as

part of an experimental script, though the engine is not necessarily required to be able to implement all HDSO changes online. These modifications are accomplished using an HDSOscript, which is a timed list of HDSO data modifications. Most of the SCCILS commands are in the form of an HDSOscript.

The full configuration of a SCASM system is currently specified by three XML files: the jointlist, a list of parameter HDSOs, and a list of sensor HDSOs.

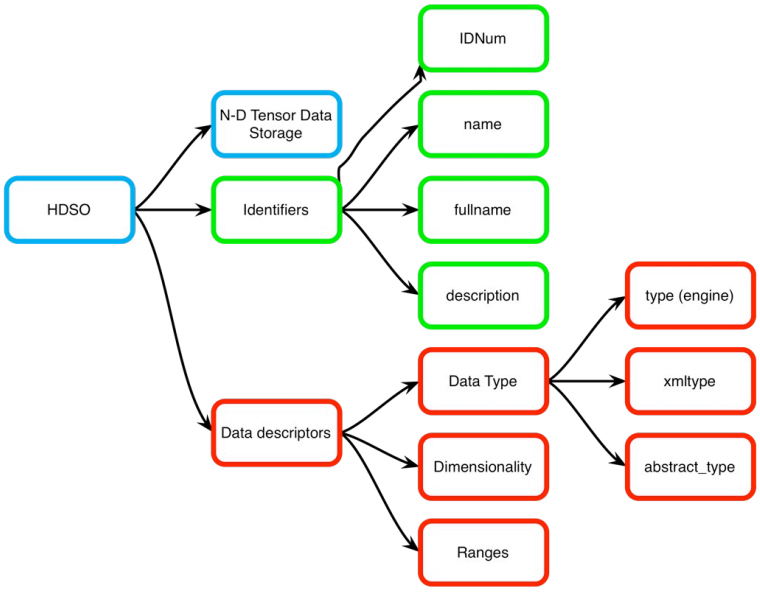


Figure 4-14 The Handy Data Storage Object; an N-dimensional data store with a set of identifiers and descriptors.

4.3.3 The Command Sublanguage

The SCCILS command sublanguage is a fairly simple set of elements: the sccils_command, sccils_command_response, and sccils_config_command. The sccils_command allows run, stop, data logging configuration, and extended commands. In order to be considered SCCILS-capable, an engine or interface must implement and respond properly to the run and stop commands, at the very least. A sccils_command may contain an HDSOscript for specification of HDSO data having to do with an extended

command. The sccils_command_response is used for acknowledging both command and configuration messages, and reporting whether these messages were received and implemented successfully.

The sccils_config_command is used to communicate which SCCILS configuration to use (if several are available to the engine), as well as to communicate configuration data having to do with the configuration of the SCCILS XML communication itself, such as setting TCP port numbers for data streaming.

4.4 Examples: Synthesis and Restructuring

4.4.1 Synthesis: The minimal SCASM stepper.

"…What's still missing in terms of robotics is design methodologies. How to do it properly; how can you get the system producing the right behavior…" – Auke Ijspeert in (Waibel, Ijspeert, Hauert, et al., 2007). Inspiration from biology can be an effective way to gain agility and efficiency. The design of novel controllers using these tools remains a difficult problem, however; see discussion in (Ijspeert, 2008; Ijspeert, et al., 2007). The synthesis method demonstrated here could be useful both for the exploration of biological hypotheses and for the development of novel robot controllers that make use of the neuromechanical concepts embedded in SCASM.

This section presents a set of steps that may be used to specify sensory influences which, when combined with appropriate action dynamics, generate a desired behavior for a SCASM system. It does not address in detail how action dynamics should be specified, which does leave considerable design work left to do. The work on muscle models in chapter 7 addresses some of the effects that simple changes in action dynamics can have on global system behavior, and as suggested there it is likely that actions which exhibit

some sort of output saturation will lead to more stable and better-conditioned overall behavior. In any case, work so far has shown that with the proper sensory influences specified, reasonable desired behavior can often be achieved with extremely simple action dynamics.

For this exercise we will use the minimal leg-like mechanism capable of producing forward or reverse motion. While we have used this process for 3-DOF robot legs, and it can be used for higher dimensions, we present the minimal case here to support brevity and conceptual clarity.

This minimal stepper has a leg with two degrees of freedom: protraction & retraction at a "Thorax-Coxa" (ThC) joint for forward/backward movement of the leg, and levation & depression at a "Coxa-Trochanter" (CTr) joint for picking the foot up and putting it down. These are the proximal two degrees of freedom for the stick insect, and also the two degrees of freedom used in the single-leg stepper described in (Beer & Gallagher, 1992). The two degrees of freedom of this minimal leg stepper are shown in Figure 4-15.

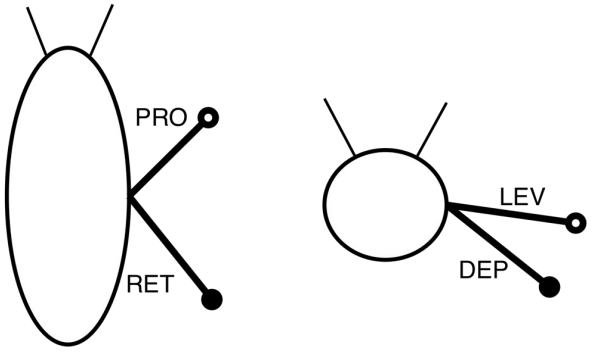


Figure 4-15 The minimal stepper; top view on the left showing the protraction-retraction DOF. Rear view on the right showing the levation-depression DOF. The open-circle foot shows the protracted, levated leg at the end of swing (for forward stepping); closed-circle shows the retracted-depressed leg at the end of stance.

Synthesis of a SCASM controller can be accomplished by following the set of steps listed below:

1) **Structural decomposition rule**: Choose modules which represent the physical system, as suggested in (Büschges, 2005). A SCASM controller takes account of physical plant mechanics implicitly, largely through the choice of modules, actions, and sensory events. As is always the case, behavioral success depends on the proper combination of controller and plant dynamics. Since coordination is generated by sensory-mechanical signals, the line between "controller" and "plant" is blurry here. Although different from the classical engineering approach, this is not necessarily bad and more accurately represents the way in which these problems are addressed in animals.

   In simple systems start with one module per DOF. If the system is sufficiently complex that this results in so many modules that later synthesis steps are too difficult, choose subsystems to be associated with the modules. For instance, it *may* be useful to use one module for a multi-DOF joint under some circumstances. Also keep in mind that it may be best to organize the control of a more complex system hierarchically, since an “action” of a higher-level SCASM system can itself be an influence which changes the configuration or triggers a state change in a lower-level SCASM system. For gait control, one higher-level module per leg, used to modify lower-level sensory thresholds triggering swing or stance, could work.

   In this simple example, we choose one module per DOF; ThC for the forward/backward motion of the leg, and CTr for the upward-downward motion.

2) **Start with minimal set of actions**. For each module, try to choose as few simple actions as possible to control the associated system subset. It is probably a good idea to concentrate on "few" to begin with; two is a good goal for a single-DOF subsystem. One can often generate more complex behavior by either adding more actions or making the dynamics of already-existing actions more complex; see for example the piecewise-constant muscle model in chapter 7. It is often easier to add more actions, but this is likely to lead to more confusing event-space and global state diagrams, and may lead to more free parameters overall when compared to an increase in complexity of one or more actions. In this example it is natural to choose Protraction and Retraction as the actions for the ThC module, and Levation and Depression for the CTr. These modules are shown at the bottom of Figure 4-17.

3) **Start with simple sensory events** which sense the limits of these actions; i.e. sensory events that these actions should eventually produce during the desired overall behavioral cycle. Some potential events for two of the example actions are shown in Figure 4-16. In many cases, you will get better performance by choosing limit events that are *related to interaction with the environment*. These will lead to some degree of automatic sensory-driven adaptation of system behavior to environmental variation. The minimal set of sensory events, though, should consist of things that are expected to happen in every behavioral cycle. With this in mind, choose "ground contact" for depression. Obstacle collision is not something we expect to happen every step, so for protraction let's instead choose "Protracted angle". For the other actions in these modules we'll start with the

opposing sensory events, “no ground contact” for levation and “retracted angle” for retraction.

In the controller implementation associated with this design step, one must choose specific sensory conditions, usually thresholds operating on raw or processed sensory data, that represent these sensory events. To avoid undefined or erratic behavior, the real sensory conditions associated with the switching events for a module must not overlap one another, and should probably have a deadband in sensory space which does not elicit action switching. An appropriate example is shown in Figure 4-10. Additionally, the deadband should not be so large that there is an unnecessary risk of “missing” a state transition. Plotting the desired or actual sensory state trajectory on a transition rule plot such as Figure 4-10 can be a valuable tool in designing, modifying and debugging sensory event definitions.

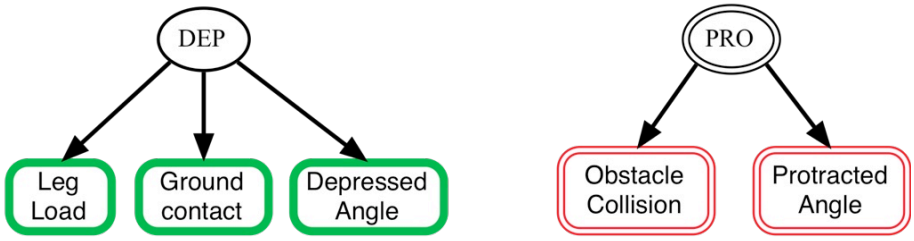


Figure 4-16 Some possible sensory events that could result from the depression action at the CTr joint or the protraction action at the ThC joint.

4) **Construct global state diagram(s)**, using the combinations of module actions. If there are too many modules for this to be easy (this probably means more than 3), either start by making a sub-global state diagram using three modules most likely to be important, or make a diagram using only the global states nominally involved in the desired behavior. It might also be useful to choose a decomposition based on one module that is highly involved with global state

description. For a 4-module leg, choose to decompose it into two 3-module systems; one in which the fourth module is in the "stance" state, and another with it in "swing". **Describe and label these global states**, or at least the global states you want to be in the target behavior. This has been done in Figure 4-17.

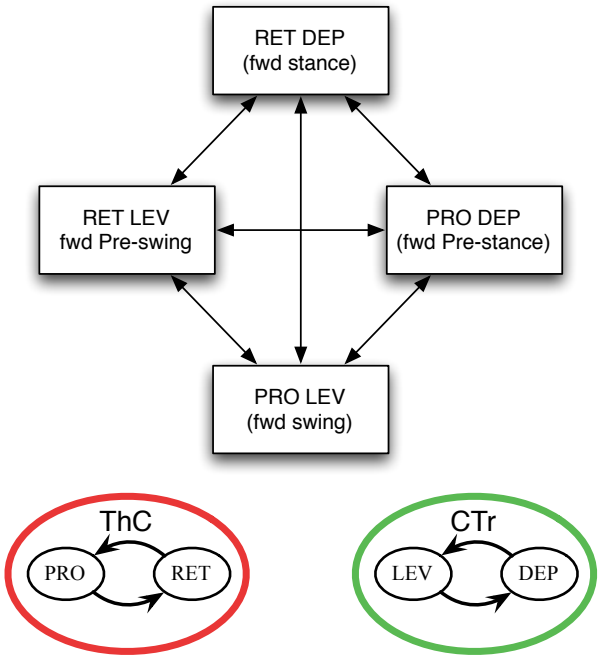


Figure 4-17 Global state diagram for minimal SCASM stepper

5) **Choose a desired progression of global states**. This is done for forward stepping in Figure 4-18. To some extent, the use of transient global states, such as pre-stance and pre-swing, may represent both a more sensory-driven behavior and a more robust system overall. Choosing only "swing" and "stance" both represents something like central coupling between the modules, and ignores the complexity associated with these transitions, thereby perhaps causing the system to deal less well with, e.g., changes in substrate height. One way to express this directive is to try to choose global state progressions in which each step changes the fewest

actions possible– preferably just one at a time. This may give you some transient global states that don't seem inherently useful, but is likely to make the system behavior more robust and adaptive.

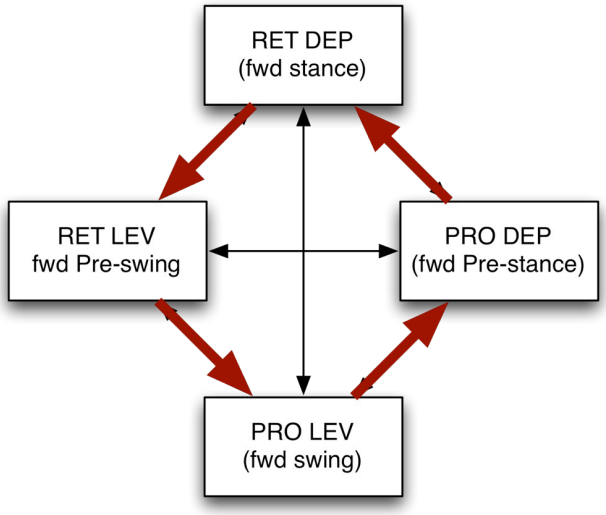


Figure 4-18 Global state progression for forward stepping, minimal SCASM stepper.

6) **Construct the event space diagram**. Start in one of the desired global states; preferably a non-transient one (i.e., one which takes a large percentage of overall behavioral cycle time). Identify the limit sensory events this state will lead to. Choose events (ideally at least one that is external or from a module that does *not* change actions in the next global state) to trigger the appropriate action switching to reach the next global state. Iterate until finished, as shown in Figure 4-19 and described below.

Start in stance. Next state changes from DEP to LEV at CTr, choose the ThC signal being generated in stance (RET-> ThC Retracted) and have it promote CTr Lev.

(line 1)

Now in pre-swing, next state changes RET to PRO at ThC. Choose CTr signal being generated in Pre-swing (No Ground Contact) and have it promote ThC PRO

(line 2)

Now in swing. Next state changes LEV to DEP at CTr, choose ThC signal being generated in swing (PRO-> ThC Protracted) and have it promote CTr DEP

(line 3)

Now in pre-stance. Next state changes PRO to RET at ThC, choose CTr signal being generated in pre-stance (DEP -> Ground Contact) and have it promote ThC Ret.

(line 4)

Now we're back in stance, and done.

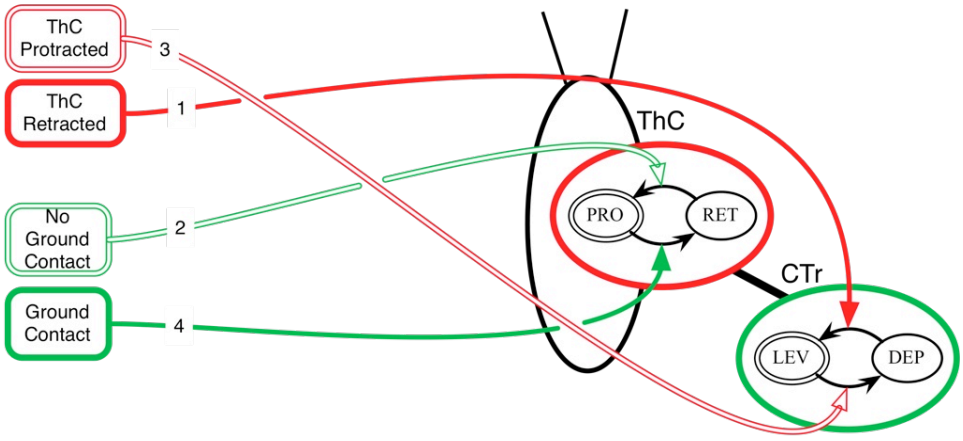


Figure 4-19 Event space diagram for forward stepping in the minimal SCASM stepper. The switching event lines are labeled according to the construction steps in the text.

Cross-module events (events which promote switching in modules different from the module(s) causing the event) are the ones that actually generate the sensory coupling, and

are likely to give rise to more robust behavior. Compounded events may be more robust if properly configured, but the higher input dimensionality makes it easier to accidentally overlap transition areas in sensory space, which would produce undefined state switching behavior. This would correspond, for example, to an overlap of the FLX and EXT areas in Figure 4-10, which would happen automatically if the FLX rule used OR instead of AND.

An example SCCILS configuration for this minimal SCASM stepper is given in the *lin_stickmiddle_minimal* example in the SCCILS 0.1 specification document in section 10.2 of the appendices.

4.4.2 System restructuring: robot to robot.

Any major modification of the structure of a SCASM system will require steps similar to some subset of those used in synthesis. For example, we had a working, neurally-based 3-module system for controlling a 3-DOF stick insect model leg, and wanted to apply this as a hypothetical controller for a (reduced actuation) 3-DOF cockroach leg controller with as little modification as possible. The restructuring here is somewhat constrained since it is between robotic model systems, particularly with respect to the sensory signals available. The ambiguities of such model signals are addressed in section 4.5, particularly in Figure 4-21. In the type of modification where a system is to be applied to a substantially different physical system, the primary task is to locate the modules and sensors such that they match function in the new system as closely as possible with the source system.

The event space diagram for stick insect forward walking is shown in Figure 4-4. The primary kinematic roles of the joint degrees of freedom in this system are as follows:

• Thorax-Coxa (ThC): protraction & retraction of leg.

• Coxa-Trochanter (CTr): levation & depression of leg.

• Femur-Tibia (FTi): ab/adduction of foot.

The roles of the sensors are:

• ThC angle: protraction & retraction for triggering transfer to swing.

• CTr load: leg load for detection of ground contact.

• FTi angle: foot position, important in swing/stance transitions.

The anatomy of the cockroach mesothoracic leg is substantially different from the stick insect, particularly in the kinematic role of the joints, as is described in chapter 5. The event space diagram for cockroach forward stepping shown in Figure 4-20. The primary kinematic roles of the joint degrees of freedom in the cockroach leg are as follows:

• Thorax-Coxa 1 (ThC1): levation (during ThC1 retraction) & depression (during ThC1 protraction) of leg.

• Thorax-Coxa 2 (ThC2): adduction and abduction of leg.

• Coxa-Trochanter (CTr): protraction (during CTr levation) & retraction (during CTr depression) of leg.

• Trochanter-Femur (TrF): depression (during TrF reduction) and levation (during TrF passive return) of the foot.

• Femur-Tibia (FTi): ab/adduction of foot.

In some representations of these degrees of freedom in the cockroach leg, the actions are double labeled. In these cases the first label is the anatomical term which corresponds to previous work, and the second label describes the effect this motion actually has on the leg. Therefore the ThC1 actions will sometimes be labeled ret(LEV) and pro(DEP) to indicate that the leg is levated during "retraction" and depressed during "protraction". Likewise the CTr actions are sometimes labeled lev(PRO) and dep(RET).

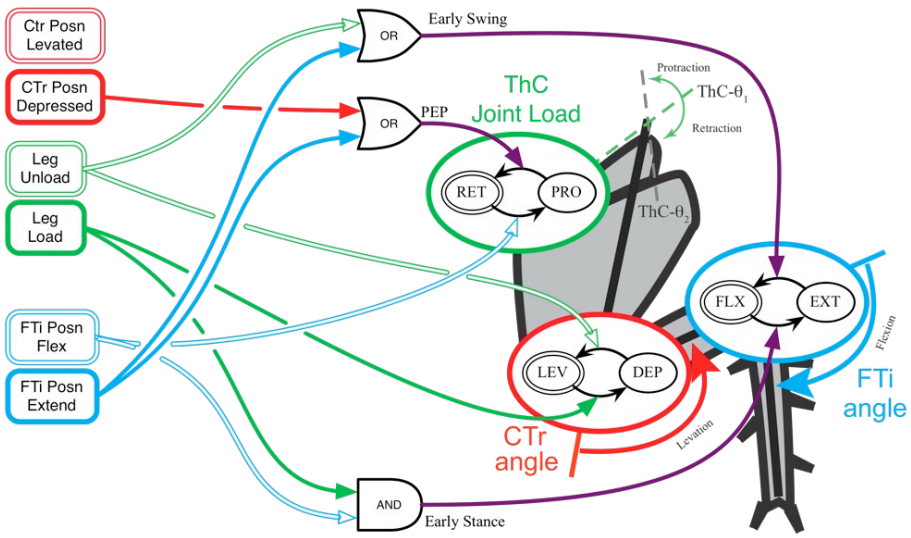


Figure 4-20 Event space diagram for cockroach forward stepping.

Since the ThC2 and TrF degrees of freedom were the least well analyzed and/or hardest to build, the other three were chosen for the initial 3-DOF implementation. It can be seen that the roles of the ThC1 and CTr joints are switched, when compared to the stick insect; so the CTr stick insect module was placed at ThC1, and the stick insect ThC at the cockroach CTr.

The sensor roles in the cockroach robot are as follows:

- ThC load: leg load for detection of ground contact.

• CTr angle: protraction & retraction for triggering transfer to swing.

• FTi angle: foot position, important in swing/stance transitions.

Thus the detecting sensors for the sensory events were also switched. The resulting system is shown in Figure 4-20. One more change was necessary in order to produce forward stepping; due to the different placement of the FTi joint, in this system during forward walking it is most appropriate for it to extend during stance rather than flex. Therefore, the load influences at the FTi module were reversed.

This SCASM system represents an entire set of hypotheses regarding the generation of leg movement in the cockroach; the process of extracting and describing these hypotheses is described in section 4.5.3.

4.5 Translating between systems and representations

4.5.1 Methods of representation

As mentioned in the introduction section 2.1, one important element of model usage is the maintenance of a mapping between the model and the target system, supporting the *theorizing* and *representing* processes in Figure 2-1. Various tools for representing SCASM systems have been introduced, this section presents two examples of using these tools. Each translates between two systems, using several representations to do so at varying levels of accuracy, generality, detail and medium. These include sensory flow diagrams (first introduced above in Figure 4-2), event space diagrams (Figure 4-4), global state diagrams (Figure 4-8), and the SCCILS XML model definitions and associated NeuRoMod UI elements.

There are three types of systems addressed in this work: insect, simulation and robot. The translation between insect and robot is one of the broader stretches, so the examples given are of that mapping. Once the reader understands these examples the more straightforward relationship between simulation and robot, and between different robot models, should be relatively easy to accomplish.

4.5.2 Forward example: stick insect to robot.

This was originally done directly from neural influences to Boolean transition rules (Ekeberg, et al., 2004). This is doable, but lacking in immediate clarity for relation to the modeled system. Also, the Boolean system representation is very specific with respect to the model construction and sensory inputs; a less specific representation may be useful and may more accurately represent our understanding of the "hypothetical mechanism" of the target system.

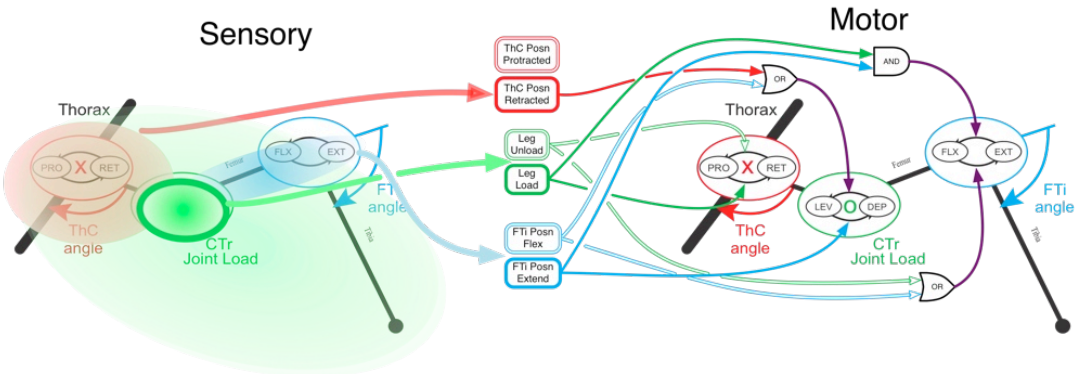


Figure 4-21 Possible vagueness in the relationship between sensors and sensory events in the stick insect. The event space diagram makes the sensory events look discrete and well organized; in the animal they may be transduced by multiple sensors in multiple locations. The "leg load" events in particular might be represented by a multitude of force and contact sensors throughout the leg. The "sensory" leg and the "motor" leg actually represent the same physical leg; they are separated here to reduce the tangling of arrows.

This translation might instead be done with an event-space diagram as an intermediary representation of the hypothetical mechanism; that was indeed the original usage for such

diagrams. As shown in Figure 4-21, each input sensory event may represent a range of actual sensory signals; this abstract symbolic representation is therefore amenable to translating between specific systems and models.

Building a behaviorally functional event space diagram directly from biological knowledge requires a very large amount of said knowledge. In many cases a single line (or maybe a pair of lines) requires an entire set of publishable experiments, as shown in Figure 4-22. The construction of this figure is an example of the *theorizing* process shown in Figure 2-1. This representation is already significantly abstracted from the referenced work; "leg load" refers to signals either from the trochanteral or femoral campaniform sensilla, FTi positions "extend" and "flex" refer to both static thresholds, and combined static threshold and movement (joint velocity) events. An influence of FCO flexion velocity which promotes FTi flexion is not included (as was the case in the simulation), partly due to the lack of good velocity data, and partly due to the fact that that would lead to cases where "flex" would simultaneously promote opposing influences. This would require separating the "flex" sensory event into velocity and position components (Bässler, 1988).

Note: though the input sensory events are a bit fuzzy/general, each *line* can indicate a specific, separate, implementable sensory threshold/event.

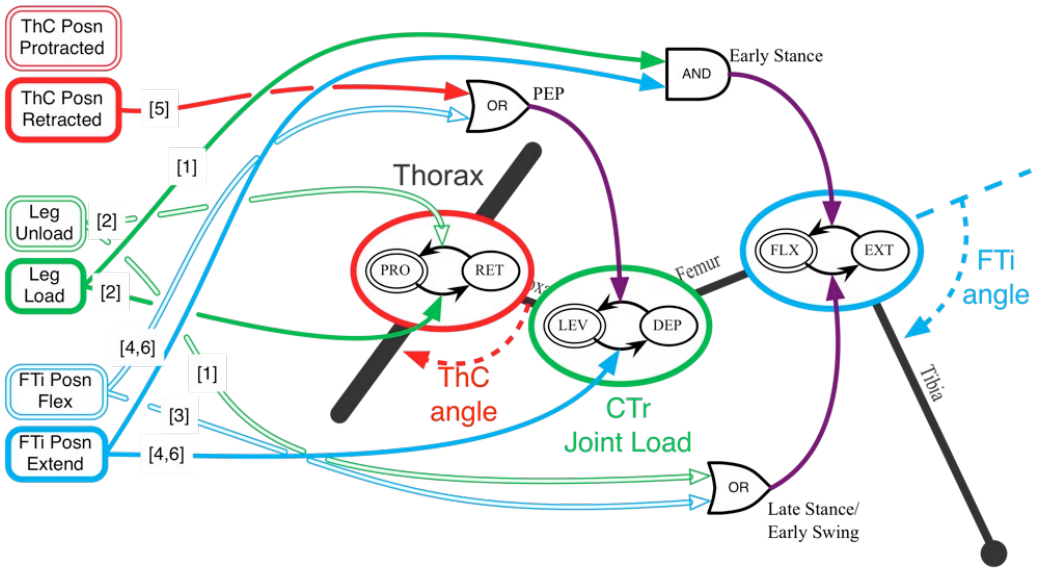


Figure 4-22 Stick insect event-space diagram showing references to the biological work used to construct it. These influences are those described in Table 1 of (Ekeberg, et al., 2004); it does not include all related references in the text, or any of the documented magnitude influences. For complete references, see (Ekeberg, et al., 2004) pp. 288-290. 1) (Akay, Bässler, Gerharz, et al., 2001) 2) (Akay, Haehn, Schmitz, et al., 2004) 3) (Bässler, 1988) 4) (Bucher, et al., 2003) 5) (Cruse, 1985) (behavioral) 6) (Hess & Büschges, 1999).

The representing process of going from the general-input event space diagram to the transition rules used in the robot system requires two steps:

1) Choose the actual sensors used to detect the sensory events.

   In the stick insect robot model, the ThC angle is used to detect ThC Posn Retracted, CTr joint load is used to detect Leg Load and Leg Unload, and FTi angle is used to detect FTi Posn Flex and FTi Posn Extend.

2) Derive the Boolean transition rules using these sensor values and the connections of the event space diagram.

   a. ThC

      i. Protraction condition: Leg Unload. Actual sensor event: CTr load < ThC_Protraction_CTr_load

ii. Retraction condition: Leg Load. Actual sensor event: CTr load > ThC_Retraction_CTr_load

b. CTr

i. Levation condition: (PEP) FTi position flexed or ThC position retracted. Actual sensor event: ((FTi_angle > CTr_Levation_FTi_angle) OR (ThC_angle < CTr_Levation_ThC_angle))

ii. Depression condition: FTi position extended. Actual sensor event: FTi_angle < CTr_Depression_FTi_angle

c. FTi

i. Flexion condition: (early stance) Leg Load and FTi Position Extend. Actual sensor event: ((CTr load > FTi_Flexion_CTr_load) AND (FTi angle < FTi_Flexion_FTi_angle))

ii. Extension condition: (late stance or early swing) Leg Unload or FTi Position Flex. Actual sensor event: ((CTr load < FTi_Extension_CTr_load) OR (FTi angle > FTi_Extension_FTi_angle))

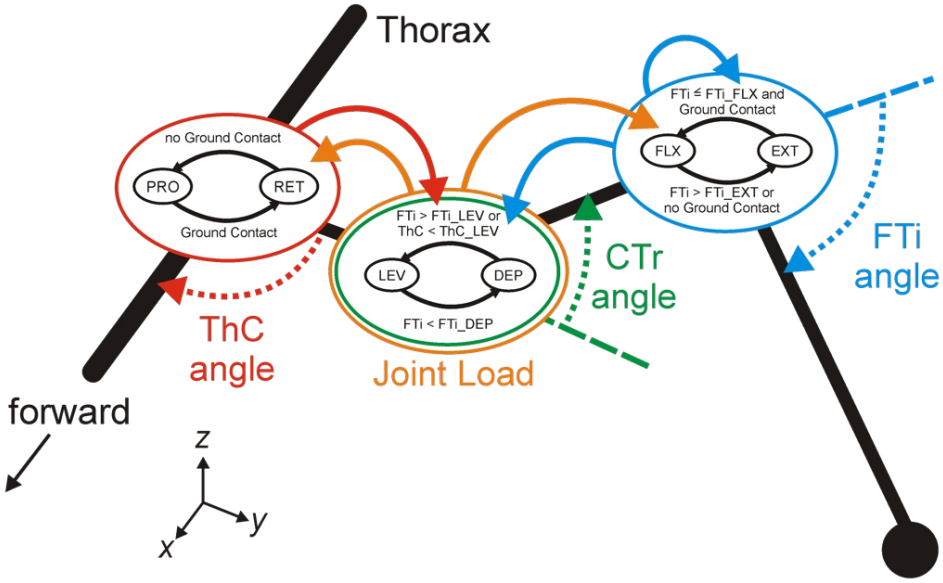


Figure 4-23 Implementation-specific sensory flow diagram for the stick insect middle leg; Boolean switching rules are shown next to the state switch they trigger.

These rules are represented within their respective action switching modules in Figure 4-23. Note that without the Boolean rules, the sensory-flow diagram is a more general *structural* representation than the event-space diagram; It represents the sensory influences between modules, but not the sign or strength of these influence, which may change for different behaviors.

Also, the sensory thresholds and action dynamics must be tuned so that the system will actually function. Some of these parameters can be determined by inspection; e.g. "make PEP here -> CTr_Levation_ThC_angle = -25°" but the entire process is still the subject of some finesse. Though we have not developed a method for translating specific values of neural excitation to specific sensory thresholds or muscle excitation parameters, *changes* in one have a specific representation as changes in the other. For example, an increase in the sensitivity or strength of the leg load influences on promotion of FTi FLX

in the animal would correspond to a smaller FTi_Flexion_CTr_load threshold value, which would lead to this influence being “tripped” at a lower sensory value.

4.5.3 Reverse example: robot to cockroach; forward stepping behavior

In this section we will follow the mapping in the other direction, from the cockroach robotic model to the animal system; this is the reverse pathway from simulation to target system shown in Figure 2-2.

Start with the implementation-specific sensory flow diagram. In the case of a model using SCCILS, this can be derived directly from the XML configuration; the diagram for cockroach forward walking is shown in Figure 4-24. The FTi EXT rules and parameters are also shown in the SCCILS graphical user interface representation in Figure 4-12.

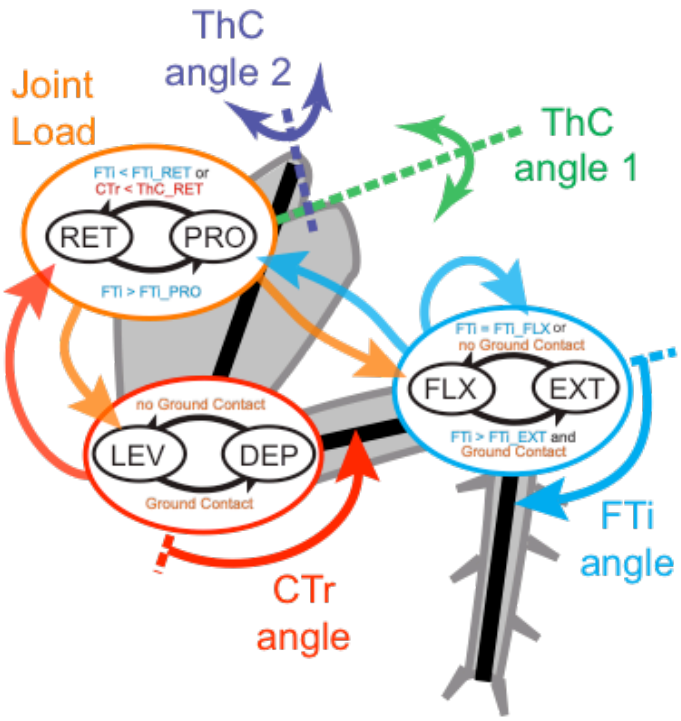


Figure 4-24 Cockroach middle leg forward stepping implementation-specific sensory flow diagram.

This reverse-representation requires the reverse of the steps taken in section 4.5.2.

First, map the sensory specific sensory signals to general sensory input events. There are several influences of the form (FTi > something) or (FTi < something). These correspond to FTi posn. flexed and FTi posn extend, respectively. Likewise, several labeled as “Ground contact” and “no ground contact” but implemented (as seen in the UI) as load influences. Let’s keep them as load: “Leg Load” and “Leg Unload”. Finally, there is a (CTr < something) influence; this represents CTr position depressed.

Then, put these five events on the input event side of the event space diagram, and derive the lines for each of the switching events:

1) ThC retraction (LEV): ((FTi < FTi_RET) OR (CTr < CTr_RET)) Switching event: OR of FTi Posn Ext and CTr Posn Dep; extended leg; PEP
2) ThC protraction (DEP): FTi > FTi_PRO Switching event: FTi Posn Flex
3) CTr levation (PRO): No Ground contact Switching event: Leg Unload
4) CTr Depression (RET): Ground Contact. Switching event: Leg Load
5) FTi Flexion: ((FTi < FTi_FLX) OR (No Ground Contact)) Switching event: OR of FTi Posn Ext and Leg Unload (Early Swing)
6) FTi Extension: ((FTi > FTi_EXT) AND (Ground contact)) Switching event: AND of FTi Posn Flex and Leg Load (Early Stance)

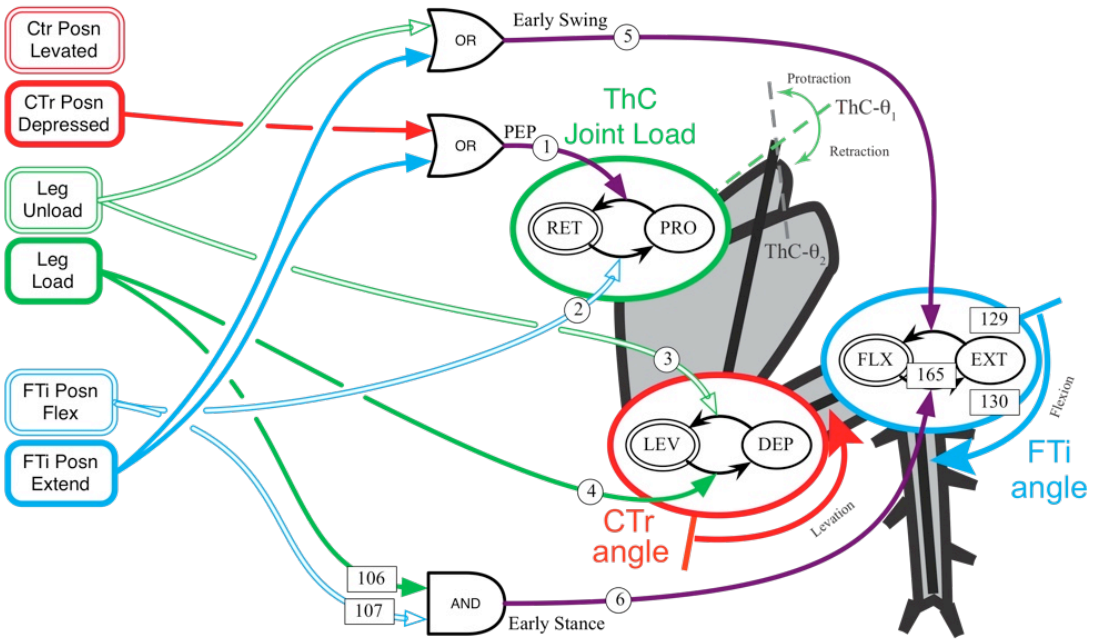


Figure 4-25 Cockroach forward stepping event space diagram. Circled numbers label the lines determined through the mapping step (above). Boxed numbers show the conceptual location of the HDSO parameters shown in the UI screenshot of Figure 4-12.

This result is shown in Figure 4-25, representing the hypothetical mechanisms for this behavior in the cockroach. In order to translate this into specific hypotheses regarding animal function, (the reverse *theorizing* step of Figure 2-2) consider sensors which might contribute to the discrimination of these events. As represented for the stick insect in Figure 4-21, this can be a one-to-very-many mapping; one easy guess for FTi position would be the femoral chordotonal organ (FCO).

5 Functional Cockroach Leg Kinematics

5.1 Introduction

We needed to know how the thorax-coxa joint in the cockroach middle leg worked, so that we could accurately model critical physical aspects of the leg and make relevant hypotheses about its control. The information in the literature is in some cases sufficient to describe leg motion, but the description of the mechanisms for generating this motion was insufficient to build a model with a good structural match. So, we proceeded with experiments to discover the truth. The work in this chapter is an excellent example of useful interactions between engineering and biology.

A functional kinematic model of the major joints of the *Blaberus discoidalis* mesothoracic (middle) leg was developed using observations of the motion of legs in behaving, anesthetized, and dissected specimens. Joint degrees of freedom (DOF) and excursions used by the animal were estimated primarily by observing motion (the function of the joints), rather than from observation of fine anatomical features (their form). These estimated DOF were used to construct a physical kinematic model for verification, and the information was then used to construct a mathematical description of the major joints using the Denavit-Hartenberg formulation of the kinematic equations for a set of linked appendages; see (Asada & Slotine, 1986). This T2 leg kinematic description has 2 DOFs for the thorax-coxa joint and 1 DOF each for the coxa-trochanter and femur-tibia joints. Initial descriptions are also developed for the T1 and T3 ThC joints, and degrees of freedom in the T2 leg which are not yet included in the models.

5.2 Motivation

A diagram of the segments of the cockroach leg is shown in Figure 5-1. The skeletal connetion to the body (thorax) is primarily through soft scleral tissue and several small, complexly shaped cuticular plates. These include two called trochantins, and two more proximal plates called episternums.

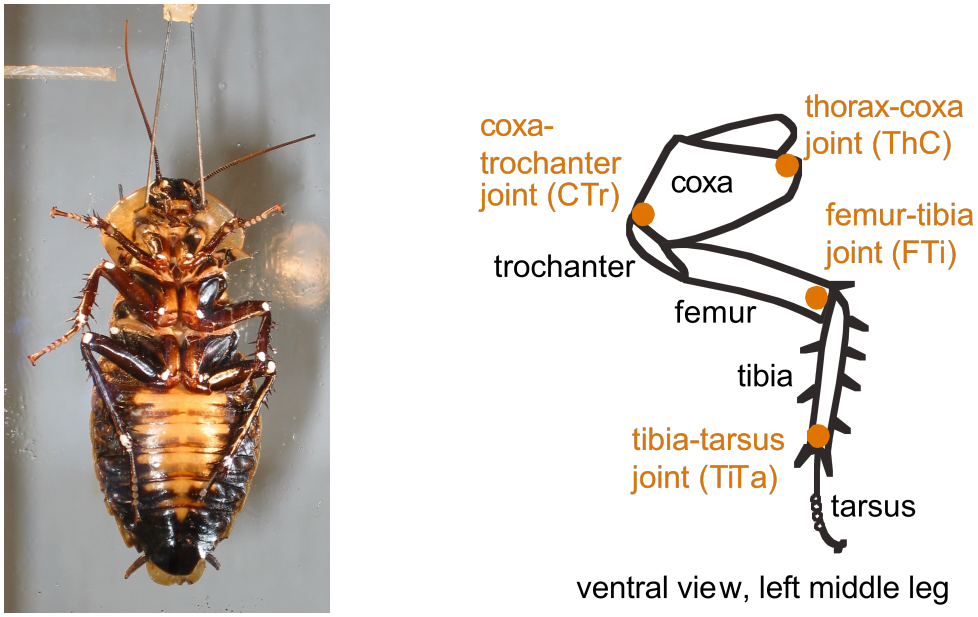


Figure 5-1 Left: ventral view of *Blaberus discoidalis*. Right: Diagram of the segments of the *B. discoidalis* left middle leg, with points used for 3D kinematic digitization. Not labeled in either figure is the Trochanter-Femur (TrF) joint, which is a functioning joint only actuated in one direction (James T. Watson, et al., 2002). Figure from (Bender, Rutter, Simpson, et al., 2008)

A detailed and apparently accurate depiction of the T2 ThC joint structure of *Periplaneta americana* is presented by (Dresden & Nijenhuis, 1953), and describes three degrees of freedom. However a) they do not mention kinematic order, which is a critical descriptor of kinematic function described below, and b) though these might in fact be the DOFs for *P. americana*, only one of the axes they described (P in that paper, ThC1 in this work) appears to correspond to the actual function we observed in *Blaberus discoidalis* middle legs. A detailed description by (Laurent & Richard, 1986) includes the

musculature of the pleurocoxal (thorax-coxa) joint in the prothoracic (front) leg of the cricket *Gryllus bimaculatus*. Though they describe three approximate degrees of freedom and the muscles that actuate them, they refer to the joint as a "ball-and-socket". This description is more apt at the prothoracic leg than at others, but it is still really not true; there are kinematic restrictions introduced by the trochantins and the distinct trochlear notch in the pleural joint.

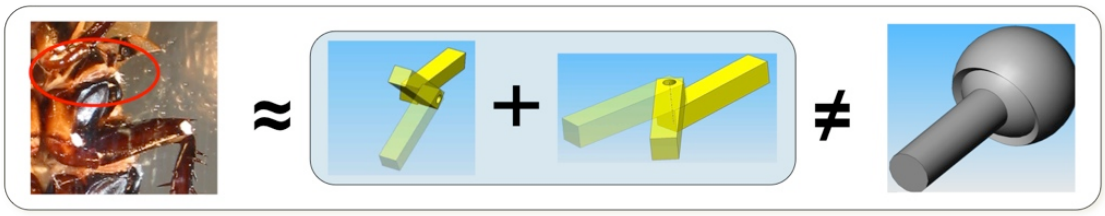


Figure 5-2 In many multi-DOF arthropod joints (on the left: ventral view of cockroach T2 left ThC joint), the cuticular plates and folds form a linkage that can be approximated as a series of hinges. This is not equal in function to a ball-and-socket joint. From (Rutter, Taylor, et al., 2007).

Many multi-DOF joints in arthropods consist of a series of cuticular plates and folds, and their arrangement can be quite complicated. The cockroach T2 ThC joint is a good example of this, as is shown in (Dresden & Nijenhuis, 1953). This is substantially different from the often-used descriptor "ball-and-socket joint", as visually exemplified in Figure 5-2. A true ball-and-socket joint has three independent rotational degrees of freedom. Using these, the rod in Figure 5-2 could rotate up and down, side to side, and twist along its axis – each without regard for the current pose of the other two DOF. In joints or linkages consisting of multiple hinge-like single-DOF elements, the effect of motion in outbound DOF will depend substantially on the position of the preceding DOF. Therefore the order of these DOF, the *kinematic order* of the serial mechanical linkage, is important in describing the function of the system. For a more thorough introduction to these concepts, see (Asada & Slotine, 1986; Greenwood, 1965).

If one only wishes to describe the relative pose of the segments on either side of a joint, it is entirely sufficient to use an arbitrarily chosen set of Euler angles, as was done by (Nelson, et al., 1997). If the order and orientation of these angles do not coincide with the actual physical joint DOFs, however, the behavioral description provided by these angles will be less useful. Motion that involves only one DOF in the animal would in general result in multiple simultaneous changes in these Euler angles. In this case it is harder to interpret the control methods being used in the animal system.

Because of the neuromechanical nature of these locomotor systems, incorrect model mechanics can reduce or destroy the validity of the model as a whole. Both the kinematic order and orientation of the DOFs in a serial mechanical linkage are a critical part of describing system mechanics, so it is reasonable to assume that taking account of these joint kinematic properties is a critical part of physical modeling.

5.3 Methods

The term "functional kinematics" reflects the fact that we are more interested in how the leg *works* than how it is put together. Description of anatomy in these joints can be very complex; going from this description to the information we needed would require an accurate quantitative anatomical model. Constructing such a model is difficult, and might still leave us with inadequate knowledge in the end. What mattered to us at this level of modeling was the behavioral output and basic dynamics of the joint, not the fine details of the mechanism for producing it.

The functional kinematic description here primarily involves only the kinematic function of the skeleton. No attempt was made to describe the muscles or their functional effects. This information would provide a much more complete description of the control

of the ThC joint, particularly since it is likely there are muscles that affect multiple DOF. There is much more work to be done in this regard to gain a complete understanding of functional leg kinematics.

The estimation of joint function done here consists of several steps. First, limbs of anesthetized animals were physically manipulated to estimate skeletal limits of motion, preferred directions of motion in the ThC joint (the joint DOFs), and the kinematic order of these ThC DOFs. This initial DOF estimate was primarily the result of joint manipulation by people with an intuitive understanding of kinematic order. Essentially, this process consists of finding axes about which the coxa prefers to rotate, then determining which other axes' position is important in defining the orientation of each axis. The axes that help define an axis' orientation are more proximal (nearer the body) than the current axis in the kinematic order; those that do not are more distal.

The behaving animal was then observed to verify that its self-induced motion was within the ranges and constraints estimated above. Visual observation of walking and searching behaviors were used in this step, with the naked eye, under a dissecting scope, and with the assistance of high-speed video. This was a visual estimation of motion, rather than a precise analysis using video digitization. The data presented below are the result of myself, Brian Taylor and Richard Bachmann all going through this process separately (single-blind), and coming to a consensus. The final step of the process was the use of 3D imaging of intact awake animals to better estimate the orientation of the coxa and its degrees of freedom in standing animals.

5.4 Results

The kinematic data presented here were obtained with the assistance of Brian Taylor, Richard Bachmann and Michael Cohen.

5.4.1 Apparent Degrees of Freedom of the Mesothoracic Leg

The apparent degrees of freedom in this leg were as follows; organized by joint from proximal to distal. Note that the names of the actions of the joints do not necessarily represent what they actually do in the cockroach; these names adhere to standard insect anatomical descriptions that address a more stick-insect-like leg anatomy.

I.Thorax – Coxa (ThC)

1.Promotion – Remotion

Shown in the top of Figure 5-4; promotion depresses the entire leg about axis AB of Figure 5-5 and retraction levates it. The range of motion of this DOF is approximately 80°.

2.Adduction – Abduction

Shown in the bottom of Figure 5-4; adduction brings the leg closer to the body centerline about point A of Figure 5-5 and abduction moves it away. The maximum range of motion of this DOF is approximately 30°, and becomes smaller as coxa protraction increases.

The orientation of the ThC1 axis used in the robotic cockroach model, based on 3D image analysis of several animals, is given by two coordinate rotations from a body reference frame where positive x is forward with y pointing to the left. The first rotation is about body X: Thetax = 130 degrees The second rotation is about the new Y: Thetay = -20 degrees. The resulting coordinate system points Z along ThC1 into the body, as shown in Figure 5-3. The complete Denavit-Hartenberg

description of each model leg is given in the associated joints.xml configuration file; provided in the SCCILS Appendix 10.2.

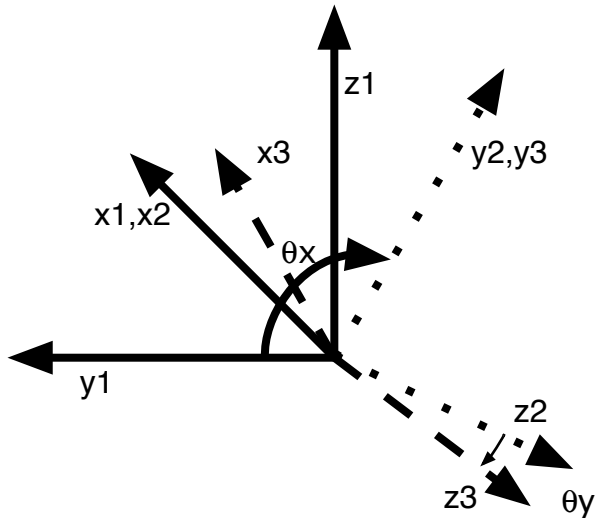


Figure 5-3 The two rotations from the body coordinate system (1) to the ThC1 frame (3) used in the cockroach robot. There is first a rotation of Thetax = 130° about x1, then a rotation Thetay = -20° about y2, resulting in the coordinate system (x3, y3, *z*3). The ThC1 DOF rotates about *z*3, which points medially and slightly ventrally and caudally.

II.Coxa – Trochanter (CTr)

1.Levation – Depression

Levation rotates the trochanter and femur rostrally in the standard coxal pose shown in Figure 5-5, resulting in leg protraction. The limit of levation results in an overlap of coxa and femur, with the distal end of the femur nearly reaching point A of Figure 5-5. Depression rotates the trochanter and femur caudally in the standard coxal pose, resulting in leg retraction. The range of motion of this DOF is approximately 100°.

III.Trochanter – Femur (TrF)

1.Reduction – Passive Return

Reduction rotates the femur about the TrF attachment line shown in Figure 5-1, levating it away from the ground (into the page) in standard coxal pose, and re-orienting the plane defined by the femur and tibia. In most leg poses encountered during stepping behavior, this results in a slight depression of the foot. Passive return brings the femur back towards the ground in standard pose, bringing the femur-tibia plane coincident with the coxal plane. The mechanism of this motion is described in detail in (James T. Watson, et al., 2002). Both the angle of TrF orientation with respect to the tibia and the range of motion of this joint were observed to be somewhat less than the approximate 45° reported there; perhaps as little as 30°. This estimate is still approximate; see section 9.2.2.12 in the future work.

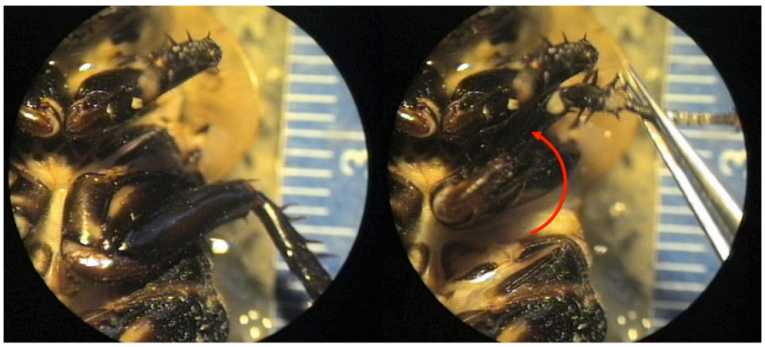
IV.Femur – Tibia (FTi)

1.Flexion – Extension

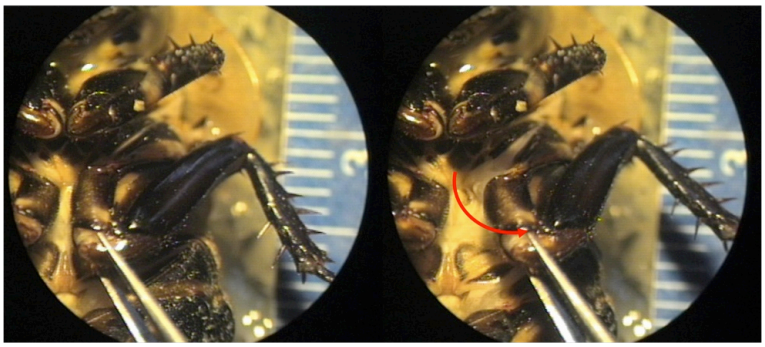
Flexion and Extension are both straightforward descriptors of the motion of this joint with respect to the femur. The range of motion here is approximately 150°, more or less centered around a pose of 90° from the line of the femur.

V.Tibia – Tarsus (TiTa)

The motion of the tarsus is complicated. There are at least two actuated degrees of freedom at the tibia-pretarsus joint in agreement with (Alsop, 1978), as well as active tarsal curl and the passive compliance of tarsal segments.



ThC-1 Ventral View, Left Leg



ThC-2

Figure 5-4 Demonstration of T2 ThC degrees of freedom. Top: ThC1 remoted (left) and promoted (right). Bottom: ThC2 adducted (left) and abducted (right). These manipulations are similar to some of those used in the first step of estimating joint DOFs. From (Rutter, Taylor, et al., 2007).

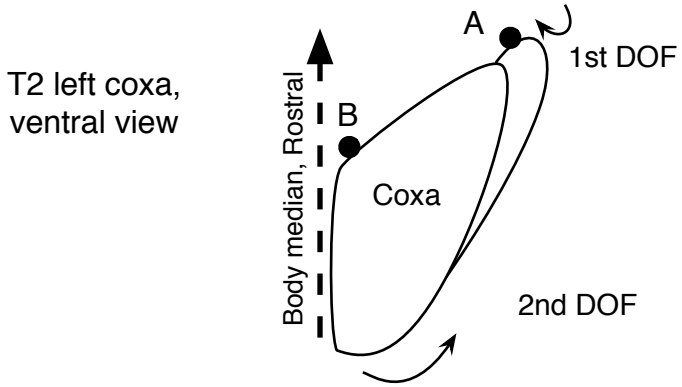


Figure 5-5 T2 *B. Discoidalis* left coxa, ventral view. The body midline is vertical just left of B, and rostral is up. The plane of the leg and coxa are coincident with the page; this is the "standard coxal pose" also used in Figure 5-6. The first DOF, ThC1, is along line AB. The second DOF, ThC2, is about an axis straight out of page through point A. From sketch on p 79 of Rutter biorobotics lab notebook #1.

5.4.2 Apparent Degrees of Freedom in the T1 and T3 ThC Joints

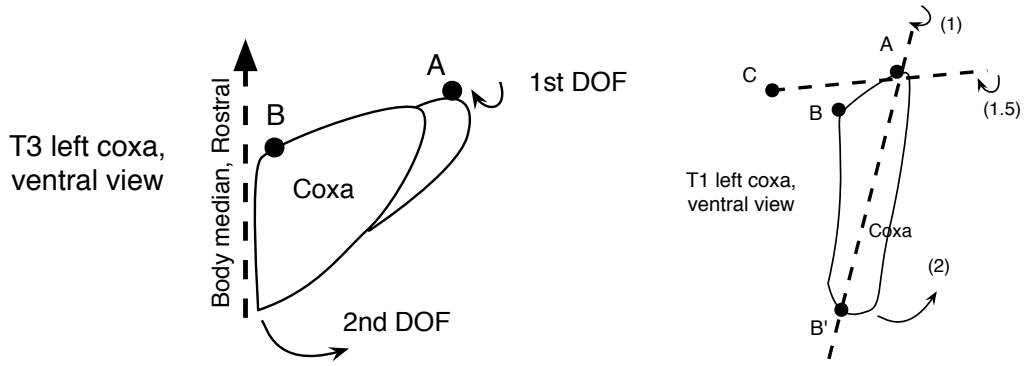


Figure 5-6 The apparent functional degrees of freedom at the T1 and T3 ThC joints, in the standard coxal pose described in Figure 5-5. Left: T3; similar to T2 but with the first AB DOF (ThC1) pointed more away from the body center line. Right: T1, which differs significantly from T2 and T3. The AB' line defines ThC1, which in this case has moved away from the anatomical thorax attachment line AB. ThC2 is still about a line through A perpendicular to the plane of the leg, A degree of freedom has apparently been introduced between these two, very approximately along AC.

The degrees of freedom and order of the T3 ThC joint are similar to those in T2. Estimated ranges of motion are 70° for ThC1 and 30° for ThC2. As in T2, minimal rotation of the coxa in the plane of the thorax is possible, and not dependent on pose of the other DOF. This does not appear to be associated with obvious actuation or behavior, however.

The kinematics of the T1 ThC joint are substantially different from the T2 and T3 joints. The anatomy is much less restrictive; there are more soft tissues and less restrictions arising from cuticular plates such as trochantins. ThC1 and ThC2 still exist, but ThC1 is now more accurately described as a rotation about the axis of the elongated coxa, and has a range of motion of approximately 90°. An intervening DOF has been introduced which is approximately about line AC in Figure 5-6. It is suggested that this is referred to as ThC1.5, although renaming the ThC2 DOF might also be sensible in this case. The range of motion of ThC2 in this leg still varies with ThC1 pose but has a much

greater maximum excursion (about 90°) than in the other legs, and is used extensively in walking and more than ThC1 in searching.

5.5 Applications and Conclusions

The functional degrees of freedom and their kinematic order in the *B. discoidalis* mesothoracic leg were estimated. These approximate kinematics were used to construct a physical kinematic model for evaluation and instruction, shown in Figure 5-7. Additionally, a subset of these DOFs were both used in the construction of the cockroach robotic model shown in section 6.4.2, and have been used to guide the further behavioral and kinematic analysis (Mu, 2007), (Bender, et al., 2009 (In Prep)).

The following simplifications were made in both the kinematic mathematical model in (Mu, Taylor, Rutter, et al., (in preparation)) and the robotic model:

1. The ThC1 DOF is in a constant, canonical orientation for all individuals.
2. Axes of the Thoraco-Coxa DOFs intersect (at the point A in Figure 5-5)
3. The axes for the Coxa-Trochanter, Trochanter-Femur, and Femur-Tibia DOF are all parallel, and the coxa, femur and tibia are co-planar.
4. The Trochanter-Femur and Tibia-Tarsal DOF are excluded due to measurement difficulties and apparent low behavioral relevance.

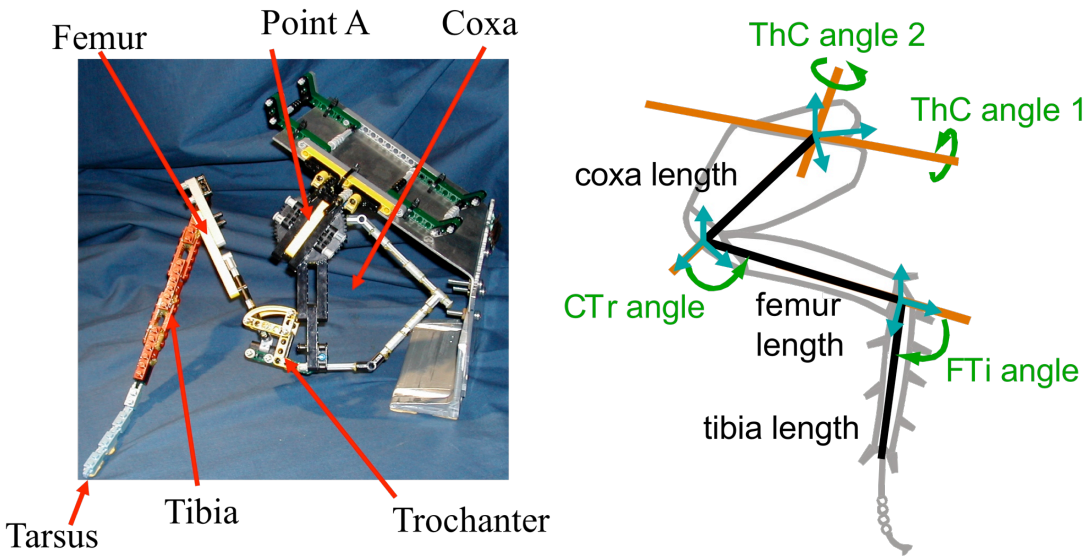


Figure 5-7 Left: physical model used in demonstration of the degrees of freedom in the mesothoracic leg. (built by Brian Taylor) Right: diagram of the DOFs used in the robotic model, and kinematic model of (Mu, 2007; Rutter, Taylor, et al., 2007), from (Bender, Rutter, et al., 2008).

A kinematic modeling system using these kinematic data was constructed (Mu, 2007; Mu, et al., (in preparation)), and used to investigate the roles of each joint in generating foot motion during walking and turning behaviors. The importance of ThC pose and the coordination between CTr and FTi in changing from walking to turning behaviors are demonstrated using this kinematic model.

In the modeling work here, as in nearly all previous work, the action of the TrF joint was left out. It was demonstrated in (James T. Watson, et al., 2002) that this joint is important in climbing, but it was supposed that it might be ignored for forward walking at least. Recent more thorough measurement by (Bender, et al., 2009 (In Prep)) suggests that this is not the case, and even that the TrF joint might be the element in the animal which primarily controls foot levation and depression, rather than the ThC1 DOF used for this purpose in the robotic cockroach model.

The model builder should note that the ranges of motion given here are related to skeletal limits rather than behavioral observations. Models of biomechanical systems should generally be built with these less restrictive physical limitations, unless there are clear and well-supported reasons for doing otherwise. The walking animal is dealing with this physical system; not one with a more limited range which can produce walking. Building a model based on behavioral observations substantially limits the models' usefulness to those behaviors. If artificial muscles such as braided pneumatic actuators with a force-length characteristic are used, one must additionally keep in mind that most animals are capable of exerting substantial force even at the skeletal limits of their joints. Artificial muscles and joint geometry should be chosen, then, where the range of muscle contraction during joint motion is substantially less than the maximum that the muscles can produce. This is unless, of course, biological data to the contrary is available.

6 Neuromechanical Robotic Models

6.1 Introduction

This chapter describes the SCASM-controlled robotic leg models as implemented in the NeuRoMod control program suite (operating manual in Appendix: 10.4), and the physical stick insect and robot models. Both the control program and the hardware used to run it are described. Both the hardware and software in these models have gone through a prototyping process in which the implementation is begun as simply as possible and improved as needed to adequately meet the requirements for reverse-path hypothesis generation mentioned in section 2.2. The design requirements for the robotic models included "portable" and "electrical actuation"; potential problems with poor dynamic match were seen for the electrical actuation requirement, but these have been dealt with sufficiently for this initial use. The control software was designed with a significant amount of modularity and configurability. The modularity has been necessary for straightforward additions when new functionality has been necessary, and the configurability has allowed the same control software to deal with different and evolving robotic models.

6.2 The NeuRoMod Engine Control Architecture

This section describes the *engine* portion of the NeuRoMod control suite, in the SCCILS sense of "engine" described in section 4.3.1. The local control system is designed to allow modular addition of muscle models, dynamic muscle activation levels, and the continuous modification of activations and state transition conditions based on higher-level control inputs. This local control might be thought of as occupying the mid-

level sequencing and low-level reactive portions of a three-layer architecture like those described in 3.6.1.

This software runs on Pentium III machines using RT-Linux (Yodaiken & Barabanov, 1997), a real-time operating system which is described in more detail in Appendix 10.4. The control code was implemented as a number of concurrently running real-time threads contained within two real-time kernel modules. This allows for greater control modularity than single-process designs, and is a better structural match to the organization of the associated computational processes in the animal. The conceptual location of these threads in a three-layer architectural description is shown in Figure 6-1. The same threads are shown in the context of a neuromechanical joint in Figure 6-2.

Documentation for the programs and custom function libraries referred to here is automatically generated using Doxygen (van Heesch, 2009, www.doxygen.org) and is available in the code repository. Including all 10k+ lines of C code in the appendices would be unreasonable, but a summary index of the documentation is included in Appendix 10.1.

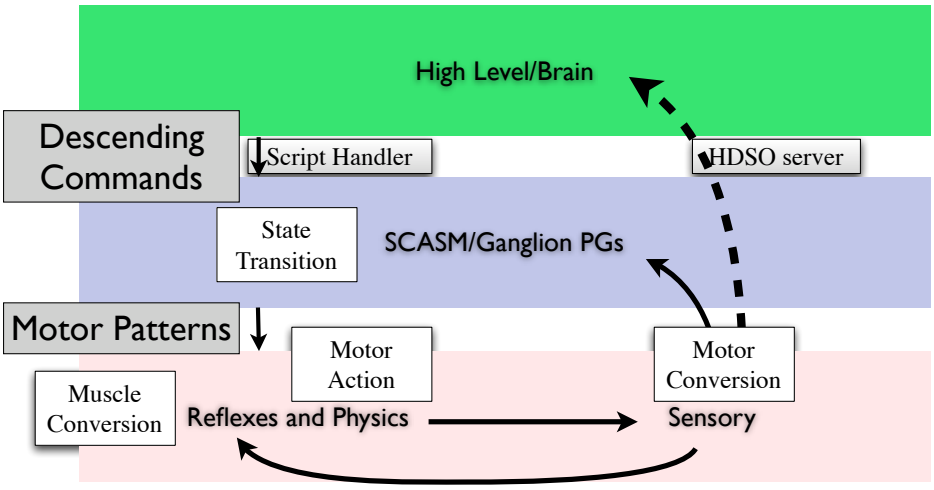


Figure 6-1 Location of real-time threads on the three-layer architecture backdrop. High-level control influences SCASM via descending commands; SCASM specifies motor patterns for lowest-level control. All arrows represent the flow of information between system elements. White rectangles are threads in the engine module; the state transition thread initiates state transitions and sets muscle activations and any low-level reflex rules, making it the thread that actually enacts SCASM. The light grey boxes are threads in the script module and handle SCCILS HDSO I/O: the script handler takes parameter modifications from the high-level (interface) and enacts them in lower levels. The HDSO server provides low-bandwidth access to most of the sensory data *and* parameter values in the lower levels of control.

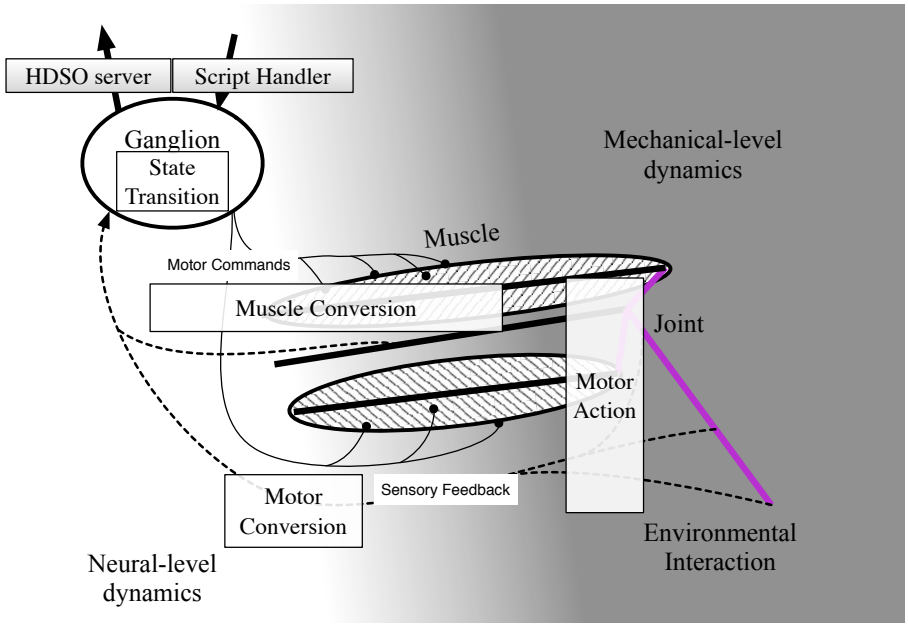


Figure 6-2 Approximate location of real-time threads in the context of the neuromechanical joint system. The script handler and HDSO server threads process higher-level interaction. The state transition thread implements SCASM, modeling the joint pattern generators. The muscle conversion thread models muscle and joint dynamics. The motor action thread handles low-level physics modeling and physical I/O. The "motor conversion" thread implements low-level sensory filtering and processing.

6.2.1 Enginemod: The Control Real-Time Module

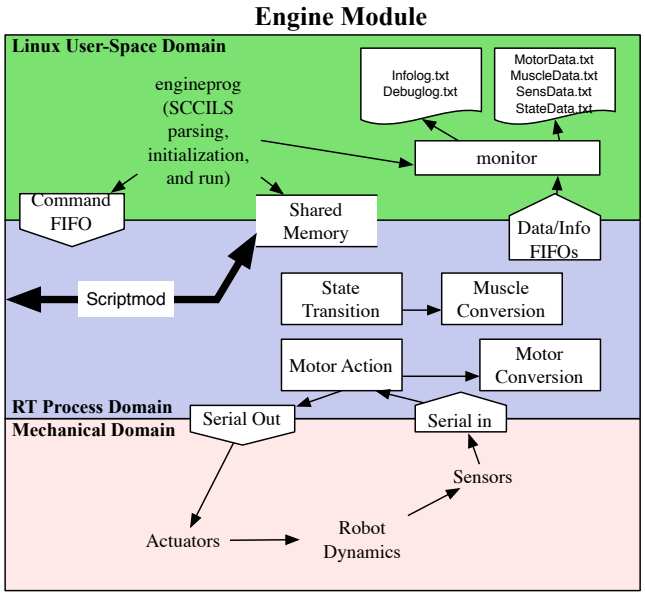


Figure 6-3 Enginemod thread event relationships. State Transition and Motor Action threads run at their own periodic timebases, and kick the conversion threads when necessary. All four threads access shared memory and produce output that goes through the data/info FIFOs. Many shared memory variables may be considered the "output" of one thread and the "input" for another. Scriptmod interacts with the controller through its connection to this shared memory.

Enginemod runs four concurrent threads, but control takes place primarily in the Motor Action and State Transition threads. The Motor Action Thread handles the feed-forward force control and runs up to a maximum rate given by computation and serial communication overhead, currently 197 Hz. The second SCASM control thread runs at a lower priority and handles the state transition and activation calculations. The update rates of these threads are both configurable; currently recommended settings of 100 Hz and 40 Hz respectively. All control and sensory data are stored in shared memory accessible both from real-time and user space, allowing online modification of the control. Each thread logs data every execution cycle via FIFO to a high-priority user space program that writes all data, including sensor readings, states and muscle activations, to files.

6.2.1.1 The Motor Action Thread

The motor control thread handles low-level hardware I/O, and the feed-forward torque control calculations. The system is fast enough that angle feedback can be used in conjunction with the servomotors' proportional control to implement a crude, but stable, feed-forward torque control. For a given desired torque output, the servo is commanded to go to some delta of position from the current position; the proportional control used by the servo then generates a torque proportional to this delta. This system generates smooth compliant motion from an electrical motor system with relatively little computational overhead.

This is implemented as follows: in the servomotor, the current/torque command sent to the motor coils is at least primarily a proportional control, calculated as in Equations 6-1, where $\theta_{command}$ is the desired angular position and $\theta_{measured}$ is that reported by the servo's sensing device. In servo delta control, the command angle is calculated at a higher level and at a high rate by using Equation 6-2. In this case θ_{err} , and thereby the motor current/torque, can be controlled as shown in Equations 6-3.

$$current\_command = P * \theta_{err}$$
$$\theta_{err} = \theta_{command} - \theta_{measured}$$ **Equations 6-1**

for servo delta control, set

$$\theta_{command} = \theta_{measured} + \theta_{servo}$$ **Equation 6-2**

then

$$\theta_{err} = \theta_{measured} + \Delta_{servo} - \theta_{measured} = \Delta_{servo},$$
$$current\_command = P * \Delta_{servo}$$ **Equations 6-3**

6.2.1.2 Motor conversion thread: Sensory Processing

The motor conversion thread actually just does low-level sensory processing and filtering on the raw A/D sensory data that the Motor Action Thread records in shared memory. Angle data are used both in raw 8-bit form and after having a calibration applied, but no sensor filtering is done. The current data are filtered using the following IIR filter (after initialization).

$$\text{filt\_current} = 0.05*\text{current} + 0.95 * \text{prev\_filt\_current} \qquad \textbf{Equation 6-4}$$

6.2.1.3 State transition control thread: SCASM

The state transition thread sets the state at each joint by checking the sensory thresholds described in the Boolean state transition rules. In so doing, this is the thread that implements the action switching of SCASM in these programs. It also calculates and sets the activations for the muscles at each joint as specified by the current action. Though most configurations currently use constant muscle activations, they are dynamically calculated and can use any available sensor and any of the generic functions available in the *utility.c* library.

To simulate sensory-motor control loop delays, the state transition thread is run at less than the maximum possible. A period of 25 milliseconds is used in the cockroach robot, corresponding to a hypothesized sensory loop delay of about 7.9 ms in the animal. As discussed in section 9.2.4, this method has drawbacks because it also affects the fineness of sensory event detection, and the speed (and therefore fidelity) at which muscle dynamics are calculated. Motion was not obviously changed by adding the 25 ms delay here, however.

6.2.1.4 Muscle conversion thread: Muscle modeling

The muscle conversion task handles muscle model calculations, and sets the Δ_{servo} values used by the Motor Action thread. Here, the term "muscle model" refers to a crude mathematical model of a biological muscle, where the muscle is represented as a contractile element whose force at a particular level of activation is scaled by its current length and velocity. The length of the element, in turn, is determined by modeling the joint as a constant-radius pulley; an inaccurate but computationally simple joint model. This model of muscle activation is simpler than those intended to accurately model muscle dynamics (e.g. that of (Hill, 1970)), which can include a more complex arrangement of passive nonlinear stiffness and damping elements.

The servo delta at each joint is computed using the following relationship:

$$\Delta_{servo} = \sum_{muscles} [r \cdot f]$$
$$f = PS + activation \cdot FL \cdot FV \cdot f_{max}$$

Equation 6-5

where r is the radius of the pulley associated with each muscle, f is the current force of that muscle, PS, FL and FV are the values of the muscle's functions for passive stiffness, force-length activation scaling, and force-velocity activation scaling respectively, and f_{max} is a parameter intended to represent the maximum force which can be exerted by each particular muscle. PS, FL and FV can be calculated using constant, linear, parabolic or hyperbolic relationships, as implemented in *generic_function* in *utility.c*. As an example, the linear force-length characteristic FL was a function of nondimensionalized muscle length l':

$$FL = a + b \cdot l'$$
$$l' = \frac{(\alpha - \theta) r + l_0}{l_0} \qquad \textbf{Equation 6-6}$$

with a and b constant parameters, θ the joint angle, l_0 the "resting" length of the muscle and α the joint angle at which the muscle reaches this length.

Though it is not modeled explicitly and has not been quantified, there is also a force-velocity dependence inherent in the force control method used in the Motor Action thread. As a joint moves more quickly in the direction it is being driven, the average distance between the commanded position and the joint's actual position will become less, reducing the torque applied to the joint. The inverse is also true; decreasing joint velocity increases force. In the limit of the static case, θ_{err} is always equal to Δ_{servo} and will therefore actually be exerting the associated torque. At high speeds the low-level θ_{err} may never equal Δ_{servo}, since the low-level $\theta_{measured}$ will have moved closer to the calculated $\theta_{command}$ by the time the Δ_{servo} command reaches the servo. Additionally, there is some speed-output torque relationship for any physically existing motor and transmission, which also reduces maximum torque output at higher speeds.

6.2.2 Scriptmod: The SCCILS Interaction Real-Time Module

The low-level hooks for online SCCILS configuration changes are in the HDSO data store in the RT-Linux shared memory, accessed by enginemod. Scriptmod provides timed read and write access to these data through its two threads: the script handler and the HDSO data server.

The script handler thread receives HDSOmod data change commands (already translated from XML to the native engine representation) and implements them as closely

as possible to the times specified. Once an HDSOmod has been applied, the script handler puts that HDSOmond on the Scriptlog FIFOs, marked with the time of actual implementation.

The HDSO server thread receives HDSOspec specifiers for requested HDSO data. It then reads the associated data from the shared memory HDSO store and writes them to the HDSOlog FIFOs, marked with the current time.

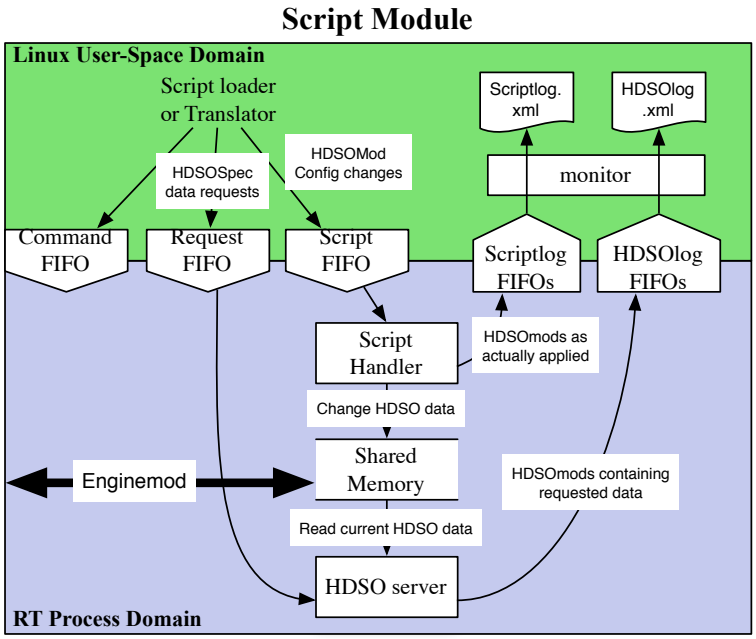


Figure 6-4 Scriptmod data flow. The script handler takes HDSOmod config changes via the Script FIFO, implements them at the desired time, and records actual implementation via the Scriptlog FIFOs (to Scriptlog.xml). The HDSO data server takes requests for HDSO data via the request FIFO, then looks up the specified data and writes it to the HDSOlog FIFOs in HDSOmod form. The "Shared Memory" in this figure is the shared memory HDSO store, a subset of the "Shared memory" block in the enginemod diagrams; thus the "connection" with enginemod.

6.2.3 User-space programs

Although the actual control computations take place in the real-time modules described above, there are a number of things that happen in programs in the "Linux User-Space Domain" elements of Figure 6-3 and Figure 6-4. This reflects the fact that user

interaction, initial system configuration, and file parsing and I/O are better handled in the normal Linux programming environment.

Parsing and configuration are accomplished primarily by the *engineprog* command-line program. It is used to load and unload the RT-Linux modules, and makes heavy use of functions in the *userio.c* library. The *monitor* program logs data from RT data FIFOs to disk, and is run as a very high-priority user-space task. The *datalogging* command-line program is used to load and unload the daemon-like *monitor*, and it does some parsing of raw FIFO data written to disk by *monitor*.

The Python SCCILS translator and the command-line enginecli program are higher-level programs for running the engine interactively or executing experimental scripts. Each of these makes use of *engineprog* and *monitor/datalogging* for the services they provide. The Python translator, of course, interacts with the graphical user interface via SCCILS, and via that SCCILS communication is capable of interacting with higher-level control of any nature.

6.3 Control hardware

6.3.1 Computers

Control is implemented on computers running RT-Linux (Yodaiken & Barabanov, 1997) that also record all state and sensory data for analysis and allow on-line modification of all SCCILS-aware system parameters. The computers used for the work presented here are 400 & 500 MHz Pentium-III machines. All input/output for these prototypes is via RS232 serial communications with the AIMotor actuator/sensor packages.

6.3.2 AIMotors

AI-series servomotors from Mega Robotics (Megarobotics Co., Ltd. Seoul, Korea) were used in both the stick insect and cockroach model legs. Both the AI-701 and higher-torque AI-1001 motors were used. Communications with the motors was handled with the *aimotor.c* minimal driver library in the NeuRoMod software. The RS-232 serial line from the computer connects to a combination level converter and power board; all motors then get power and TTL-level RS-232 serial communications through a four-wire bus. These servos are controlled via an RS-232 serial data line, and provide 8-bit angle and load feedback to the host controller over the same line. Angle feedback is used in conjunction with the servomotors' proportional control to implement a crude feed-forward torque control described in section 6.2.1.1 above. Motor current feedback is available at a resolution of ~18.4 mA per A/D tick, and position command and feedback are both at a resolution of 0.654 degrees per tick.

6.4 Physical Models

6.4.1 Stick insect

The stick insect model, shown in Figure 6-5, is a 14.3:1 scale 3-DOF model of the right middle leg of the stick insect *Carausius morosus* constructed primarily by Brian Taylor. The segment lengths are coxa = 22.6mm, femur = 192.8mm, and tibia = 189.0mm, giving coxa-relative lengths of 1, 8.58 and 8.41, and a total leg length of 404.4mm. These ratios are within the range reported in (Cruse & Bartling, 1995). The orientation of the ThC DOF uses the values from (Cruse, 1976); first a rotation of -85° about the body Z axis, then a rotation of 40° about the resulting Y axis (see Figure 5-3 for the corresponding rotations in the cockroach). Though the animal has more than one DOF at the thoraco-

coxal joint, only the primary protraction/retraction DOF was used, as in the model of (Ekeberg, et al., 2004). A smooth, low-friction foot, constructed using a half of a table-tennis ball, simulates the frictionless surface used by Ekeberg *et al.* in their modeling when used on a hard, smooth surface. Note that this is the extent of our contact modeling, and was not an explicit design issue. The joints are actuated using AI-series servo motors described in section 6.3.2. The ThC and FTi joints use model AI-701 while the CTr joint uses the higher-torque AI-1001. The femur and tibia segments of this prototype were constructed from 9.5 mm diameter plastic coat-hanger shaft, and these are connected to the motors using adapters made from Delrin®plastic which interface with the standard slide-in connectors of the AI-series servomotors. The motor driving the ThC joint is connected to a body link, which is attached to the base on two vertical sliding rails in order to adjust the modeled body height.

In this model, the current load on the CTr motor is used as a measure of leg load, instead of directly measuring ground contact. This load sensor is located similarly to some of the load sensing organs in the animal which serve the same purpose (Ekeberg, et al., 2004) (Hofmann & Bässler, 1982).

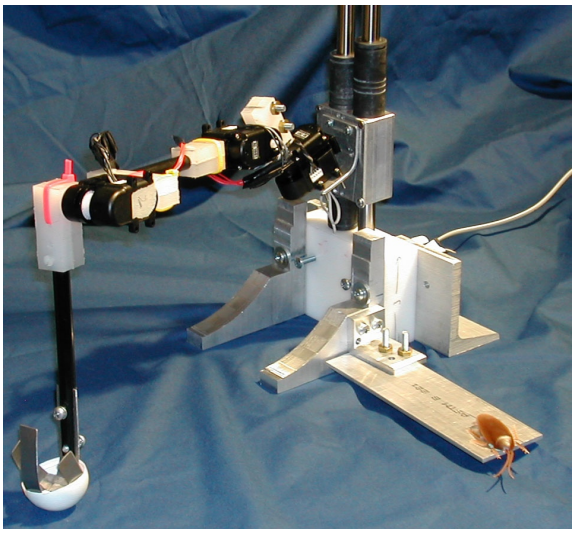


Figure 6-5 Robotic scale model of the stick insect right middle leg. This view is from the front; the bar extending to the bottom right indicates the orientation of the thorax in the horizontal plane.

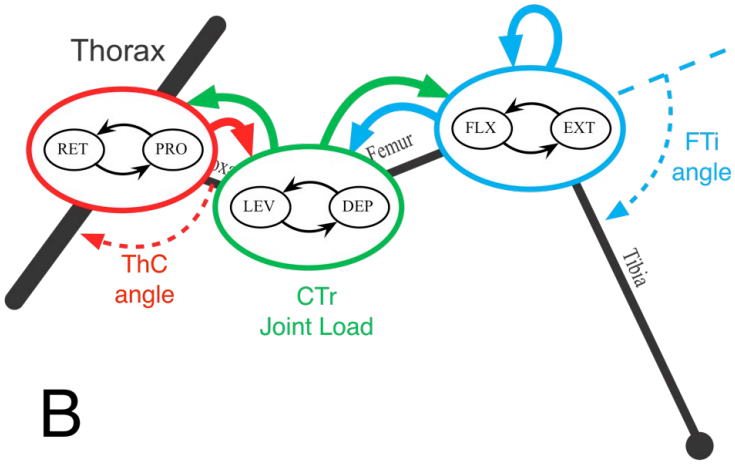


Figure 6-6 Sensory flow diagram valid for all stick insect modeling done in this work. Signs & strengths of connections can change, but this figure represents which signals can have any effect on each action switching module.

6.4.2 Cockroach

The cockroach model, shown in Figure 6-7, is a 10.1:1 scale 4-DOF model of the left middle leg of the cockroach *Blaberus discoidalis* constructed primarily by Tyson Papay and Devon Parker, based on original designs by Brian Taylor. The segment lengths are coxa = 87.2mm, femur = 93.2mm, and tibia = 74.7mm, giving coxa-relative lengths of 1,

1.069 and 0.857. The orientation of the ThC1 DOF is described in Figure 5-3; the rest of the leg is articulated within one plane. The coxa, femur and tibia segments are solid machined aluminum, while the connection between the motors for ThC1 and ThC2 are a plastic-aluminum sandwich beam to increase stiffness. Similarly to the stick insect model, a rounded plastic foot simulates the low-friction oiled-plate environment often used in animal experimentation (Gruhn, Hoffmann, Dübbert, et al., 2006; Tryba & Ritzmann, 2000). There is an AI-Motor at each joint; the ThC1 motor is model 1001, the others are model 701.

Although there is a motor for the ThC2 DOF, this is left in position control mode at a fixed angle for all of the modeling work done so far.

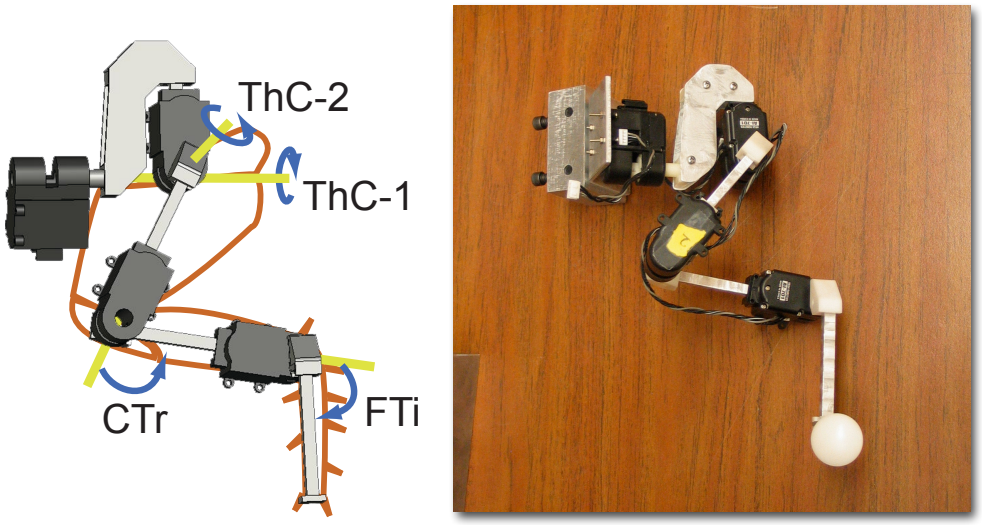


Figure 6-7 Cockroach left middle leg model, ventral view.

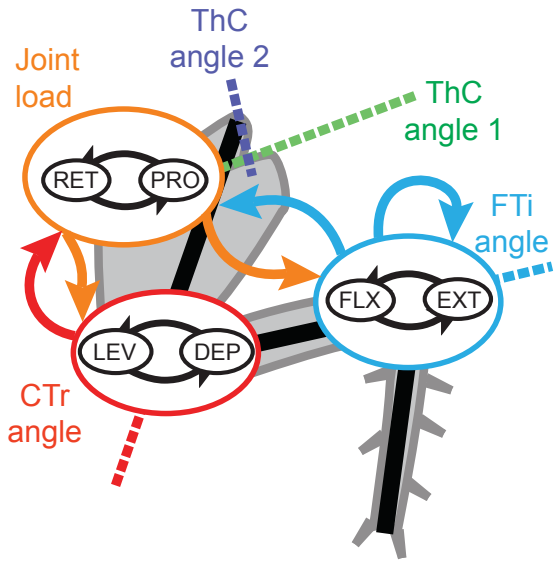


Figure 6-8 Sensory flow diagram for all cockroach modeling done in this work.

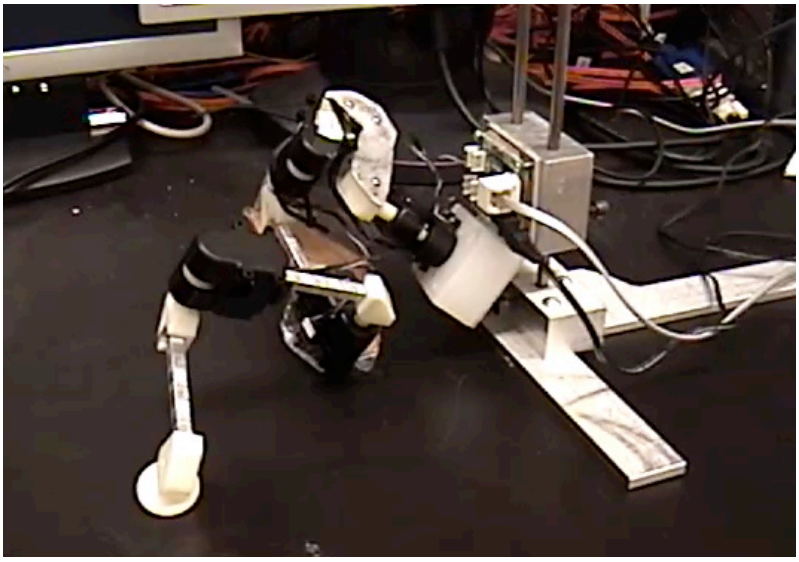


Figure 6-9 Cockroach leg on mount and ready for an experimental run (image taken from experimental video data. The base element pointing down and to the right is parallel to the simulated body center line, and pointing caudally.

7 Experiments

This chapter presents experiments that have been performed as examples of the kinds of questions that can be addressed using these neuromechanically based robotic models. Appropriate experimental practices are introduced, and methods of displaying and interpreting the resulting data are also shown. The first section covers basic description and testing of model functionality. The second presents a preliminary analysis of the effects muscle models can have on overall leg behavior in the stick insect model. The next two sections present a series of experiments that begin to address the question: what is necessary to modify behavior of the system? First, a brief example of a method to change stepping speed is presented. Then transitions between walking and turning behaviors in the cockroach model are addressed.

7.1 Initial Testing and Functionality

Some initial demonstration of functionality is shown here for three of the systems described in Chapter 6; stick insect simulation (Ekeberg, et al., 2004), stick insect robot, and the BILL-ANT two-leg test platform. Many of these results were originally presented together in (Rutter, Lewinger, Taylor, et al., 2006), (Lewinger, et al., 2006). The inclusion of these systems together is useful in demonstrating the use of SCASM for control and modeling in systems that are similar but not identical. To the engineer of legged systems, this may be interesting in the demonstration of breadth of applications. A biological modeler could find this interesting both in a comparison of the capabilities of the various systems (which might be evident in an examination of Figure 7-4), and in the examination of which control parameters vary the most or least widely between systems (see, e.g., Table 7-1).

All three leg systems generated stable forward stepping behavior when appropriately tuned through the interactive selection of thresholds for switching events and muscle activation or action dynamics parameters. Restricted stepping, in which the ThC joint is fixed, was also used in development and some experiments.

7.1.1 Adaptability demonstrated in restricted stepping experiments

We first tested the operation of the stick insect robot leg in the restricted single leg preparation, which corresponds to a reduced biological preparation in which all legs but one are removed and the ThC joint of that leg is immobilized (Bässler, et al., 1991; Fischer, et al., 2001). This preparation allows a stepping movement sometimes called "sideways walking". During sideways stepping tests, the body link was adjusted such that the ThC joint was about 16 cm above the ground, corresponding to an insect-scale height of about 11 mm.

One of the most difficult aspects of stepping control in these models is the swing-stance transition. Initial tests for the stick insect model were done with a null muscle model and the leg would sometimes stop movement instead of completing this transition. It was believed that a muscle model with a simple force-length characteristic would help address this problem, which is described in section 7.2. As an interim measure during initial testing and before a muscle model was implemented, a third state was added to the FTi joint in which the extensor muscle activation is reduced after extension has progressed past a certain point. This made the swing-stance transition much more reliable when the appropriate muscle activations and sensory thresholds were properly tuned, as shown in Figure 7-1.

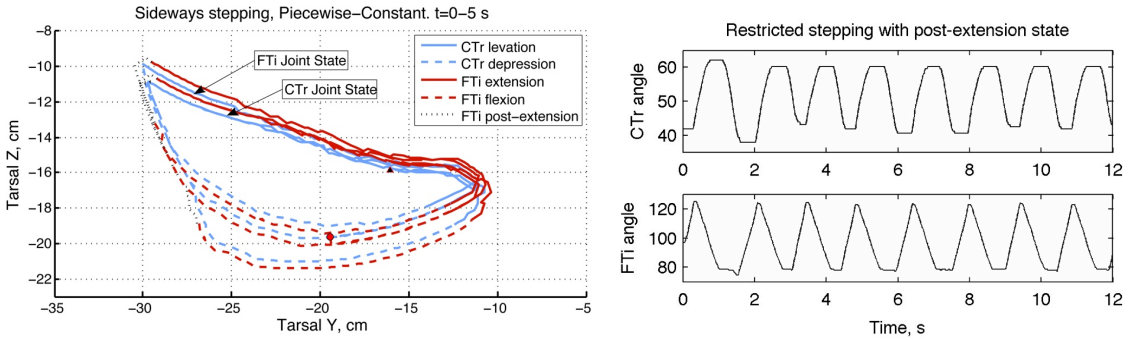


Figure 7-1. Path-state plot and joint trajectories of restricted stepping in the stick insect robot model, with the third FTi "Post Extension" state. In this example the system quickly stabilizes. The path-state plot is a plot showing simultaneous foot path and joint states, and is used in several of the following figures. Each line represents the state of one joint, styled as shown in the legend. The foot path calculated from joint angle data is represented by the line which starts at the circle and terminates at the triangle; in this case that is the line representing CTr joint state. In this and in Figure 7-8, negative Y is away from the body since the model is a right leg (see Figure 4-1). The origin is at the ThC joint, and this is a projection of the already nearly planar restricted stepping motion is therefore onto a transverse plane.

The adaptability of a SCASM system and its dependence on the environment and sensory feedback are readily demonstrated using this preparation. As the height of the body link is increased from approximately 16 cm to 26 cm, the leg continues side stepping in contact with the ground, without modification of control parameters. The control system was not obviously vulnerable to the initial state of the joint control modules, and behaved well over a range of initial kinematic poses, including at least those contained in whichever behavior was currently configured. If the structure of the leg is disconnected, leaving control connections in place, the motions of the joints do not exhibit the same patterns and may stop. Re-connecting the leg structure causes the system to resume stepping motion. An experiment demonstrating this, in which the physical structure of the leg was temporarily disconnected just below the CTr joint, is shown in Figure 7-3. Disconnection and reconnection were done by hand, and each took about two seconds. In some cases of this experiment (not shown), the system is trapped in one of the plateau regions and motion stops until the leg is externally disturbed.

It is apparently necessary for me to point out that although there are some similarities, this is **not** equivalent to a deafferentation experiment, or indeed any experiment where neural function is disturbed. Even if the efferent connections to the levators and depressors of the trochanter (near the leg disconnection point) were temporarily disconnected, it would still not be an equivalent experiment. **Physical** force information would still travel up the leg to the trochanteral campaniform sensilla, and the forces applied by the body and ThC joint would still influence the motion of the rest of the leg. These *physical* carriers of information are disrupted in this mechanical dissociation experiment. This is shown graphically in Figure 7-2. Once the structure of the robot is

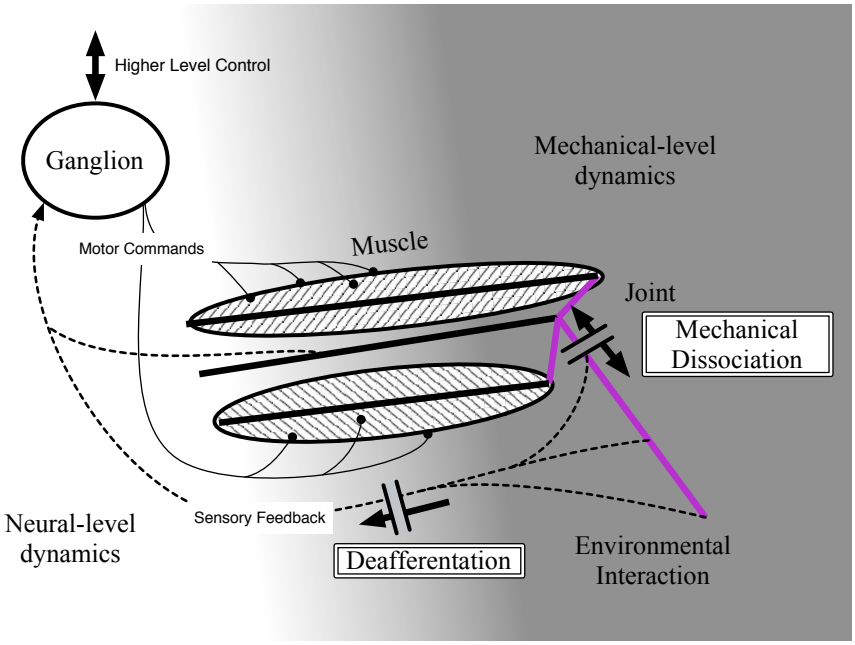


Figure 7-2 Differences between a deafferentation and a mechanical dissociation, shown in the context of the neuromechanical joint. The shown deafferentation stops the flow of sensory information from receptors in that section of the leg to the local ganglion. Mechanical dissociation at the point shown stops the flow of mechanical information, in the form of force, in both directions- sensors in the more proximal segment which may have responded to forces induced there by environmental interaction will no longer have such input. The dynamics of the environmental interaction that both the proximal and distal segments experience during dissociation may be dramatically different from the usual arrangement.

disconnected all neural information travel is unaffected, but the physical information is disrupted, and can be reassembled during behavior.

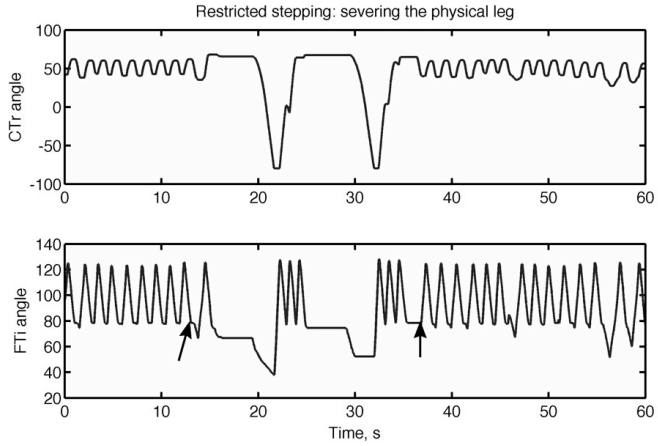


Figure 7-3. Disruption of the restricted stepping pattern by disconnecting part of the leg structure. Disconnection starts at the first arrow, and reconnection ends at the second arrow.

This experiment, which may be impossible in the animal, is perhaps one of the clearest demonstrations of the *neuromechanical* nature of this system, and those like it. The mechanics is a critical, integral part of system behavior, not just something to be overcome or compensated for by control. This is in contrast to a centrally coordinated system. If this experiment were run on such a system, the motor activities of the joints might remain in phase. This demonstration is far better shown in a physical robot than it would be in a computer simulation. Also, this result was first discovered accidentally as a result of a robot "malfunction"; this kind of serendipitous discovery is a benefit of working with physical model systems.

7.1.2 Preliminary forward stepping experiments

In the forward stepping experiments, motion of the ThC joint is allowed. With some sets of parameters, this is the only difference between forward and restricted stepping; all other parameters of the control system may remain the same. The three systems were able

to successfully generate forward stepping motion as shown in Figure 7-4. The action switching event parameters necessary for generating these motions are given in Table 7-1. In the absence of muscle models or other actuation-limiting control, the stick insect robot model tends to drift toward one or the other extreme of ThC movement and ceases behavior eventually, though it may operate stably for several minutes. When operating well as for the run that produced the data in Figure 7-5 and Figure 7-6, the leg spends about 0.8 s in each of stance and swing, and the foot moves about 30 cm rearward during each stance phase.

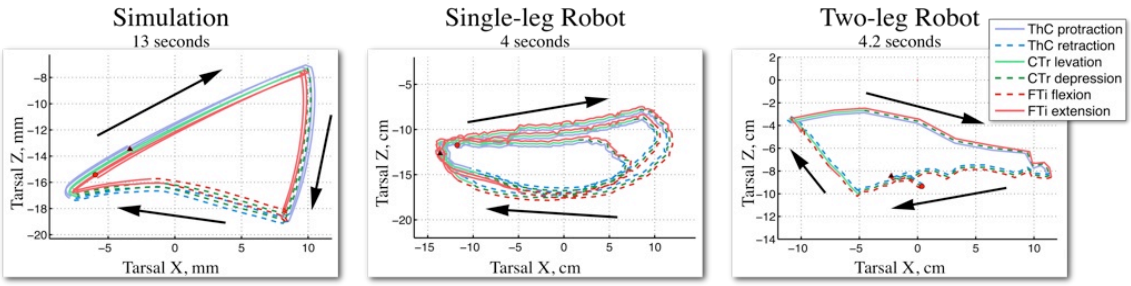


Figure 7-4 Path-state plots of forward stepping in three legged systems. The path which starts at the circle and terminates at the triangle represents the foot path as computed from recorded joint angle data. Each of the three lines along the path represents the action of one joint at that point, styled as shown in the legend. Coordinates are relative to the body; the origin is at the ThC joint. From (Rutter, et al., 2006)

Joint	State Transition	Joint Angles (degrees)		Ground Contact?
		ThC	FTi	
Original Ekeberg *et al*. Values				
ThC	PRO→RET	--	--	yes
	RET→PRO	--	--	no
CTr	LEV→DEP	--	< 70	--
	DEP→LEV	< -25	> 120	--
FTi	EXT→FLX	--	≤ 105	yes
	FLX→EXT	--	> 105	no
Single-Leg Platform				
ThC	PRO→RET	--	--	> 0.5
	RET→PRO	--	--	< -0.5
CTr	LEV→DEP		< 80	--
	DEP→LEV	< -25	> 115	--
FTi	EXT→FLX	--	≤ 100	> 0.2
	FLX→EXT	--	> 120	< -1
	EXT→EXT2†	--	< 80	<= 0
Two-Leg Platform				
ThC	PRO→RET	--	--	yes
	RET→PRO	--	--	no
CTr	LEV→DEP	> 35	< 48	--
	DEP→LEV	< -35	> 107	--
	DEP→LEV*	< -90	> 180	--
FTi	EXT→FLX	--	< 46	yes
	EXT→FLX*	--	< 0	yes
	FLX→EXT	--	> 90	no

Table 7-1 Action switching event conditions for forward walking in three legged systems. * Special, unreachable values when the other leg is in swing; prevents more than one leg being in swing at a time. † Additional state added to improve performance (see section 7.2).

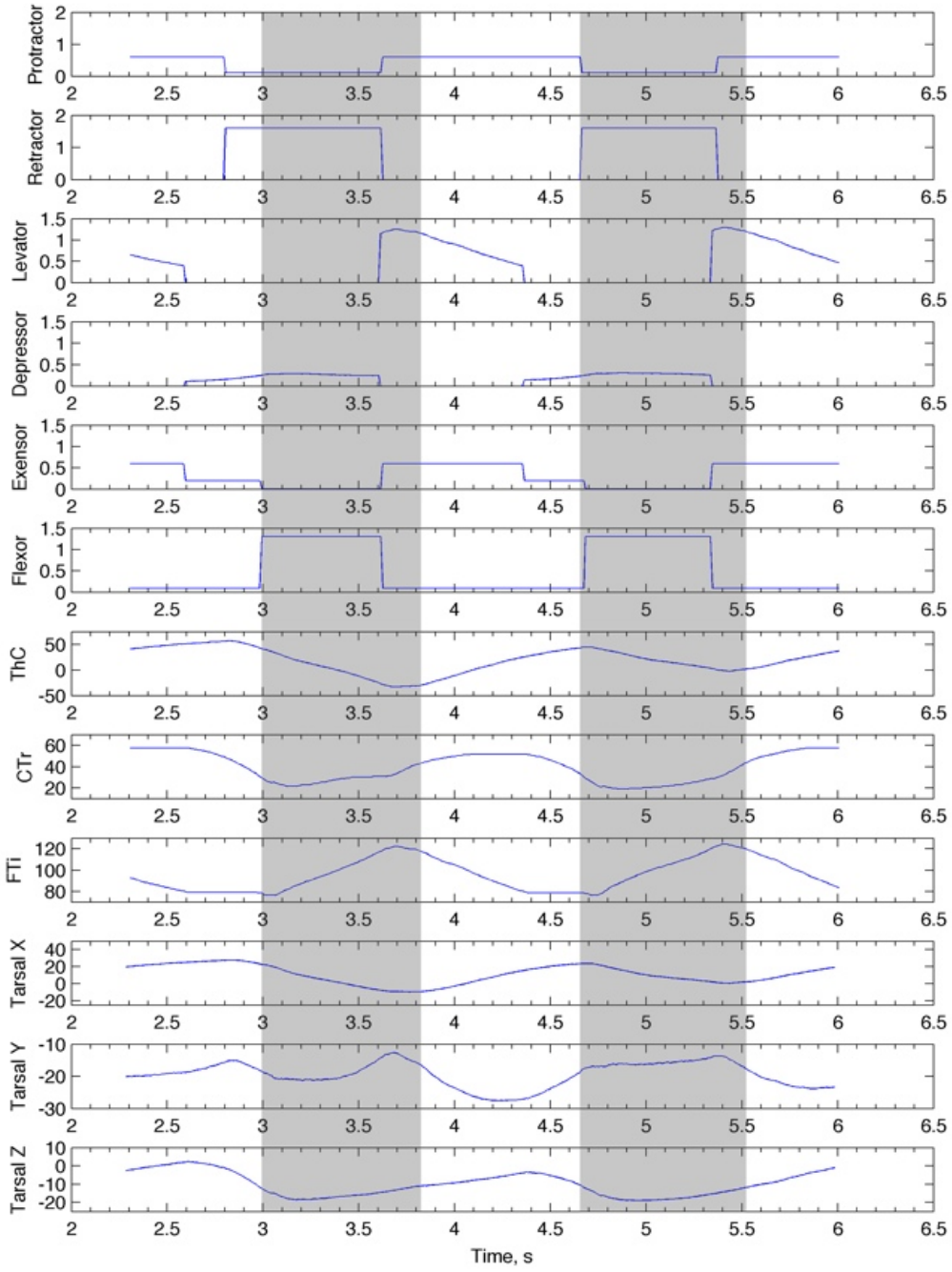


Figure 7-5. Sample data from forward walking in the single- leg platform with piecewise-constant FTi extensor muscle model, intended to parallel the presentation in Fig. 4B of (Ekeberg, et al., 2004) Shaded portions represent time the foot is touching the ground. Positive X is forward, negative Y is away from the body, and Z is up; dimensions are in centimeters. Origin is the point of the Thoraco-Coxal (ThC) joint. Note the differences between start of ThC retraction and start of FTi flexion, due to differing thresholds for the CTr load representing ground contact for those state changes. Also note the considerable lag between changes in stance-swing activity (i.e. FTi flexion to extension) and the time when the foot actually leaves the ground. From (Lewinger, et al., 2006)

Muscle activation levels are plotted with angle data and foot position in Figure 7-5 for the same run as shown in Figure 7-6; they are set similarly to the levels shown in Table 7-3. The lack of a constant-Z portion during stance in all these runs is due to a large amount of flexibility in the leg structure; precise foot position cannot be inferred accurately from measured joint angles. This information is not necessary, however, for SCASM to function.

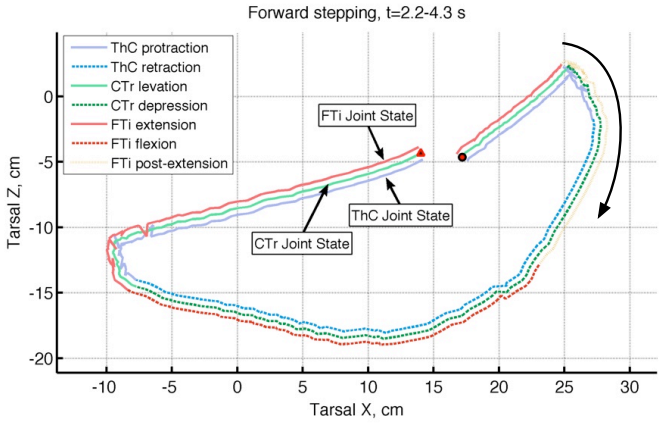


Figure 7-6. Path-state plot for a single step cycle in forward stepping of the single-leg platform. Positive X is forward; this is a projection of the 3-D foot path onto the sagittal plane.

7.2 Effects of muscle models on stepping behavior

Biomechanical studies strongly indicate that the performance of biological locomotor systems not only relies on the dynamic neuromuscular transform between the nervous and musculoskeletal systems of a walking animal, but can also profit from the contribution of specific muscle properties (Abbas & Full, 2000; Josephson, 1993). The control methods used by the nervous system are likely to rely on these complex properties of the associated mechanical systems. To the model builder, this presents a possibly critical element of structural accuracy in the construction of a leg model. To the

engineer of legged systems, it is important to determine what part of the dynamic properties of muscle are required for adequate system function.

As demonstrated in the physical disconnect experiment in the previous section, physical properties are critical to overall system behavior in a SCASM system. Coordination is a neuromechanical phenomenon, which we might expect to arise from the complete dynamics of the constituent modules and actions. Because muscles have mechanical substance and dynamics of their own, it is important to ask how accurately we need to model the muscle dynamical properties in order to create a sufficient behavioral match. This is also a useful question to ask in light of the extensive characterization and modeling of insect muscle activation and physical response dynamics done by (Guschlbauer, Scharstein, & Büschges, 2007) and many others.

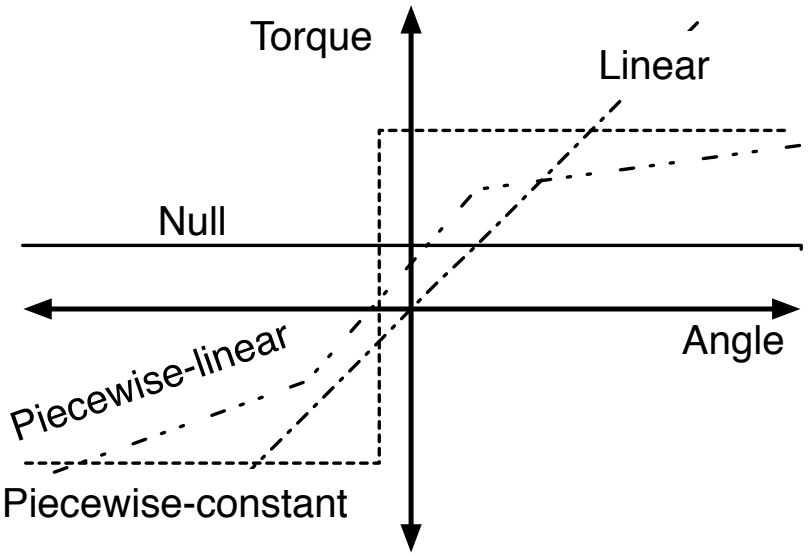


Figure 7-7 Torque vs. angle for various muscle model types. These graphs represent what would happen as angle changes if the muscles at the joint have a constant activation applied; this is the case for the ThC and FTi joints in the stick insect.

In this section three types of muscle models are tested on the FTi joint of the stick insect robot leg: null, piecewise-constant, and linear, as depicted in Figure 7-7. The FTi

joint was chosen because of its apparent role in instances where the robot failed to make the swing-stance transition. In the null muscle model, the motor torque output resulting from a model muscle has no relationship with joint angle. In the piecewise-constant model, this torque changes discontinuously at a specific angle. In the linear model, motor torque decreases linearly as the active model muscle contracts. Results show that adding very simple models of muscle properties at a single joint cause marked improvement in the performance of a neurally-based step generator for a 3-degree-of-freedom robotic leg. Much of this section, including figures, was first published in (Rutter, Lewinger, Blümel, et al., 2007)

The action switching conditions used for these experiments are shown in Table 7-2; muscle activations are based on the current state for each joint. These activations, shown in Table 7-3, are constant for muscles at the ThC and FTi joints. At the CTr joint the activations are dynamically set based on known activation magnitude reflexes, as was done in (Ekeberg, et al., 2004).

JOINT ANGLE CONDITIONS FOR STATE CHANGES

Joint	State Transition	Joint Angles (degrees) ThC	FTi	CTr Load
		Sideways Stepping		
ThC	→RET	--	--	> 0.5*
	→PRO	--	--	< -0.5
CTr	→DEP		< 80*	--
	→LEV	< -25	> 115*	--
FTi	→FLX	--	≤ 100	> 0.2*
	→EXT	--	> 120*	< -1
	→EXT2†	--	< 80	<= 0
		Forward Stepping		
ThC	→RET	--	--	> 0*
	→PRO	--	--	< -0.5
CTr	→DEP	--	< 75*	--
	→LEV	< -20	> 120*	--
FTi	→FLX	--	≤ 100	> 0*
	→EXT	--	> 105*	< -1
	→EXT2†	--	< 80	<= 0

Table 7-2 Action switching event conditions. * Condition differs between restricted and forward stepping. † Additional state implementing the piecewise-constant muscle model. This state is only enabled for tests of that model.

MUSCLE ACTIVATION LEVELS BASED ON JOINT STATE

Joint	State	Agonist Activation	Antagonist Activation
ThC	PRO	0.6	0
	RET	1.6	0.1
CTr	LEV	Eq. (1)	0
	DEP	Eq. (2)	0
FTi (sideways)	EXT	0.6	0.09
	FLX	1.3	0.001
	EXT2	0.2	0.09
FTi (forward, piecew. const.)	EXT	0.8	0.09
	FLX	6	0.001
	EXT2	0.15	0.09
FTi (forward, linear)	EXT	0.8	0.09
	FLX	1.7	0.001

Table 7-3 Muscle activation levels for the various joint actions.

The variable activations in Table 7-2 are given by:

$K_1(FTi\_angle)$ Equation 7-1

$2*K_2(FTi\_angle) + K_3(CTr\_angle)$ Equation 7-2

$K(x) = \max[0, (a + b \bullet p(x, wmin, wmax) + c \bullet q(x, wmin, wmax)]$ Equation 7-3

$p = x\text{-}wmin/(wmax\text{-}wmin); q=1\text{-}p$ Equation 7-4

ACTIVATION FUNCTION PARAMETERS

K	a	b	c	wmin	wmax
1	0	0.8	0.01	61	100
2	.08	-0.205	0.008	61	100
3	0	0	0.06	-40	80

Table 7-4 Activation function parameters

Three types of muscle model were used in these experiments: null, piecewise-constant, and linear. Muscle model computations and force control were done as described in section 6.2.1. For all of the modeling in these experiments, force-velocity and passive stiffness functions were set to their constant "no effect" values (1 and 0 respectively). Thus the muscle models here have only a force-length characteristic, if any. All robotic models, of course, have the implicit force-velocity relationship mentioned in the model description, as a result of low-level motor and control dynamics.

The muscle model parameters for the various tests are shown in Table 7-5. There is no FTi_EXT2 muscle. The EXT2 state just sets activations differently for the FTi extensor and flexor muscles. The FTi post-extension (EXT2) state is in effect a piecewise-constant force-length characteristic for the FTi extensor muscle. This is not precisely correct since it is only in effect during extension, but putting it into effect during flexion would likely have little effect due to the low extensor activity during that action.

MUSCLE MODEL PARAMETERS

Muscle	l_0	r	α	f_{max}	a	b
Sideways Stepping						
ThC_RET	--	-0.5	--	10	--	--
ThC_PRO	--	0.5	--	10	--	--
CTr_DEP	--	-1	--	10	--	--
CTr_LEV	--	1	--	10	--	--
FTi_FLX	--	1	--	10	--	--
FTi_EXT	--	-1	--	10	--	--
Forward, Piecewise Constant						
ThC_RET	--	-0.5	--	15	--	--
ThC_PRO	--	0.5	--	15	--	--
CTr_DEP	--	-1	--	10	--	--
CTr_LEV	--	1	--	10	--	--
FTi_FLX	--	1	--	20	--	--
FTi_EXT	--	-1	--	10	--	--
Forward, Linear						
ThC_RET	--	-0.5	--	18	--	--
ThC_PRO	--	0.5	--	15	--	--
CTr_DEP	--	-1	--	12	--	--
CTr_LEV	--	1	--	15	--	--
FTi_FLX	5	1	95	30	-1.5	2
FTi_EXT	5	-1	100	14	-1.4	2

Table 7-5 Muscle model parameters. In sideways stepping, the same parameters are used for the null and piecewise-constant models; only the EXT2 state is added.

7.2.1 Sideways Stepping with different muscle models

We first tested the operation of the stick insect robotic model in the restricted single leg preparation, as described in section 7.1. We compared the operation of this system using the null and piecewise-constant muscle models for the FTi joint only. In development of the system, it became apparent that one of the most problematic aspects of stepping control is achieving a reliable swing-stance transition.

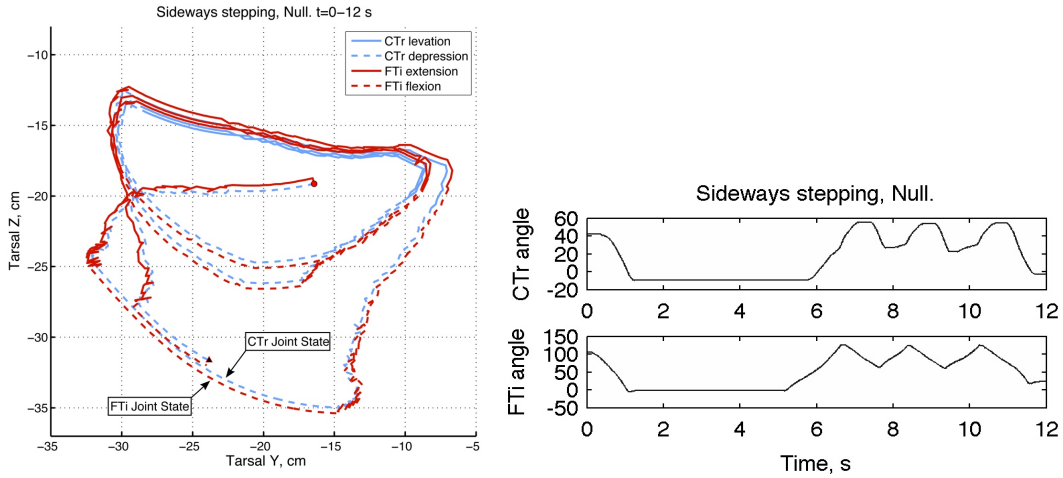


Figure 7-8 Path-state plot (left) and joint trajectories in degrees (right) of sideways stepping with null muscle model. This system ceases motion after one second due to postponed detection of ground contact, and must be brought back to the feasible range of joint angles by hand. It again detects ground contact too late at about 11.5 seconds.

On this platform the signal used to detect ground contact and complete this transition is load at the CTr joint (See Figure 6-6 and Figure 4-21). Therefore a measurable increase in load is critical for switching to the stance phase. However, with the FTi joint extending, this increase in torque can be very small, since FTi extension is acting to bring the foot up while CTr depression acts to bring the foot down, and the sliding foot does not firmly engage the substrate upon contact. As shown in Figure 7-8, this can result in delayed detection of ground contact, which disrupts the stepping cycle and eventually brings the system to a halt. Introduction of the piecewise-constant muscle model at the FTi extensor makes the swing-stance transition much more reliable, because the lower FTi extension force around the time of ground contact increases ground contact force and decreases likelihood that the FTi joint will overextend.

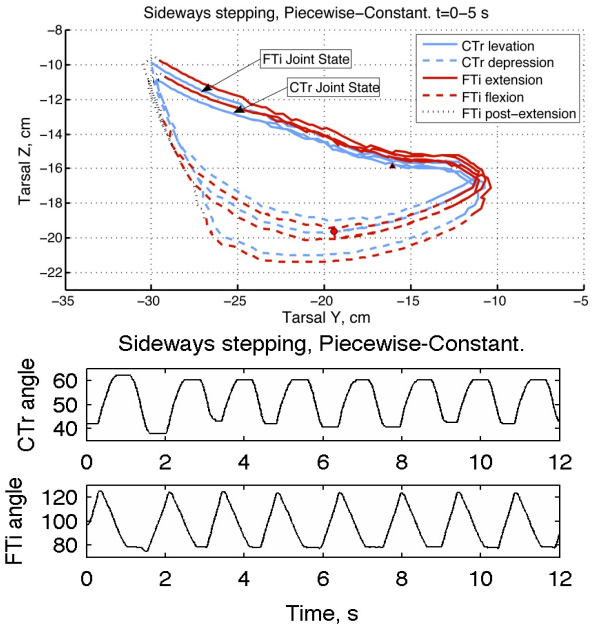


Figure 7-9 Path-state plot and joint trajectories with the piecewise-constant muscle model, implemented using a third FTi "Post Extension" (EXT2) state. In this example the system quickly stabilizes, although "ground contact" was apparently detected before actual foot contact during two of the first three step cycles, which are depicted in the top plot.

7.2.2 Forward stepping with different muscle models

In forward stepping, motion of the ThC joint is allowed. This is the primary difference between this and restricted stepping; most other parameters of the control system can remain the same and still produce stepping, though in this case they were tuned to produce better behavior as seen in Table 7-2.

Forward stepping with the null muscle model usually fails to detect the ground and transfer to stance muscle activations on the first step; therefore the histograms and statistics shown in Figure 7-10 compare only the piecewise-constant and linear FTi muscle models. These histograms were generated from analysis of ten minutes of stepping for each of the two types of muscle model. In each case it was possible to generate reliable stepping behavior, though it took more effort to find a set of control parameters that accomplished this for the piecewise-constant case; this appeared to be

due to a higher sensitivity of the system to the transition parameters in that case. Additionally, leg motion when using the linear FTi muscle model appears smoother, but this has not been quantified.

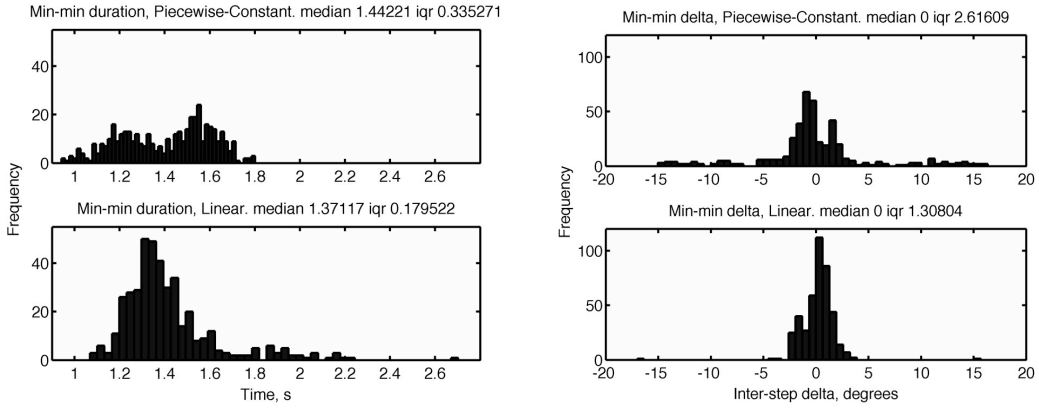


Figure 7-10 (Left) Histograms showing the distributions of durations of the entire step cycle, as measured from one minimum of ThC angle to the next. The medians do not differ significantly according to the Wilcoxon test (p=0.22), and the distributions differ with $p = 1x10^{-8}$ according to the Kolmogorov-Smirinov test. Note the bimodal behavior in the piecewise-constant data. (Right) Histograms showing the distributions of difference of the minimum ThC angle from step to step. The medians do not differ significantly according to the Wilcoxon test (p=0.14), but the distributions differ with $p = 3.7x10^{-5}$ according to the Kolmogorov-Smirinov test. Additionally, the runs test indicates that all of the underlying distributions are most likely random (rather than consisting of alternating highs and lows) for all but the min-min delta for the linear muscle model ($p < 0.001$). The lack of the wide tails on data taken using the linear muscle model indicates a more consistent placement of the foot at transition from stance to swing.

7.2.3 Muscle Modeling Conclusions

In (Lewinger, et al., 2006) it was suggested that the state transition modules of SCASM should consist of actions which saturate, or limit actuation capability towards the extremes of the range of motion. This was suggested in order to generate the desired stable repetitive motion sequence. Here we have demonstrated that a linear reduction in actuation force with displacement at only the FTi joint significantly conditions the emergent motion of the system. It tightens the distribution of foot liftoff positions, which one might expect to be unaffected since that event is most tightly coupled to the ThC joint. It also reduces the spread of step durations, while making the distribution of step

durations unimodal. It is worth noting that the linear models here which create such significant improvements are used only at the FTi joint, while the improvements are measured at the ThC joint. This further indicates the inter-joint sensory and mechanical influences can carry changes in the dynamics of one joint throughout the behavior of the entire system. These linear models are also simpler than the piecewise-linear models used in the two-leg platform of (Lewinger, et al., 2006), as depicted in Figure 7-7.

These results suggest that the required complexity of actuation-limiting models for stabilizing and conditioning SCASM-based control may be relatively simple in general. This is a further example of the low computational complexity necessary for SCASM control, and suggests that application to a broad range of control problems (and underlying actuation mechanisms) may be both conceptually and practically straightforward for systems which require cyclic sets of coordinated state transitions.

Additionally, it is apparent that adding complexity or conditioning of the underlying plant at one critical module or in one critical action, e.g., by improving the associated muscle model, can significantly modify dynamics of the entire system without further computational effort associated with other modules. This may simplify the task of the system designer, who might identify modules involved in a critical behavioral point, and concentrate on those. It also suggests that modulation of overall system behavior such as that necessary in turning or climbing could be implemented as modulation of one or a few critical modules from a higher-level control mechanism. This ties in very well with the project modeling goal of investigating descending influences on local control, which is addressed in the next two sections.

7.3 Changing speed

Coordination in SCASM is driven by sensory switching events, and muscle activations affect how quickly these events occur following a switching of actions. It was hypothesized that tonic changes in these muscle activations would be sufficient to modify the overall stepping speed. The representation in SCASM has a different set of muscle activation rules for each action; this corresponds to a pattern-generator gating of tonic drive to the motor neurons. To generate changes in speed as shown here, only the agonistic muscle activations were changed. In the FTi joint, for example, this would be the flexor activation during the FLX action, and the extensor during EXT. Note that this results in an increase in motor activity during both swing and stance. This is not quite the same as what is known from biological observations, which suggest that swing-phase motor excitation and limb speed remain the same over a broad range of stepping frequencies. This was found to be successful for both the stick insect and cockroach robot models. The results for the cockroach model are shown here; this was done using an early configuration with null muscle models.

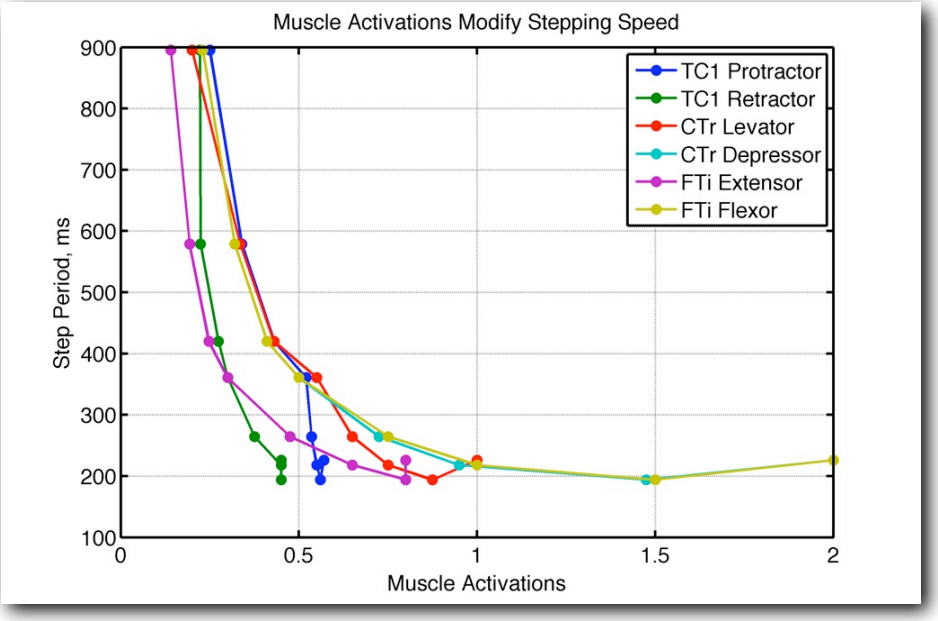


Figure 7-11 Muscle Activations vs. Step Period. Note that the appropriate increase in drive is not generally proportional between muscles, and that for each muscle pair a greater increase in activation was required for the stance agonist. As presented in (Rutter, Bender, Taylor, et al., 2008)

As seen in Figure 7-11, we modified the agonistic drive to all six virtual muscles, producing a decrease in step period; which is an increase in speed. These changes in tonic activation to motor neuron equivalents are sufficient in this reduced system. Note that this agrees with prior motor neuron activity descriptions (Gabriel & Büschges, 2007; Watson & Ritzmann, 1998b), but those studies do not definitively address whether the motor patterns observed are the result of gated tonic drive or changes in local sensory magnitude influences. The associated biological hypotheses are fairly difficult to test outright in the animal. Such tests would require the ability to modify either the tonic signals controlled here or the gating function associated with the joint pattern generators. Model failures might be used to more narrowly guide investigation; some preliminary work is discussed in section 9.3.3.

7.4 Transitions from walking to turning

Transitions between one stable pattern of stepping motion and another in these models can require a set of coordinated changes in the magnitude or sign of the feedback at multiple points within the SCASM control system. In the robotic models, these points are the Boolean action switching rules and switching event thresholds, which represent synaptic strength or neuronal excitability in the animal. This section investigates the effects of the relative timing of these changes on the stability of the leg-level behavioral output during transitions between forward walking and turning movements in our model of the cockroach middle leg. Modulating only a subset of these feedback pathways can lead to appropriate intermediate behavioral transitions in some cases. However, it is hypothesized that there are some sets of changes in feedback parameters which must occur almost simultaneously in order to avoid unstable or inappropriate behaviors. Since a primary focus of this work is to use the model to reveal hypotheses regarding the biological system, the data from these timing investigations and the configurations for the various behaviors are used to provide focused input guiding the selection of the most effective biological experiments necessary to test the hypotheses which have arisen from this modeling work.

7.4.1 Behaviors

These experiments involve transitions between three distinct behavioral regimes of stepping. A very coarse description of the differences between walking and inside turning in the middle leg was used: “FTi extends during swing and flexes during stance, as opposed to the opposite during forward stepping”. In the 3-DOF cockroach model there are two qualitative behaviors that fit this; they differ in the phase relationship between the

activity of the CTr joint and stance. Though the muscle activations are not known for all cases both have been observed kinematically (Mu & Ritzmann, 2005)(Bender, pers. comm.) All three behaviors are described below. Note that these are the second-simplest behavioral arrangements possible, the stance and swing actions at CTr and FTi switch at the same time, which need not be the case (see the stick insect forward walking case in Figure 4-8). The simplest behavior would switch between two global states, but this is unlikely to represent anything useful about the generation of these leg behaviors in the animal.

7.4.1.1 Forward Stepping

In forward stepping the FTi joint extends and the CTr depresses during stance, as shown in Figure 7-12. The event-space diagram that implements this behavior is shown in Figure 7-13. This configuration is described in detail in section 4.4.2.

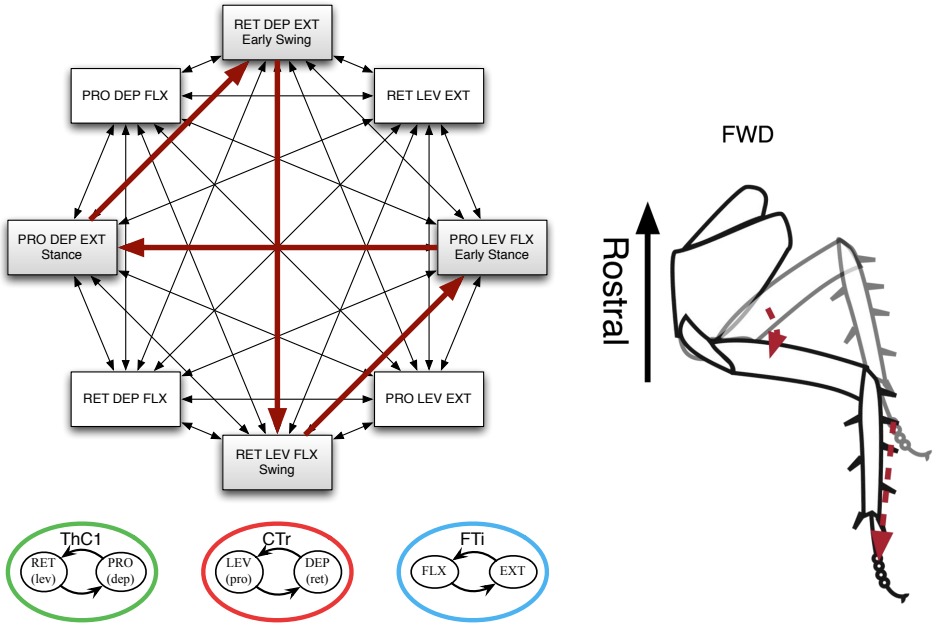


Figure 7-12 Cockroach middle leg Forward (FWD) global state diagram (left) and behavior (right). Beginning of stance shown in grey, end in black; foot path and CTr angle change are marked with the red arrows. Coxa motion ignored. The actions for the ThC1 and CTr modules are double-labeled as described in section 4.4.2; labels in parentheses indicate the actual effect of this action on the leg.

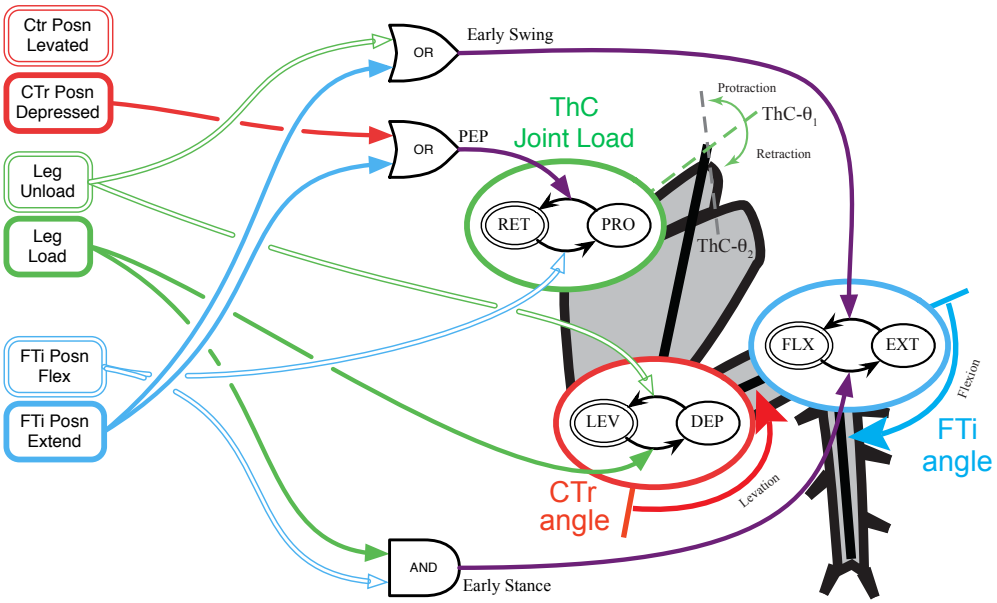


Figure 7-13 Event space diagram for cockroach forward stepping. This configuration is described in detail in section 4.4.2.

7.4.1.2 Inside Turn, Forward-type

The first type of inside turn is Inside Turn, Forward, or ITF. In this behavior, the FTi flexes during stance and extends during swing, but the rest of the behavior is qualitatively the same as in forward walking. All of the global states therefore just change the FTi state from that in forward stepping, as shown in Figure 7-14. The event-space diagram which implements this behavior is shown in Figure 7-15, marked with the HDSO SCCILS parameters that must change from the FWD configuration in order to accomplish this, together comprising the configuration changeset ΔITF. The load influences on FTi actions and the FTi angle effect on ThC1 actions have been *reversed*. The thresholds of the effects of FTi angle on FTi actions and CTr angle on ThC1 actions have also changed. Though not explicitly related to this behavior, reversal and modification of local sensory influences via descending commands in *B. discoidalis* was shown in (Mu & Ritzmann, 2008a).

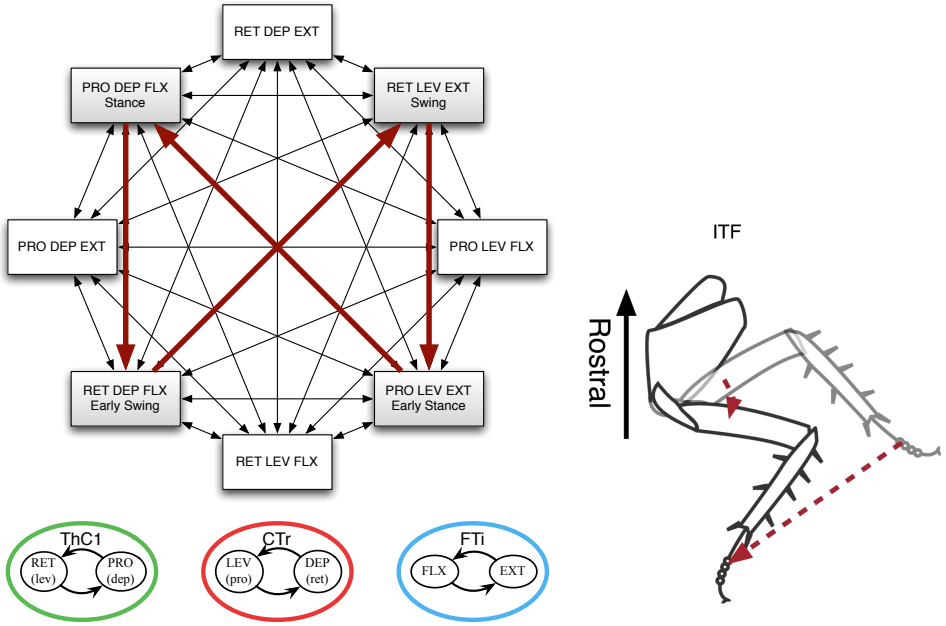


Figure 7-14 Cockroach middle leg Inside Turn, Forward (ITF) global state diagram (left) and behavior (right).

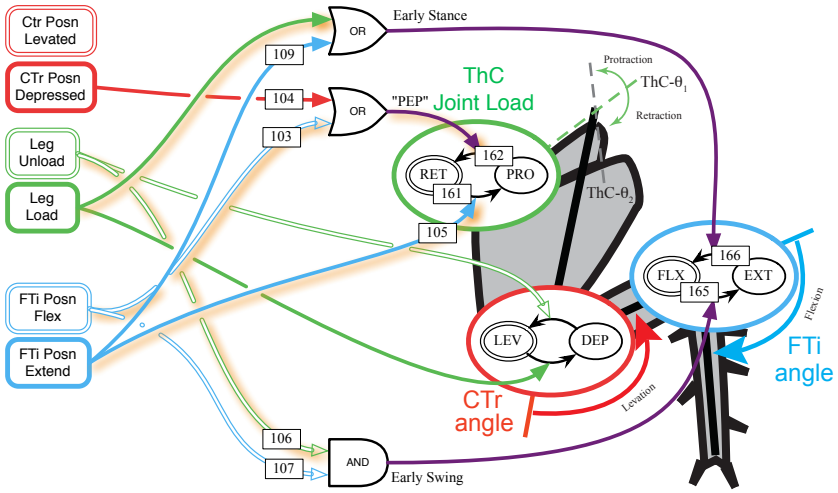


Figure 7-15 Event space diagram for cockroach ITF behavior. Changed switching events are marked with a yellow highlight; all changed HDSO parameters are noted in boxes attached to switching event lines or transition paths. The effect of load on FTi and the effect of FTi angle ThC have both been reversed.

7.4.1.3 Inside Turn, Backward-type

The backward type of inside turn, ITB, is shown in Figure 7-16; here the CTr levates during stance and depresses during swing. Note that the global state diagram is the

reverse of that for forward walking; indeed this diagram describes backward walking as well. The configuration difference between backward walking and ITB can be only a change in muscle activations. The event-space diagram for this behavior is shown in Figure 7-17, along with the parameters which must be changed to go from ITF to ITB: ΔITB. The changes here are a reversal of load/ground contact influence on CTr, and a reversal of CTr angle influence on ThC.

The CTr position influence has relatively little effect in these experiments; it can disrupt the behavior if configured incorrectly, but little change is noticeable if it is removed completely. These influences were left intact since they are a product of the translation from the stick insect controller, the source of the controller for cockroach forward stepping behavior.

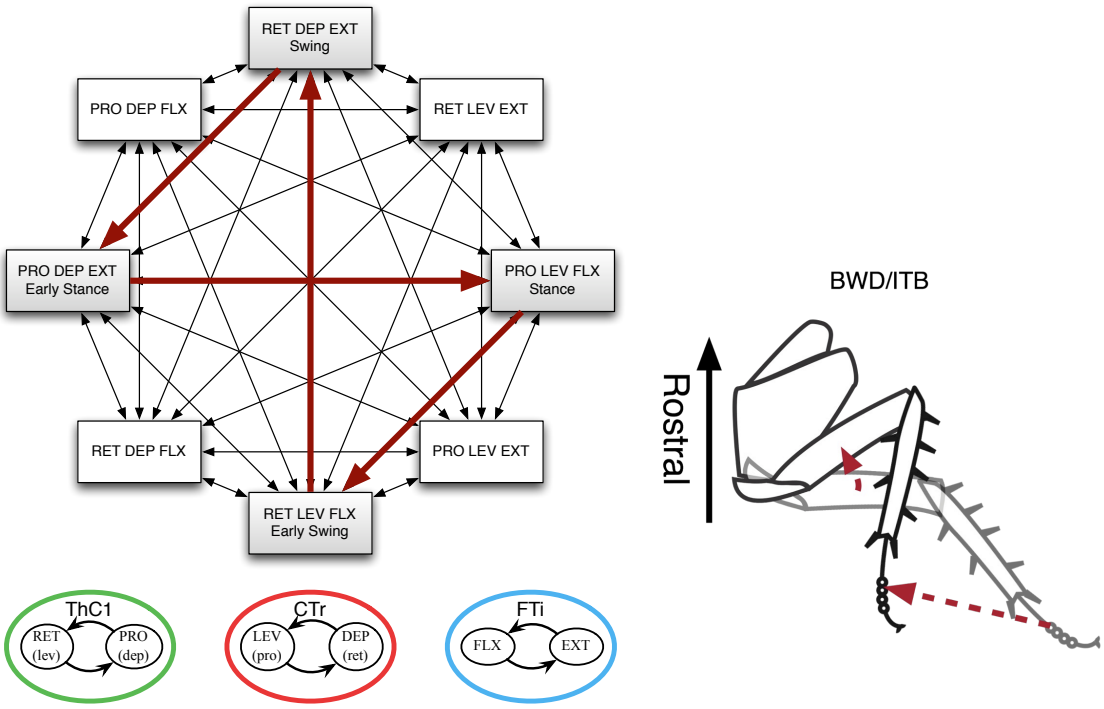


Figure 7-16 Cockroach middle leg Inside Turn, Backward (ITB) global state diagram (left) and behavior (right).

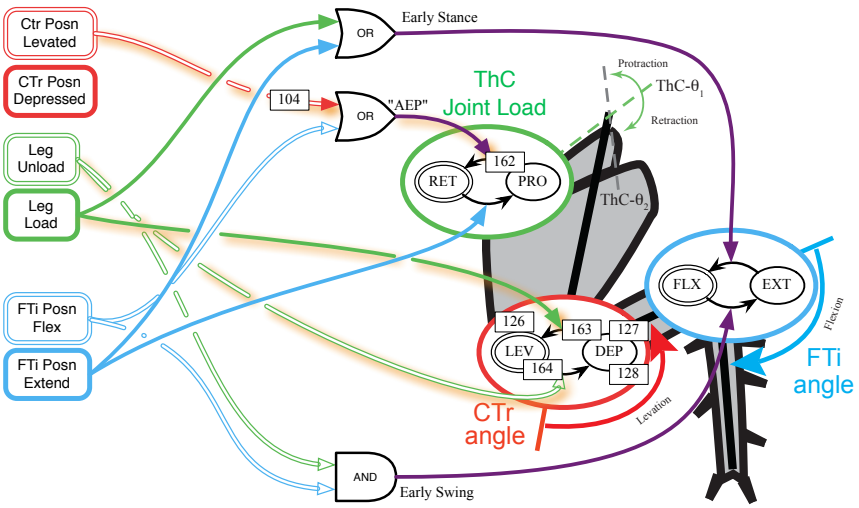


Figure 7-17 Event space diagram for cockroach ITB behavior. Switching events which have changed from ITF are marked with a yellow shadow; all changed HDSO parameters are noted in boxes attached to switching event lines or transition paths. The effect of load on CTr and the effect of CTr angle ThC have both been reversed.

7.4.2 Hypotheses and Experiments

There are many experiments which might be run on the model to look at different ways of transitioning between behaviors. Several experiments were chosen as the most directly elucidating, and are described below. The building blocks for describing these experiments are described above and repeated here: FWD, ITF and ITB are the forward, inside-turn forward and inside-turn backward behaviors, respectively. ΔITF is the set of parameter changes necessary to go from FWD to ITF, and ΔITB is the changeset which shifts ITF to ITB. The SCCILS HDSOscript which describes experiment A contains these elements, and is included in the SCCILS appendix 10.2.

7.4.2.1 A: Basic Behavioral Transitions

A simple interesting hypothesis is that the parameters which differ between the behaviors can be changed during operation, resulting in transitions between them. This is tested in experiment A (the right path of Figure 7-18) which starts in FWD for 20

seconds, applies ΔITF and waits for another 20 s, then applies ΔITB and waits for another 20 s before stopping.

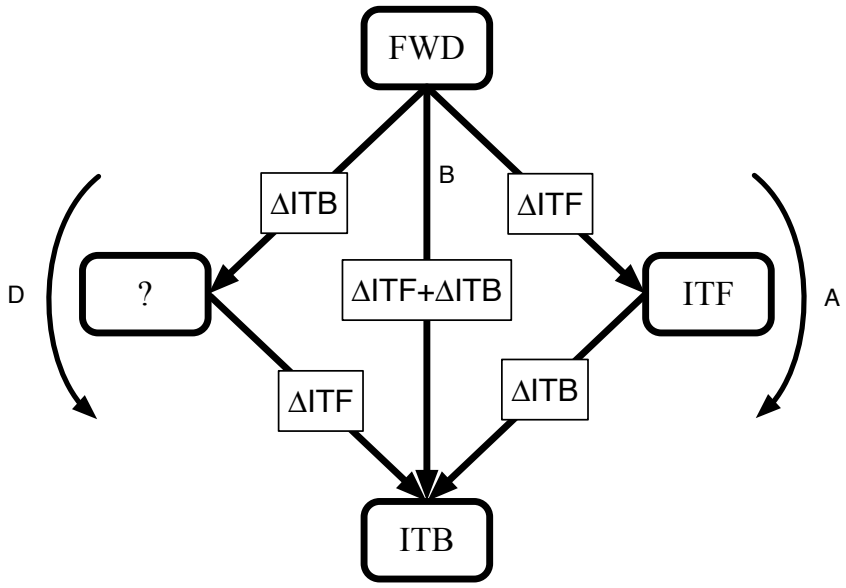


Figure 7-18 Diagram of experiments A, B and D. The path on the right side consists of the three behaviors and change sets listed above. This progression on the right is experiment A. The path down the middle applies both changesets simultaneously, and is experiment B. The path on the left reverses the order of application of the changes, and is experiment D.

7.4.2.2 B: Combined Changesets

If A works, the next simple hypothesis is that combined changesets work, and shift to the expected final behavior. This simultaneous application of ΔITF and ΔITB is experiment B, shown in the center path of Figure 7-18.

7.4.2.3 D: Reversal of Incremental Changesets

This experiment tests two hypotheses: D1) a changeset applied to the incorrect beginning behavior may not generate a stable behavior. D2) even if D1 is true, if another changeset is applied which then defines a valid combined changeset, the result will be the expected final behavior. Experiment D tests these hypotheses by starting in FWD, then applying ΔITB before applying ΔITF, as shown in the left path of Figure 7-18.

Note: experiment C, which involves inverting the changesets and running them backwards, did not clearly address any novel hypotheses and was not run.

7.4.2.4 F: Some Subsets Don't Work

Hypothesis: For a configuration changeset, there may be subsets of the change which do not produce a stable interim behavior no matter what the order; therefore these changes must be applied in near synchrony. To examine this, ΔITF was broken up into subsets associated with each joint affected by the change; ΔITF_{FTi} and ΔITF_{ThC1}. Experiments F apply these sub-changesets in both orders as shown in Figure 7-19.

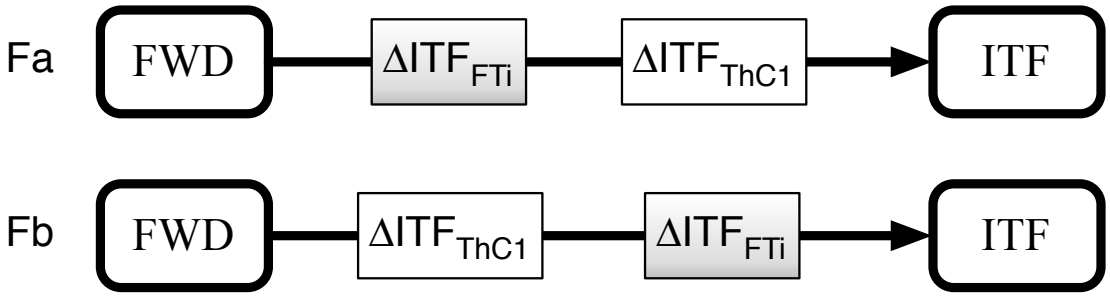


Figure 7-19 Diagram of changesets applied in experiments F.

7.4.2.5 G: Gradual Changes in Transition Thresholds

Many of the parameters involved in these behavioral configuration changesets are continuously variable. One question is then: can these be changed gradually, and if so, under what restrictions and in what relationship to the changes in discrete parameters? Based on experience operating the models, hypothesis G states: for continuously modifiable parameters associated with defining the switching events, there will be a timing relationship of continuous modification and transition rule switching that produces stable/appropriate behavior. Some timing relationships may not produce such behavior.

Experiment G tests this hypothesis by breaking up ΔITF into continuous and discrete elements. ΔITF_{thresh} consists of the continuous thresholds that fully specify the sensory events that are part of ΔITF. It is applied gradually in 10 linear steps over 10 seconds. ΔITF_{trans} consists of the non-continuous Boolean transition rules, and is applied at various single times during and slightly after ΔITF_{thresh} in experiments Ga Gb and Gc, as shown in Figure 7-20.

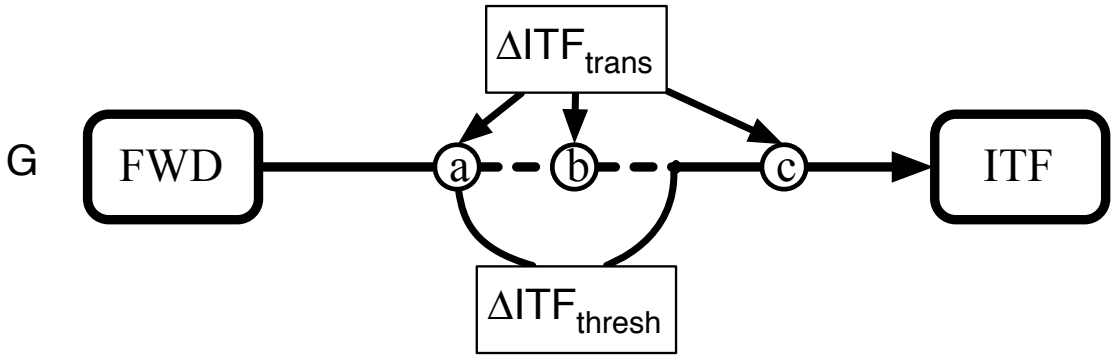


Figure 7-20 Diagram of change sets applied in experiments G. ΔITF_{thresh} is applied in equal increments once per second; ΔITF_{trans} is applied at the beginning, middle, and after the end of this gradual application.

7.4.2.6 E: Gradual Changes in Action Dynamics

In the process of interactive model operation, it was observed that some action dynamic properties may be changed significantly and still produce cyclical behavior, however there is usually a limit to this range. Based on this experience, hypothesis E states: for continuously modifiable parameters associated with the dynamics of an action, in some cases it will work and in others it will not. This hypothesis is tested in experiment E, shown in Figure 7-21. It is similar to experiment G, but the continuously variable changeset ΔITB_{act} consists of the muscle activation parameters which change in ΔITB. ΔITB_{trans} includes the sensory threshold change for the CTr influence on ThC1 in addition to all the Boolean transition rules.

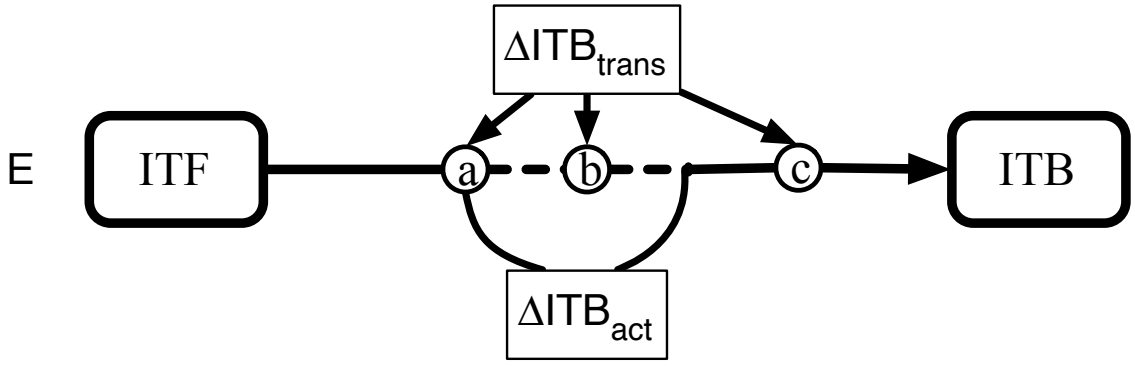


Figure 7-21 Diagram of change sets applied in experiments E. ΔITB_{act} is applied in equal increments once per second; ΔITB_{trans} is applied at the beginning, middle, and after the end of this gradual application.

7.4.3 Experimental Methods

An important part of preparing for these experiments was finding minimal change sets between behaviors. First a configuration that works for each behavior was partially derived from the event-space diagram, and then interactively tuned. Configurations for each of these behaviors were then recorded as an HDSOscript. The changesets were found by locating the differences between these scripts and identifying the associated parameter changes. To generate a script that applies a changeset, that set of HDSOmods was entered in the script with the same implementation time.

To estimate the *minimum* change set, each of the parameter changes was interactively rolled back to the value from the pre-change behavior; each one that substantially changed behavior was deemed necessary. Caveats: to be exhaustive all subsets of the changes would need to be tested, rather than doing it one by one as described here. Also, sometimes rescinding a change will break the post-change behavior, but applying that single parameter change to the pre-change behavior has no apparent effect. This may be because one behavior is more sensitive to the values of this parameter in this range. To minimize the changesets, FWD was modified slightly to avoid the need to change some

parameters in the change to ITF. This introduces a trade off between selecting "minimal changesets" and the best-tuned behaviors.

Experiment scripts were generated, then run using the enginecli command line program from the NeuRoMod software suite. Video data was taken with a DV camcorder, compressed and recorded directly to disk using QuickTime Pro. Data were plotted using the SCCILStoolbox set of MATLAB programs, primarily the *jlrelegplots* and *jlpathstateplot* functions. Notes on the experimental runs are recorded in pages 38-41 of the Brandon Rutter Biorobotics Lab Notebook #3. Data is labeled by experimental repetition as done there; for example Fa2 is the second repetition of experiment F, sub-experiment a.

7.4.4 Transition to Turning Results

This section contains results and brief discussion for each of the experiments described above in section 7.4.2.

7.4.4.1 A: Basic Behavioral Transitions

Three repetitions of this experimental script (depicted in Figure 7-18) were run, and they all produced results similar to those shown below. Full sensor and state data is shown in Figure 7-22, path state plots of the various behaviors in Figure 7-23, and transitions between them are shown in Figure 7-24 and Figure 7-25. Transitions between behaviors can occur smoothly during system operation by changing the necessary parameters all at once.

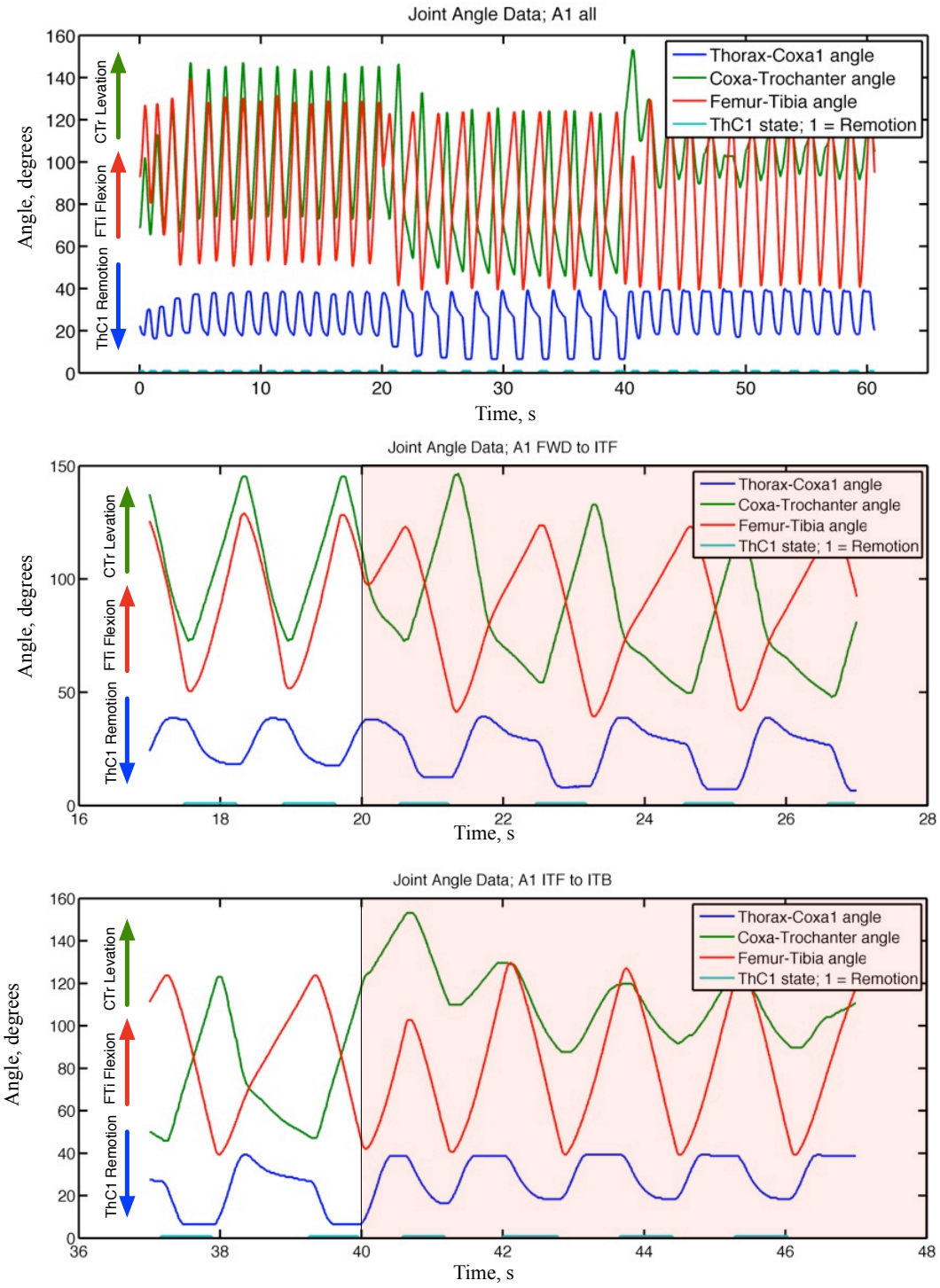


Figure 7-22 Joint angle data for experiment A1, as well as state for the Thorax-Coxa1 DOF; remotion is highly correlated with swing. Top is the full experiment, starting forward stepping from a standstill. The changeset ΔITF is applied at 20 s, and brings the system to the inside turn-forward behavior. ΔITB is then applied at 40 s, leading to inside turn-backward behavior. The lower two plots show detail of these transitions, with the shaded regions covering behavior after the configuration change. Note the changes in phase between joint angles in each case. There is transient behavior at the startup and switching points, but under experimental conditions these configurations led to stable stepping behavior.

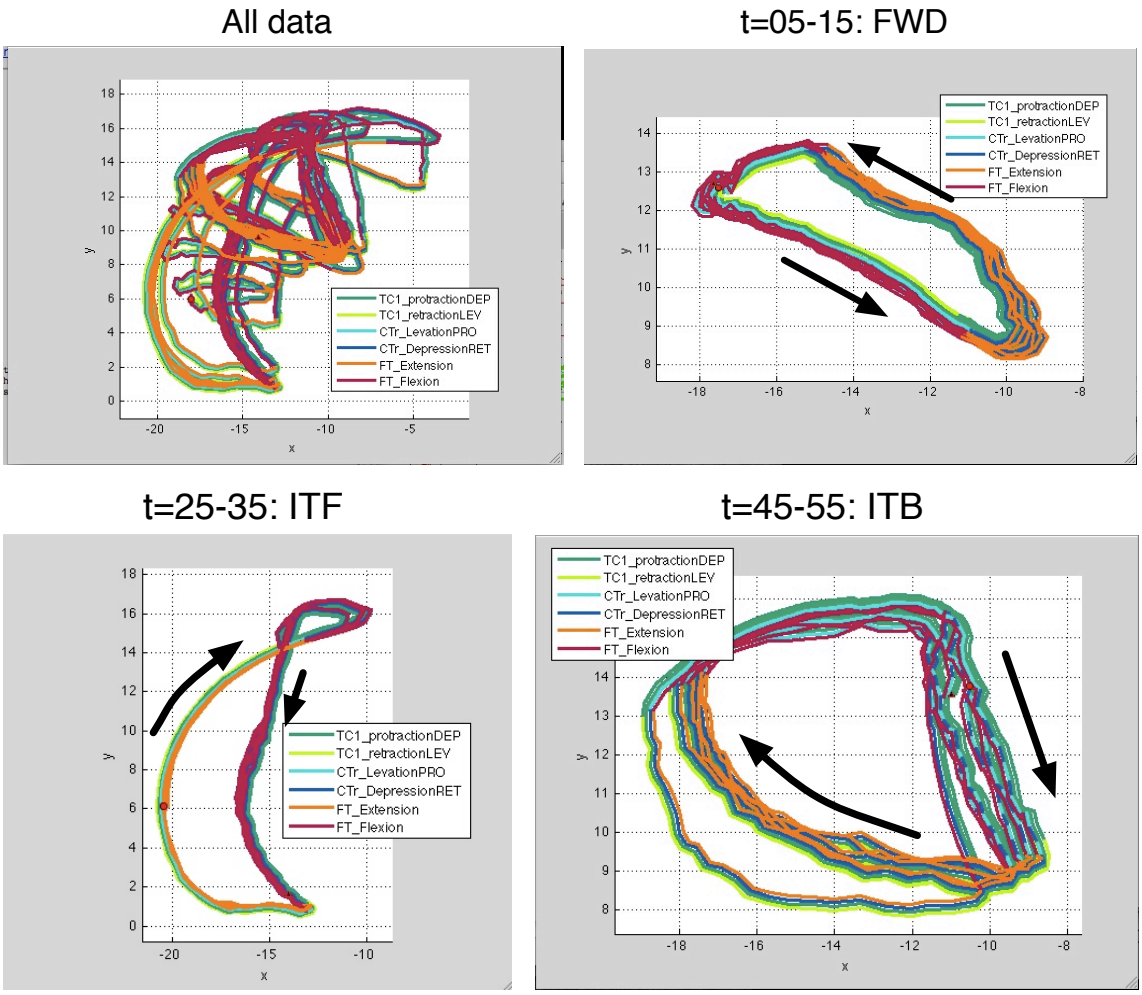


Figure 7-23 Path state plots of all data and each of the stable behaviors: Forward, inside turn-forward (ITF) and inside turn-backward (ITB). Data from the same experimental run as in Figure 7-22. These paths are viewed from above; the virtual animal would be facing right, with the body midline near and parallel to the x axis. TC1_retractionLEV is strongly correlated with swing phase. There are four parallel lines in these path-state plots because there are four motors, even though the line for the ThC2 "joint" is meaningless and constant in a fake "TC1_protractionDEP" state.

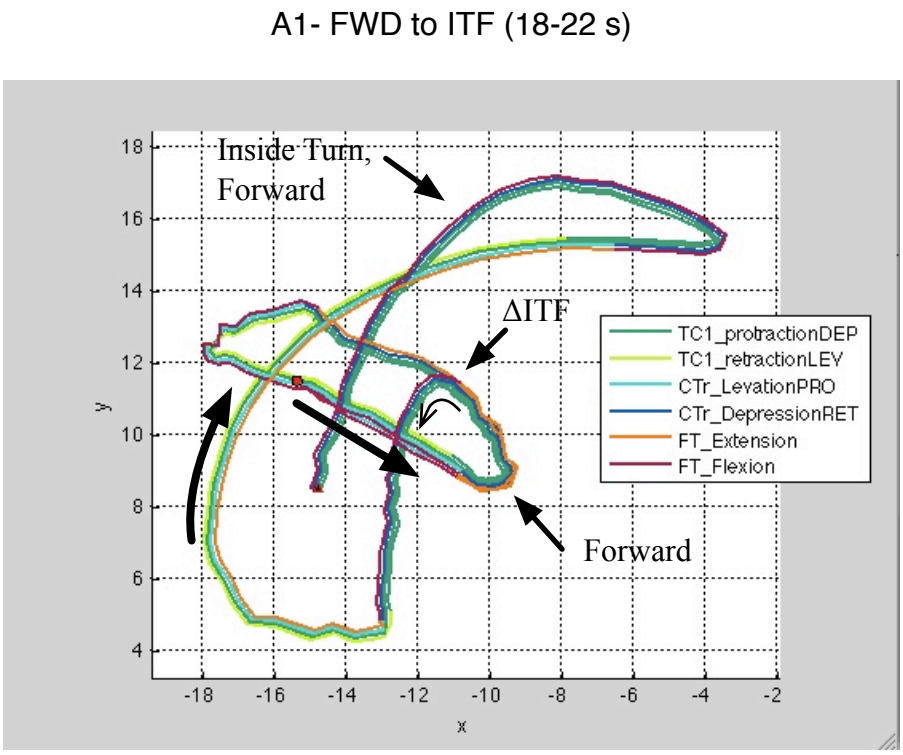


Figure 7-24 A1- FWD to ITF (18 s to 22 s). Largest and smallest arrows indicate path direction, labeled arrows point to the stable behaviors and the location of the foot at 20s when ΔITF is applied.

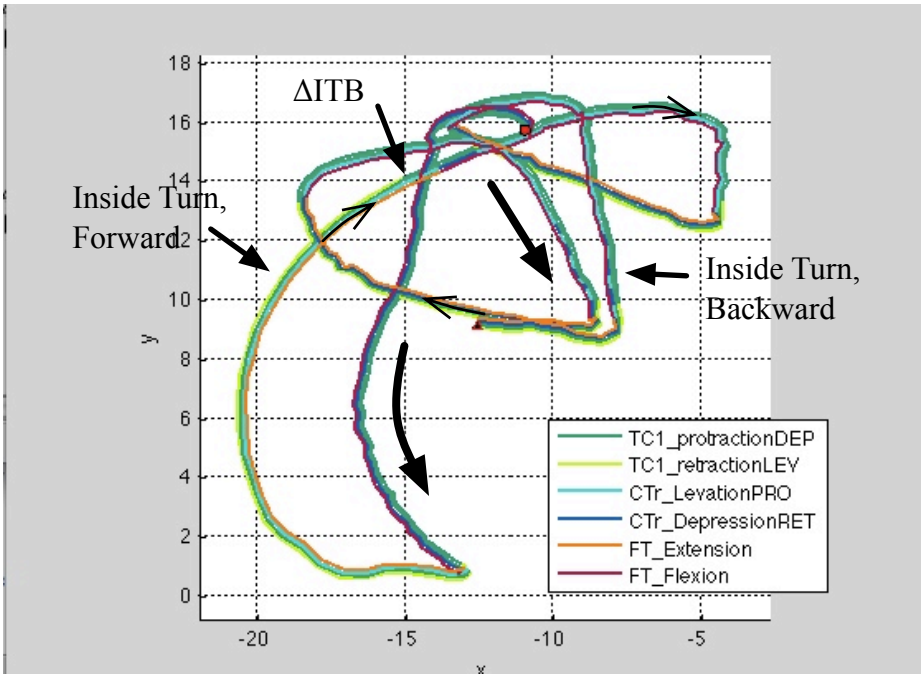


Figure 7-25 A1- ITF to ITB (t=38s-44s). Largest and smallest arrows indicate path direction, labeled arrows point to the stable behaviors and the location of the foot at 40s when ΔITB is applied.

7.4.4.2 B: Combined Changesets

Experiment B (depicted in Figure 7-18) was run three times, with a successful transition from forward stepping to the inside turn-backward behavior in each case. Experimental run B2 is shown in Figure 7-26. This demonstrates that changesets that are subsets of another changeset can be added to reach the final behavior.

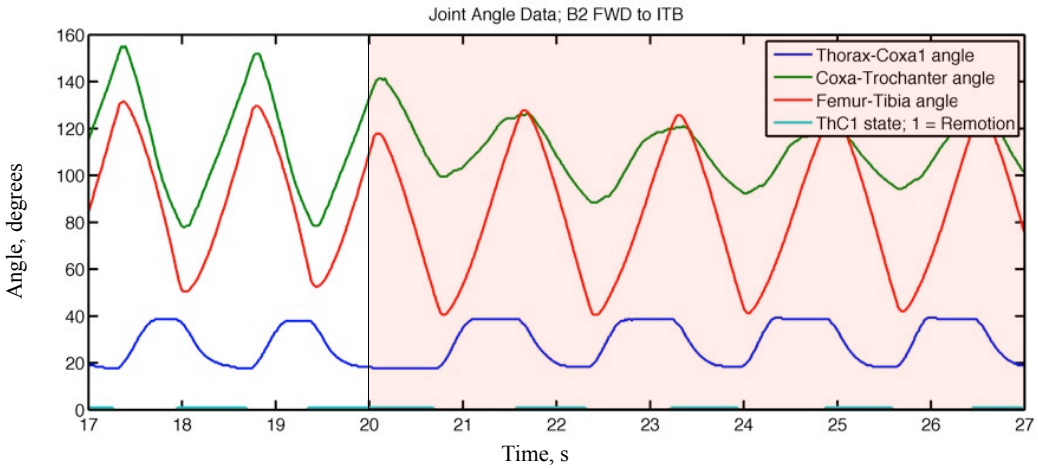


Figure 7-26 Joint angle data for experiment B2: forward to inside turn-backward. The transition is successful; note the change in the range of motion of the CTr angle, and the phase of ThC1with respect to the other joints.

7.4.4.3 D: Reversal of Incremental Changesets

Three repetitions were done of experiment D (depicted in Figure 7-18). In each case the interim state was nonfunctional, supporting hypothesis D1, and the final state was a functional ITB, supporting hypothesis D2, both as shown in Figure 7-27. The external disturbances there were done in an attempt to "kick start" cyclic behavior, a process described in more detail in the conclusion of this chapter. This shows that changesets may produce interim behavior in one order and not in another, but the order does not change the final behavior reached if all the necessary sub-changesets are applied.

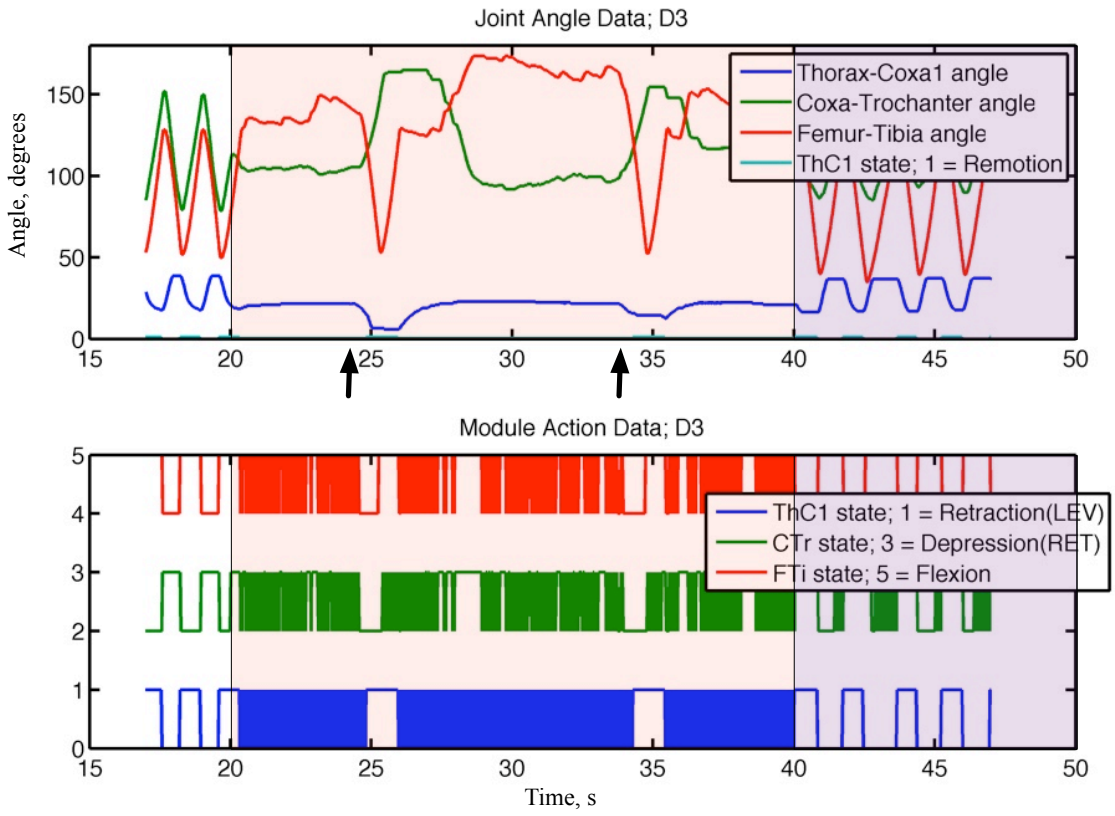


Figure 7-27 Joint angle and state data for experiment D3. Applying ΔITB to the forward behavior (at t=20s) does not produce stepping behavior; arrows point to external disturbances that triggered a still-nonfunctional short sequence of behavior. Note the rapid state switching during this period, probably indicating overlapping sensory event thresholds. Applying ΔITF at 40 s does bring the full behavior to inside turn-backward as hypothesized.

7.4.4.4 F: Some Subsets Don't Work

Three repetitions were done for each F sub-experiment (shown in Figure 7-19). None produced a cyclic interim behavior; this shows that some sub-changesets don't work separately, no matter the order in which they are applied. In Fa, shown in Figure 7-28, a kick-start was necessary to induce cyclic behavior for the final inside turn-forward behavior in one of the three trials. In Fb, shown in Figure 7-29, all three repetitions needed a kick-start to begin ITF behavior after application of the second sub-changeset.

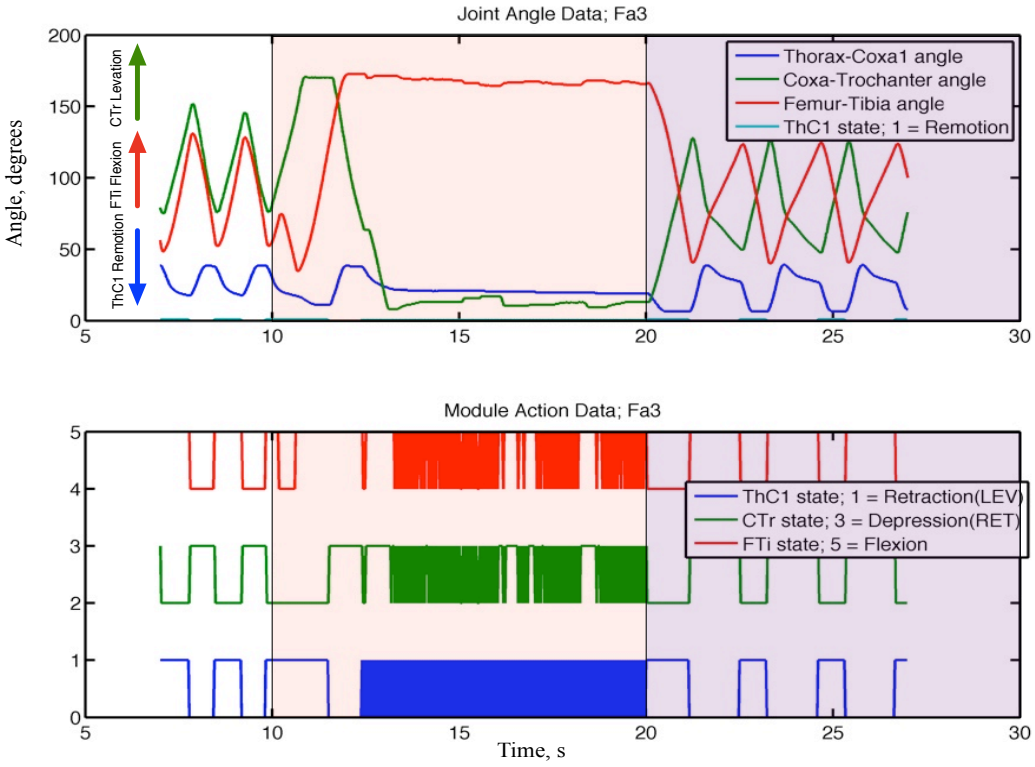


Figure 7-28 Data from experiment Fa3– ΔITF_{FTi} first. Does not produce cyclic behavior during the interim- though all modules experience rapid, useless action switching during this period.

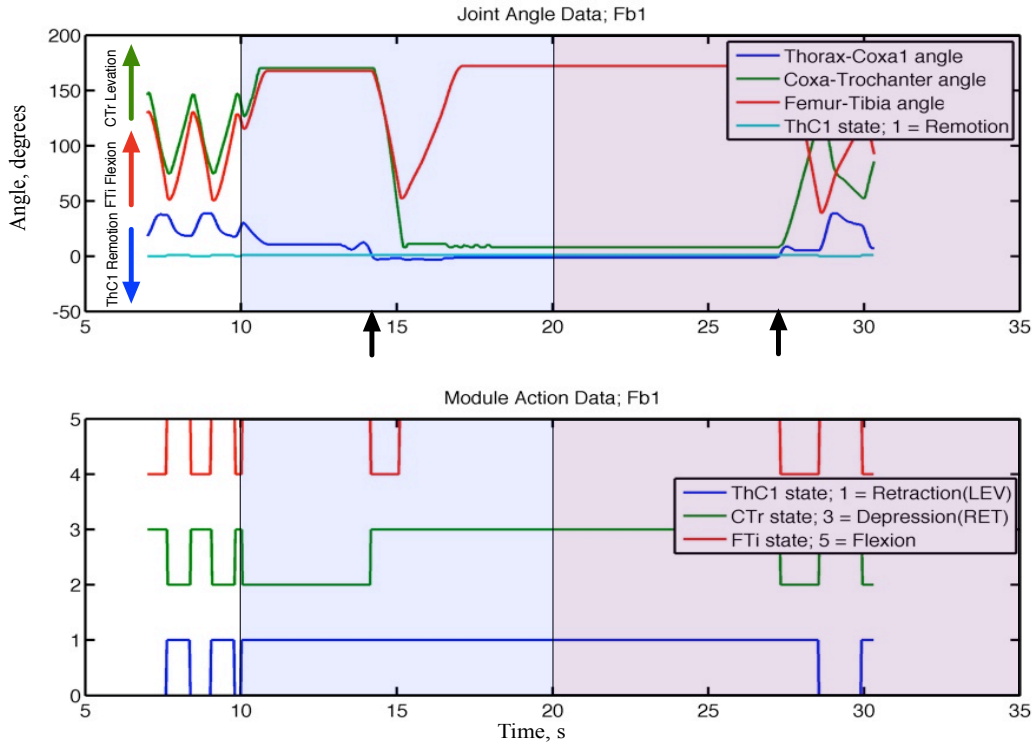


Figure 7-29 Fb1– ΔITF_{ThC1} first. Does not produce cyclic behavior during interim configuration. First arrow- kick-start of interim configuration failed. Second arrow- kick-start of final (ITF) configuration successful.

7.4.4.5 G: Gradual Changes in Transition Thresholds

At least two repetitions were done for each sub-experiment of G, depicted in Figure 7-20. In cases where the FWD behavior had to be kick-started at the beginning, an additional experiment was run. In Ga, shown in Figure 7-30, inappropriate behavior is generated before the gradual adjustment of ΔITF_{thresh} brings the system to ITF. However, one needed a kick-start to begin ITF after having gotten stuck in a different middle-stage failure. One did not start out well due to an unusual initial configuration. The figure shows data from a run that was able to complete transitions without assistance. Experiment Gb, shown in Figure 7-31, needed no kick-starts and generated relatively smooth transitions, though with some apparent instability near the transition rule changes. In experiment Gc, shown in Figure 7-32, one of the repetitions needed a kick-start to

begin FWD stepping at the beginning. In all three repetitions, the oscillatory behavior died out and eventually stopped, but was restored when ΔITF_{trans} was applied.

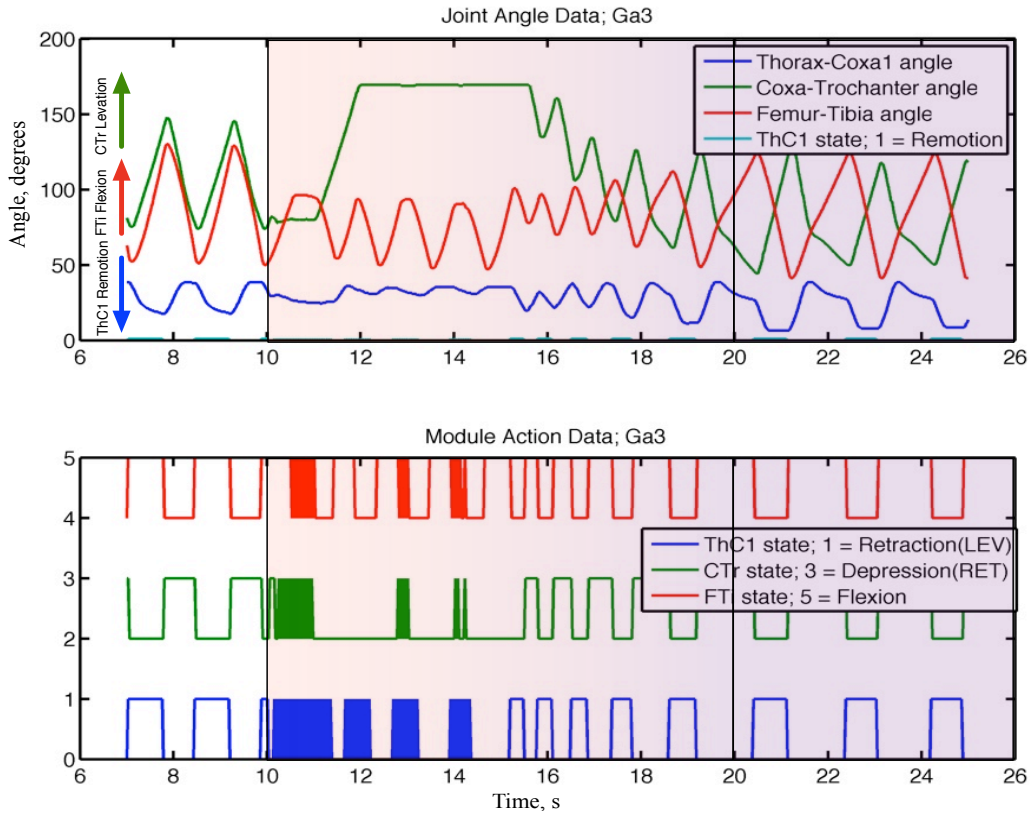


Figure 7-30 Ga3. Transition rule change before threshold change causes inappropriate, uncoordinated behavior; gradually changes to ITF with threshold changes.

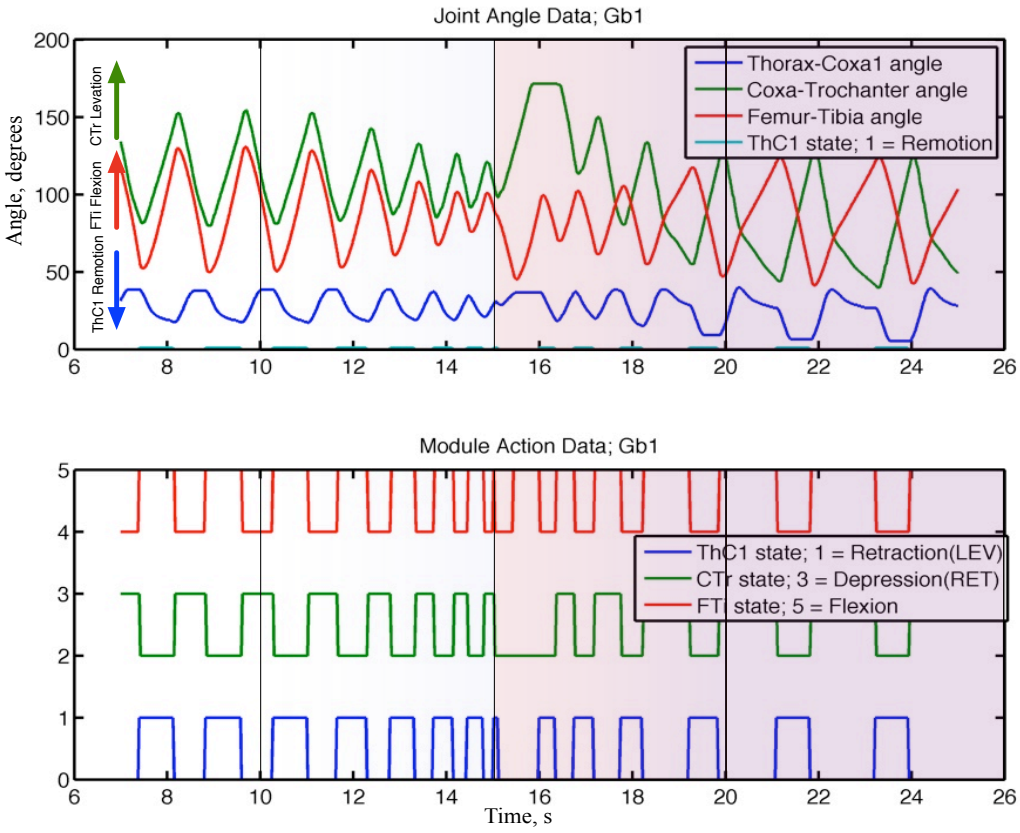


Figure 7-31 Gb1 Transition rule changes in the middle of threshold changes. Two repetitions, both completed transition successfully but with a spastic-looking phase near the transition rule change. This is apparently related to a short period after the application of ΔITF_{trans} at 15 s where the threshold values have not "caught up" with the new arrangement of transition rules.

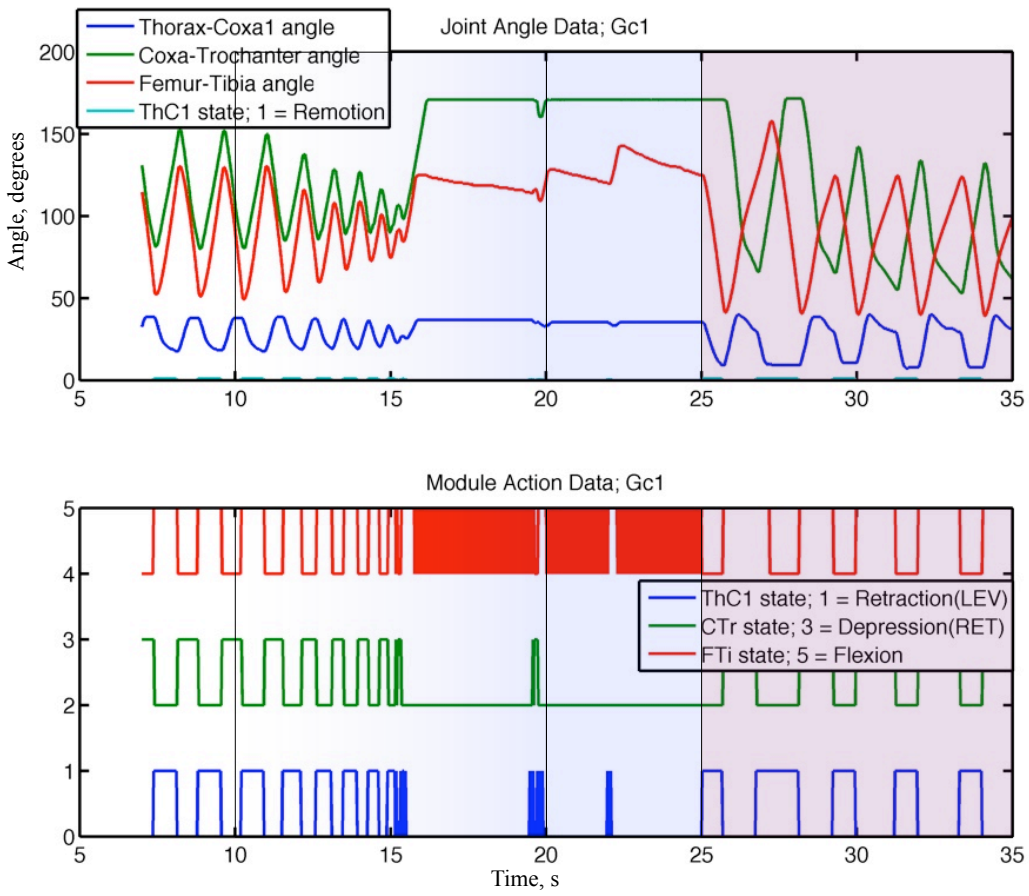


Figure 7-32 Gc1 oscillatory behavior stops after transition thresholds have changed beyond a certain point (about halfway to complete change). Transition rule change at t=25s brings system to ITF. Here, se see that gradual change keeps working slightly past 15 s before behavior breaks down.

7.4.4.1 E: Gradual Changes in Action Dynamics

Each of the E sub-experiments (shown in Figure 7-21) was run twice. In Ea, shown in Figure 7-33, the application of ΔITB_{trans} at 10 seconds shifts to something like backwards walking, then gradually shifts to ITB as ΔITB_{act} is applied. In Eb, shown in Figure 7-34, the behavior is basically ITF until ΔITB_{trans} is applied, then it takes a few backward steps shifting rapidly to ITB. In Ec, shown in Figure 7-35, behavior continues throughout the entire ΔITB_{act} application, but reaches a nearly useless behavioral state before ΔITB_{trans} is finally applied. In all cases the final behavior is ITB.

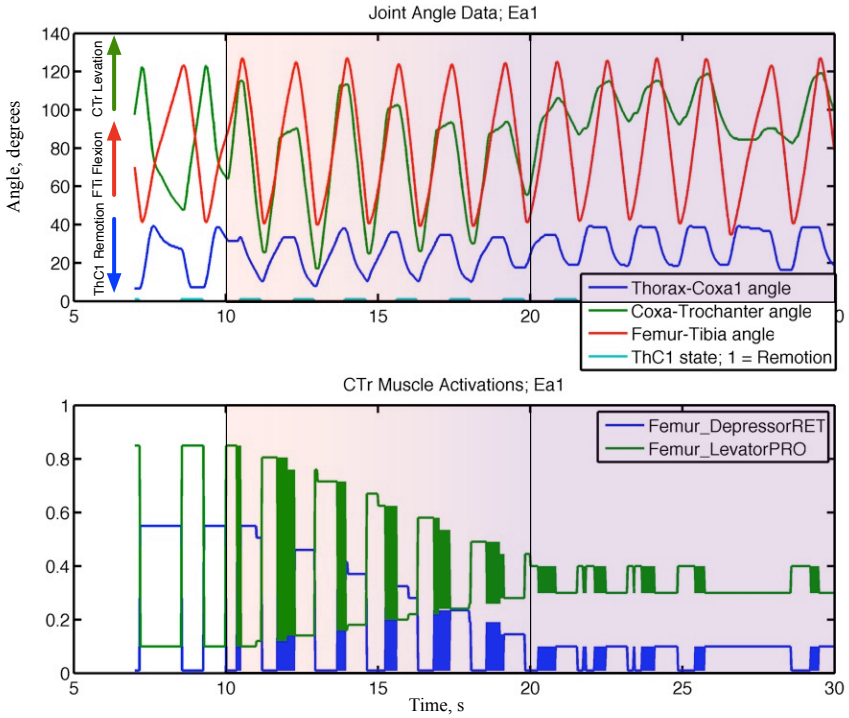


Figure 7-33 Ea1. The behavior switches to backward walking (except the foot doesn't actually leave the ground) when the transition rules are changed at t=10. This slowly changes to inside turn-backward (ITB) behavior as the muscle activations are changed.

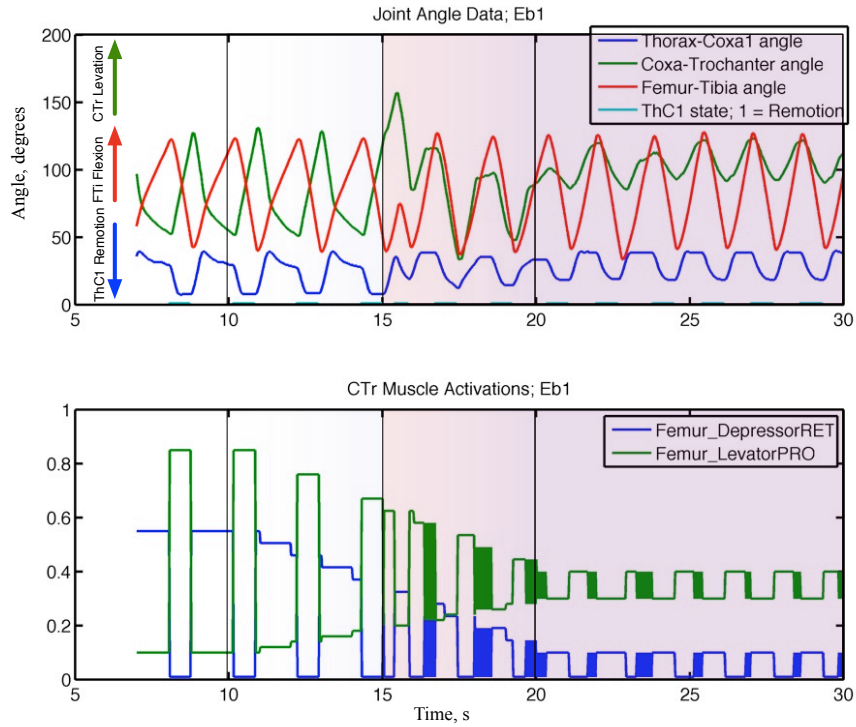


Figure 7-34 Eb1 Behavior continues as ITF until transition rule change at 15s. Then is backward for a few steps before the changing muscle activations bring the system to ITB.

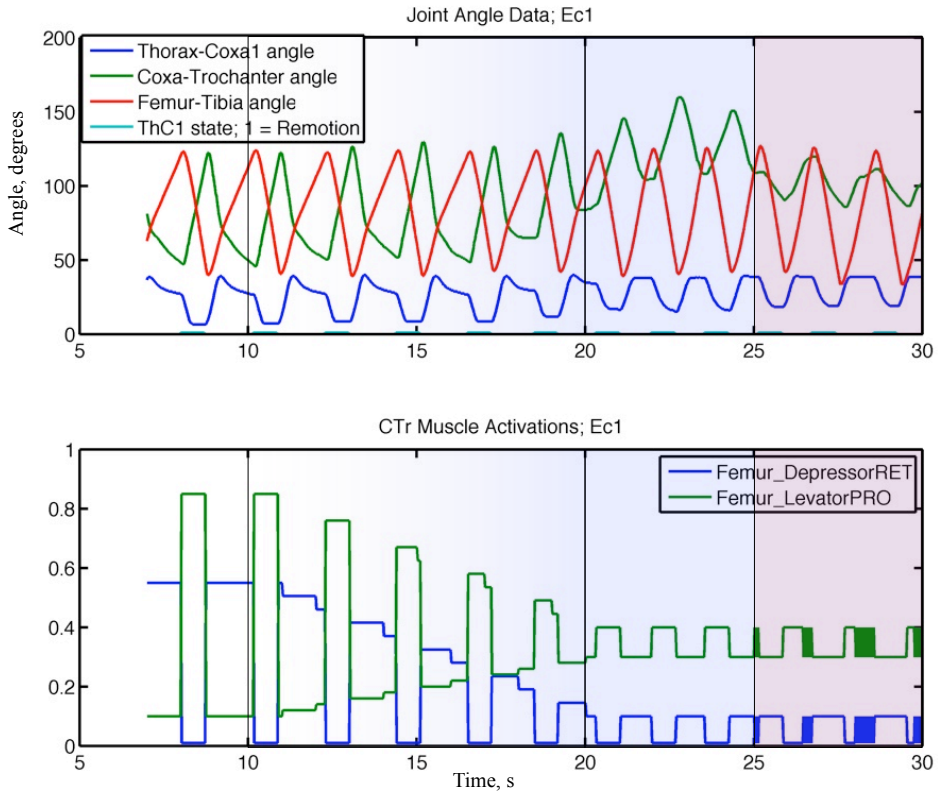


Figure 7-35 Ec1 Behavior continues even past activation changes, but reaches a nearly useless behavioral state (generating sweeping arcs during stance) before the transition parameter change at t=25s brings the system to ITB.

7.4.5 Discussion

7.4.5.1 Summary

Generally, changes between qualitatively different behaviors require a set of changes in control parameters that need to happen in close temporal proximity. Sometimes there are subsets of these change sets that also work; sometimes not. Unsuccessful subsets do not define a functioning post-change behavior; there are of course many configurations that do not define functional behavior. The results above address the hypotheses posed as follows:

Transitions between behaviors can occur smoothly during system operation by changing the necessary parameters all at once (A). Changesets that are subsets of another

changeset can be added to reach the final behavior (B). Such changesets may work in one order and not in another, most likely because the intervening configurations are different (D1), but the order does not change the final behavior reached if all the necessary sub-changesets are applied (D2). Some sub-changesets don't work, no matter the order in which they are applied, and so they must be applied simultaneously or nearly so (F).

The time over which a changeset is applied has an effect on system function during that time, and the coordination of the changes applied usually has a strong effect on system output (E, G). There appears to be a difference between sensory event detection thresholds and action dynamics parameters in this case. Sensory event thresholds may have a narrow range of acceptable coordination with the transition rule changes (G) reflecting the possibility that both of these types of parameters may be associated with the same neural excitation parameters in an animal system (for example, opposing signal strengths as shown in Figure 4-5). The sense of the inequality in the transition rule should probably change when that sensory threshold passes through zero. Action dynamics may have a broader range of acceptable coordination, reflecting the likelihood that these are associated with magnitude influences that are neurally somewhat separate. The interim behavior produced may vary significantly from both pre- and post-change behavior, however (E), suggesting that production of a specific and reasonable-looking change in behavior may still require tight coordination between the neural elements involved in this case.

7.4.5.2 Transient Behavior

In a number of experimental runs a "kick start", some disturbance of system state, was necessary to get a configuration into a cyclic stepping behavior. Though the causes for

this were not analyzed in detail, in most cases it seemed to be due to an ineffective detection of the load/ground contact signal. For example, when starting from some poses the ThC1 module will enter retraction (leg levation) but not generate a sufficient torque sensory signal to send the CTr and FTi joints into their swing states, leaving the robot stuck in the usually-transient "early swing" global state. The ThC1 current signal in the cockroach robot actually has more to do with the transient current spike generated when the ThC1 module changes actions than with the static depression or levation force applied. A weak tonic signal is discernable upon ground contact during protraction (leg depression) but no discernable tonic signal is detected during retraction (leg levation).

It is possible that there are also some problems here with ineffective behavior putting the leg outside the feasible range of poses for a particular functional behavior. In this case, muscle models that are better tuned to condition or limit their joint motion output within the generally feasible range of leg poses could help a great deal. It is also possible that switching influences could be better tuned to deal with these abnormal poses.

When the configuration/neural control is changed suddenly, as in the transitions of experiment A and others, or at system startup when all muscle activations effectively go from zero to the values specified in that behavioral configuration, there is generally a transient relaxation of the system output into the new behavioral limit cycle/attractor. This has largely to do with the response of the action dynamics; the force-based muscle and joint model output may take a few cycles to reach this new stable state. If the action dynamics were instead position-based, the global system output would shift to the new behavior in a faster and more discontinuous manner. Controlled by SCASM, however, such a system *might* still under some conditions relax into the new behavior through

environmental feedback. In a system with the neural-level dynamics being CPG-controlled (and probably in some arrangements of SCASM control), and position-based motion output system, there would be no behavioral transients beyond those required for the position control to settle. It would immediately reach the new prescribed output– whether it was behaviorally useful or not. This reflects the lesser inclusion of the mechanical-level system and environmental dynamics in the generation of behavior in such a system.

7.4.5.3 Suggested Biological Experiments

There are a number of experiments suggested by the hypothetical SCASM configurations used to generate the three stepping behaviors in these experiments. The most straightforward tests of the hypothesized sensory influences described in the event space diagrams in section 7.4.1 deal with manipulation of the leg load/ ground contact sensory events. These might be addressed either by doing substrate drop-out experiments, or by varying the load using magnets as in (Zill, et al., 2004). The reaction to leg unloading during stance of various behaviors could then be compared to the predictions in these event space diagrams; these predictions are indicated by the actions that are promoted by the "leg unload" sensory event:

In FWD, FTi-> FLX and CTr -> LEV. In ITF, FTi -> EXT (possibly only if FTi is flexed enough) and CTr -> LEV. In ITB, FTi -> EXT as in ITF, but CTr -> DEP.

This is described in more detail in section 9.3.1.1 of the discussion.

One caveat here is that the loss of ground contact also induces searching behavior in the animal, which is not really the case in the model at this point. Due to spurious

detection of leg load at ThC1 action switching, the robot may “air walk” for a few steps and appear to be searching, but the event-space diagrams do not predict this, and the joint phasing will remain in the relationship of the behavior for which the model is currently configured. See section 9.3.2.1 of the future work for further discussion on searching. This has some specific implications regarding the reflex cascade hypothesis of (Mu & Ritzmann, 2008b) as discussed below. It is possible that a magnet load/unload experiment might avoid this by involving only load and not foot contact, however it may also be that load changes felt in all legs are ignored for the purposes of intra-leg coordination, in which case a single-leg substrate drop-out or, if possible, single-leg magnet loading would be better. The use of EMGs as well as video kinematics to observe state change at a joint may allow finer discernment of those effects as well, since a load-reflexive state change might be observable in an EMG even if it is quickly overruled by a conceptually higher-level behavioral reconfiguration in reaction to loss of substrate.

The targeting of specific load sensors might also be a way to get at these questions, though the experiments may be more difficult. In particular it might be useful to investigate the roles of the proximal tibial canpaniform sensilla (CS) for tonic load, or “leg load”, and the distal tibial CS for phasic unloading, or “leg unload”, as in the work on *Periplaneta americana* summarized in (Zill, et al., 2004). From a design point of view the phasic unload would be a particularly nice signal to use for the stance-swing transition, but it is also possible that it is used only in combination with other signals, or only for inter-leg coordination of gait, or even something else entirely. A stimulation of the distal tibial CS during stance leading to the predicted swing action in one or more of

the appropriate joints would be strong evidence for the existence of that coordinating influence.

There are additionally a number of experiments suggested by the results of the behavioral transition experiments in the model. Most of the experiments run on the model in section 7.4.4 involve a much finer control over sensory feedback, which would be hard to emulate in the animal. Feasible experiments might involve the enabling, delay, or blocking of the reversal of sensory influences at one or more joints, probably through the manipulation of sensory signals. Ablation of specific fields of campaniform sensilla might be possible, but there is a great deal of redundancy in the load signals available in the leg (as suggested in Figure 4-21 for the stick insect). Manipulating the effects of coordination reversal through stimulation rather than ablation would be less likely to produce equivocal results regarding the function of the related sensory event influence.

Suppose a coordinating load influence between the tibial CS and the CTr pattern generator had been found, and this reverses between forward walking and inside turning. This could be the case if the animal implements the FWD and ITB behaviors above, using tibial CS as a "leg load" sensor. If it were possible to stimulate these CS fields in phase with the swing/stance activity of the leg, then it should be possible to disrupt the transition between these behaviors by stimulating the CS in antiphase of the signal it should be carrying, thereby reversing the reversal. One might in this case get a situation similar to the interim behavior of experiment F, where effectively only a sub-changeset is applied to the coordinating influences.

It might also be possible to use a semi-intact preparation in a manner such as that in (Akay, et al., 2007) to investigate behavior-specific influence reversals at a single joint.

Even if stimulation is done through physical manipulation, when coupled with sensory ablation such experiments might also narrow a sensory event signal down to a specific set of sensory organs.

7.4.5.4 Implications Regarding the Reflex Cascade Hypothesis

These results definitely support the general idea of the reflex cascade hypotheses described in (Mu & Ritzmann, 2008b), in that only a few changes in the local sensory feedback are necessary for some changes in behavior. However, the current model configurations cannot test the specific hypotheses for turning presented there for two reasons:

1) The ThC2 degree of freedom is held constant in these experiments. The tonic postural pose at least, if not phasic activity, of this sub-joint (described in (Mu, 2007; Mu, et al., (in preparation)) is critical to the initiation of the reflex cascade proposed therein. Though it has not been implemented, this might be produced by adding a module for ThC2; if only tonic postural changes are desired it could even be done with only one action. For some further discussion of possible involvement of both ThC2 and TrF in turning, see sections 9.2.2.12 and 9.2.2.13 in the future work.

2) The reflexes which Mu and Ritzmann identify and require for the reflex cascade are magnitude influences (as depicted in Figure 3-3), rather than coordination influences. The NeuRoMod software is capable of implementing such magnitude influences as muscle activation functions that are modified by sensory input. All muscle activations in the current cockroach model configurations are constant, however, and incapable of responding to the changed load signals (generated by ThC rotation) in the way required to change behavior.

The work in (Mu & Ritzmann, 2008b) suggests that if the appropriate magnitude influences were added to the model, it might also automatically generate searching behavior upon loss of ground contact, which it is currently incapable of doing as described above in section 7.4.5.3 .

8 Conclusion

8.1 Summary

This project is part of a larger effort, in the biorobotics group at Case Western Reserve and elsewhere, to understand and apply the methods used by animals in locomotion. The specific question investigated here is how the interaction of descending commands with local sensorimotor control networks result in the smooth behavioral changes seen in agile legged locomotion.

My contributions resulting from this work are summarized as follows: Robotic models were developed to both test and develop hypotheses about the effects of descending commands on local control of stepping. Methods for controlling and modifying these models interactively were developed, as were methods for precise system description and data handling. Experiments establishing model functionality and addressing initial questions about descending modification of local behavior were conducted. These results support the general idea of the reflex cascade hypotheses described in (Mu & Ritzmann, 2008b). Hypotheses about the biological system were extracted from the experimental process, and several of these may be feasibly testable in the cockroach. A functional kinematic model of the cockroach middle leg was developed, with particularly new insights regarding the operation of the thorax-coxa joint. In addition to its applications in modeling, the organization of local control observed in the stick insect was standardized as Sensory Coupled Action Switching Modules and presented in a way that could be useful in many control problems.

8.2 Model Development

The necessary elements of model usage and construction were developed sufficiently well to make an initial attempt at the reverse-pathway modeling flow shown in Figure 2-2, generating testable biological hypotheses. These hypotheses are presented along with the associated experimental work in chapter 7. The neurally-based control of (Ekeberg, et al., 2004) was used to generate various stepping behaviors in physical model legs, which can be experimentally manipulated both physically and through the modification of control parameters, as was desired.

Due to the neuromechanical nature of step generation in these models, some basic but critical modeling of the physics of the target system was done. A crude feed-forward torque control method was developed for the application of muscle model output to position-controlled servomotors. Without this ability for torque control of some sort, the models would have a significantly different behavioral output and a significantly worse structural match with the target systems. Another fairly simple but critical element of the physical modeling was the kinematic match of the robot legs to the target systems. In the stick insect leg this was a fairly straightforward matter of consulting the literature. It uncovered a lack of suitably complete data in the cockroach literature, however, which prompted the development of initial cockroach kinematics models from experimental data (presented in chapter 5), which is already being used here and in (Mu, 2007), (Bender, et al., 2009 (In Prep)), and is still a matter of further investigation.

In order to support the use of those models by biologists, both in interactive behavioral modification and in the execution of repeatable scripted experiments, the robot control engine was modified to allow online modifications of many model parameters, and

development of a model description language, SCCILS, was begun. Various control programs now make use of these capabilities, including the interactive graphical user interface developed by John Bender as part of the NeuRoMod software package described in chapter 6 and in Appendix 10.4.

8.3 Experiments

Throughout model development, various experiments were run. These ranged from initial testing and demonstration of SCASM-generated stepping in robots, to the development and refinement of specific hypotheses regarding the effects of descending influences on local control in the cockroach.

8.3.1 Initial Testing

Initial testing in the stick insect model revealed a resilience to, or automatic compensation for, some kinds of environmental variability. This included changes in substrate height, which is a primary element of rugged terrain. The control system was not obviously vulnerable to the initial state of the joint control modules, and started behavior well over a range of initial poses, including at least those contained in the currently configured behavior. Limits to feasible operation were apparent, however, and the sensitivity of the system to a problem with the detection of ground contact and load was clear.

A very clear demonstration of the neuromechanical nature of the behavior of these systems was discovered: the temporary physical disconnection of a portion of the leg, while maintaining all neural connectives. This procedure may be impossible in the animal. The usual cyclic behavior of the joint is severely disrupted during the

disconnection, but resumes when the leg is reconnected. This also demonstrates the resilience of the SCASM controller to temporary disturbances. The utility of a physical model is also highlighted here in two ways. Firstly, this demonstration is far more clear in the physical world than it would be in simulation. Secondly, this was discovered serendipitously during model operation, which is not something that would have happened in a computational simulation.

8.3.2 Muscle Model Experiments

In order to address the problems in the initial stick insect model ground contact/load detection and the transfer from swing to stance, two minimal muscle models were applied at the femur-tibia joint. These did improve the swing –stance transition, and significantly conditioned overall system behavior in ways not immediately obvious. These results suggest that simple muscle models might be sufficient for some useful level of model behavioral match; this is a further example of the low computational complexity necessary for SCASM control. It is also apparent that modification at a single module can have significant effect on overall system behavior, and concentrating on the improvement of a module or action involved in a critical phase of the cyclic behavior may be an effective strategy in SCASM design improvement.

The muscle model results also suggest that the state transition modules of SCASM behave better when driving an underlying plant which exhibits saturation, in order to ensure stable ongoing generation of the desired repetitive motion sequence. This does increase the overall complexity of the control system a little, but the simplicity of the "muscle models" used here demonstrates that it need not be much.

8.3.3 Experiments in Descending Control

The first demonstration of something analogous to descending control modifying local behavior was the modification of leg stepping speed. This was achieved simply through the modification of tonic muscle activation signals, effectively gated by the joint action switching modules/pattern generators. The biological hypotheses associated with this behavior in the model are difficult to test in the animal, and not directly associated with previous work in the Ritzmann lab. Therefore, this behavior was not fully characterized or investigated.

Transitions between forward walking and two types of inside turning in the cockroach leg model were examined in some detail. Generally, changes between qualitatively different behaviors such as forward walking and forward-type inside turning require a set of changes in control parameters that need to happen in close temporal proximity. The order in which these changes are applied has an effect on the interim behaviors, but the final resulting behavior will be the same. In some cases there are parameters that may be changed gradually between behaviors. If these parameters are sensory event detection thresholds, they need to be timed tightly with the changes in transition rules, reflecting the possibility that both of these types of parameters may be associated with the same neural excitation parameters in an animal system. If the gradually changing parameters are associated with action dynamics there may be a broader range of acceptable timing of the transition rule changes, but significantly different interim behaviors may be produced.

Observations may be made regarding the transient behavior of the system during transition between behaviors. In a number of experiments a “kick start”, some disturbance of system state, was necessary to get a configuration into a cyclic stepping

behavior. In most cases this seemed to be due to an ineffective detection of the load/ground contact signal. When the configuration is changed suddenly there is generally a transient relaxation of the system output into the new stable cyclical behavior. This has largely to do with the force-based dynamics of module actions, but is also a function of the nature of sensory feedback in SCASM control. In a CPG-controlled position-based system there would be no behavioral transients beyond those required for the position control to settle, and it would immediately reach the new prescribed output– whether it was behaviorally useful or not. This reflects the lesser inclusion of the system mechanics and environmental dynamics in the generation of behavior in such a system.

There are a number of biological experiments suggested by this experimental work. The most straightforward tests of the hypothesized sensory influences for the various cockroach model behaviors deal with manipulation of the leg load/ ground contact sensory events. These might be addressed either by doing substrate drop-out experiments, or by varying the load using magnets as in (Zill, et al., 2004). One caveat here is that the loss of ground contact from whole-body substrate drop-out experiments induces searching behavior in the animal, which is not the case in the model at this point. The targeting of specific load sensors might also be a way to get at these questions, though the experiments may be more difficult. In particular it might be useful to investigate the roles of the tibial campaniform sensilla (CS) fields for detection of leg load and unload signals, as in the work on *Periplaneta americana* summarized in (Zill, et al., 2004). A stimulation of the distal tibial CS during stance leading to the predicted swing action in one or more of the appropriate joints would be strong evidence for the existence of that coordinating influence.

There are additionally a number of experiments suggested by the results of the behavioral transition experiments in the model. Experiments which might be feasible would involve the enabling, delay, or blocking of the reversal of sensory influences at one or more joints, probably through the manipulation of sensory signals. Manipulating the effects of coordination reversal through stimulation rather than ablation would be likely to produce less equivocal results regarding the function of the related sensory event influence.

These results definitely support the general idea of the reflex cascade hypotheses described in (Mu & Ritzmann, 2008b), in that only a few changes in the local sensory feedback are necessary for some changes in behavior. However, the current model configurations cannot test the specific hypotheses for turning presented there because 1) the ThC2 degree of freedom is held constant in these experiments, and 2) the reflexes which Mu and Ritzmann identify and require for the reflex cascade are magnitude influences, rather than explicit coordination influences. Although it is possible to configure such influences using NeuRoMod, all muscle activations in the current cockroach model configurations are constant, and incapable of responding to changing load in the way required by the specific inside turning reflex cascade hypothesis. The work in (Mu & Ritzmann, 2008b) suggests that if the proper magnitude reflexes were added to the model, it might also automatically generate searching behavior upon loss of ground contact, as is observed in the animal.

8.4 Initial work on description and standardization of SCASM

The control organization that is apparently at least partially responsible for the generation of stepping in stick insect legs, as described by (Ekeberg, et al., 2004), has

been presented in chapter 4 as Sensory Coupled Action Switching Modules (SCASM). A number of representation and description tools have been developed to implement the "elements of model usage" described in section 2.1, as they pertain to the general description of hypothetical mechanism that SCASM provides. The informational content of event-space diagrams, global state diagrams and sensory flow diagrams have all been presented. The usage of these diagrams in describing system behavior, and in both the forward and reverse model usage pathways – hypothesis *testing* and hypothesis *generation* – have been demonstrated. Additionally, the steps necessary to construct a SCASM controller for a system from scratch have been identified and presented though the example of the minimal SCASM stepper.

A comparison of the SCASM description with finite state machines and subsumption architecture is presented. In some cases, SCASM is just one way of representing a collection of state machines, with specific advantages including representation of biological systems. SCASM may also be useful in the description and implementation of systems with smoothly changing module states, however.

8.4.1 SCCILS and the SCCILStoolbox

The SCASM Command and Configuration Interface Language Specification (SCCILS) is an XML specification that formally defines a language for defining and controlling SCASM-controlled systems. SCCILS makes use of the "ports" for system modification in the SCASM representation of control by developing an explicit representation language. This has supported the development of a user interface, representational diagrams, and data handling and display code which should be relatively easy to extend to deal with a broader class of SCASM-controlled systems.

8.4.2 General Applicability and Other Advantages

The concept of Sensory Coupled Action Switching Modules need not be limited to the control of legged systems. This representation can be used to describe any system with sub-elements that perform a set of actions, in which the sensory coupling between these elements can override central coupling which might exist. It is possible that it will be useful to use SCASM to describe *or design* any multi-dimensional control system that needs to go through a coordinated set of states in concert with the environment.

The computational simplicity with which a SCASM control system may be implemented is impressive when compared to explicit inverse kinematics or coupled dynamical oscillators. This allows for more high-level processor availability in computer control, or implementation on low-power, distributed control hardware. This in turn should allow a robustness of legged locomotion control which has not yet been reached in small autonomous vehicles.

The control of each joint, or more generally each action switching module, is also conceptually quite simple, allowing for straightforward implementation. Linking several of these modules together, as in the joints of a leg, can nevertheless generate a wide range of relatively complex behaviors. The emergence of these behaviors may not be immediately obvious, and the prediction of this emergence can be helped by the application of tools presented in chapter 4. The application of similar controllers to mechanically different robot legs in this study and, in (Ekeberg, et al., 2004), to models of different legs on the stick insect, demonstrates that one basic set of action switching modules and sensory events can, with modification to the switching event connections

and other control parameters, address an entire class of control problems. In this work, that class is the generation of repetitive stepping motions by a 3-DOF linkage.

The sensory coupling of SCASM can give rise to automatic adaptation of the behavior to changes in the environment, as is demonstrated in the substrate height changes described in section 7.1.1. Other methods of control such as inverse kinematics can require an explicit handling of such changes at a higher level. The dynamics of the underlying plant are also a more integral part of overall system behavior in SCASM. With appropriate system mechanics, this could lead to even more inherent adaptability in low-level control.

In the robotic models used here, not only is the inverse-kinematic calculation not required due to SCASM control, but large amounts of flexibility in the joints and limb segments also mean that the foot position cannot be accurately derived using forward kinematics. This information is not necessary, however, for SCASM to function. The leg control only explicitly depends on the interaction of the leg with the environment; kinematic information is used in control through joint angle sensory events, but this contains only an approximate implicit representation of foot position. This is apparently all that is needed for these behaviors.

The controllers of local step generation presented here do not directly address higher-level problems of legged vehicle control such as gait generation, posture control, and navigation. However, the experiments on speed control and transitions between walking and inside turning in the cockroach model demonstrate that the behavior of a SCASM system can be usefully influenced online by modifying action switching parameters and action dynamics. It is believed that such modification, whether from local circuits for

inter-leg coordination, or from a higher level "brain" for navigation-related control or complex obstacle traversal, can successfully broaden the behavioral repertoire necessary to get the leg to do whatever is necessary. Indeed, since a particular set of modules and switching events can solve an entire class of control problems, descending modification of the system should allow online adaptation to some range of changes in the physical plant as well.

The SCCILS language and the control programs that comprise NeuRoMod were developed from and around legged systems, and are currently still limited to these. They were built with generalization in mind, however, and should require relatively little modification to describe and control a much broader range of systems. It is possible that they will be useful in applying SCASM to describe, control or design any multi-dimensional control system that needs to go through a coordinated set of states in concert with the environment. As mentioned in the future work section, this work has already begun.

9 Future Work

As you should be able to tell from the introduction, the work presented in this dissertation is part of a broader inquiry into the basic problems of legged locomotion. I find all this stuff captivatingly interesting, and consistently work to keep broader applications and investigations in mind. This is part of the reason this section is so long. This is one example of a few good answers to an interesting question conspiring to create an explosion of new interesting questions. The systems involved are complicated, and this makes basic observation, experimentation, and modeling all take considerable time and effort. Even though this and contemporaneous work are making exciting progress in addressing long-standing problems in robotics and biology, this line of investigation is really rather new. There is a lot of basic down-and-dirty "digging it up" work to be done, and a lot of valuable knowledge to be found there. It is my hope that this section will serve as part guide and part inspiration for further work on and using of SCASM, SCILLS, and the models presented here.

The rest of this section consists of a short guide to help one choose what to work on, or what approach to take, followed by seven primary directions of inquiry. Each direction is of varying complexity itself, and some of these already contain enough questions for many theses' worth of effort. They are, essentially:

1) Biological model development: The improvement of controller, physical robot, and SCASM concepts as models of biological systems.

2) Model-driven & Model experimentation: experiments to be done using the robotic models, and experiments to be done on biological systems– resulting from development and experimentation on the models.

3) Central Pattern Generators: system development to incorporate central pattern generation, i.e. internal module dynamics, into SCASM systems. Also, experiments and hypotheses to address with these tools.

4) UI and Data Representation: Further tools for interaction of the systems and models with human experimenters. Though little of this is strictly "biology" or "engineering", there is a lot of high-value work to be done here.

5) Theoretic understanding and development: Regarding the formalization of SCASM and related concepts, and the relation of these things to prior work in finite state and hybrid systems.

6) SCCILS generalization: expanding the SCCILS XML language to more readily address the gamut of control systems that might be usefully represented in this manner.

7) SCASM in other systems: regarding the use of SCASM to design and/or implement control in systems other than intra-leg coordination.

9.1 Consultation and Choosing What to Do

It is my hope that this section will be useful specifically to investigators interested in choosing a research project that works for them. Since addressing all the questions in the rest of this chapter would probably take at least 20 person-years of hard work, and

possibly more like 100, it is highly advisable to spend some effort biting off a bit that you might actually be able to chew.

For an engineer or any other skilled service provider, one can define three levels of success at fulfilling the client's demands. These concepts may also be useful more broadly in planning, consulting, design and technical communication.

At the first level of success, you give the client what he told you he wants. This level of success is often difficult enough in itself. The desired item may be difficult to produce, and it requires a minimum of understanding technical demands. To a skilled practitioner, however, this level will rarely be satisfactory for long. This leads to the second level:

Provide the client with what they actually want, which may or may not be very close to what they told you they wanted. It doesn't take too much experience in any applied technical field to see the difference between these two levels of success. Often enough, providing only the first level creates a client who is satisfied only temporarily. They will realize before too long that what they got isn't actually what they wanted, and they'll try asking for something else which may actually be closer to what they want. If things go particularly poorly, the client may even blame the practitioner for this failure. If the practitioner sufficiently understands the client's desires and language, it is often possible to skip the interim steps and through more exact communication determine what the client *actually* wants, then work towards this solution from the outset.

At the third, considerably more difficult and substantially more valuable level: you provide the client with what they actually *need*. There are several things that make this difficult to accomplish. Firstly, one must sufficiently understand the client's surroundings, goals and limitations to have a chance at discerning actual needs that

pertain to the product or service you are providing. Secondly, having convinced yourself that a particular solution is what the client actually needs, there remains the possibly excruciating task of convincing the client that this is in fact what they need.

In choosing a project, I suggest using the methods involved in providing as high a level of successful consulting as you can, in serving yourself as the client asking the question of what to work on. What are the underlying or big picture questions you are actually addressing, or that you want to address? What are your limitations of time and effort? How would one best address those questions? How do you stay sane in so doing? It may be a useful exercise to express your answers to each of these questions at each level of success. Try to provide yourself with what you actually need.

That bit about staying sane is a crucial but often overlooked part of project planning, particularly for small or single-person projects. In my opinion, it is worthwhile to include sanity and happiness in your plans in any case. Even if you ignore this; sick, tired, annoyed, overworked, and overstressed people just plain don't get as much done as people who maintain some rest and peace of mind.

It should also be noted that these three levels of successful service apply directly to interactions between biologists and engineers, from both directions. When collaborating with colleagues from another discipline, one should work very hard to reach at least the second level of understanding and response. It will speed things up a great deal, and save considerable gnashing of the teeth. This should involve at least some amount of getting your hands dirty in your colleagues' lab(s). It takes work, and it is worth it.

Of course, even if you have a pretty good idea of what you need, there is no guarantee that you will actually be able to obtain such. Ideally, one should take account of this in project selection and planning.

Back in March of 2005 while I was in Köln at the Büschges lab, Roy Ritzmann sent us an email about how great it would be to implement the simulation of (Ekeberg, et al., 2004) in a robot for modeling purposes. I'd already been thinking about that (and so had Bill Lewinger, it turns out), but thought the requirements for modeling would be pretty difficult to meet.

> Roy: "[how about we make the most awesome robotic model of insect walking ever, with electric motors]"
>
> Me (reply to all, after some serious thought and planning): "[sounds great! I'll need 3 MS students, 2 more Ph.D. students, and five years. I don't think electric motors will work]"
>
> Roy (probably stifling guffaws, or maybe feeling an ulcer): "[no, just a minimal prototype. I'll get you a couple of undergrads for the summer. It has to use electric motors]"
>
> Me (still replying to all): "[well, I'll see what I can do, but I'm not so sure. Also, I really don't think you understand what you're asking with the electric motors. The dynamics are all wrong, and it probably just won't work at all that way]"
>
> Roy (hands around my neck, upon my return to campus): "…Let me explain something to you."

9.2 Biological Model Development

This section covers things that could be done in the model, in the physical robots or control programs, to improve its function as a biological model. Many of these items could be expected to provide better function in a robot using SCASM for locomotion control, but the primary goal of the discussion here is the improvement of the modeling ability of the system.

9.2.1 Improvements for Experimentation

There are a great deal of improvements that could be made to NeuRoMod to improve the flexibility and control of experimentation in the model. Some of these have to do with basic representation and interaction with the controller; those are covered in the later section 9.5 on UI and Data Representation. Here, there are improvements that could be made specifically to allow more flexible and biologically relevant experimentation.

9.2.1.1 Sensor-Based Scripting

Imagine that you could modify neural circuits not only at specifically scripted times, but also based on a condition in the system; e.g. “ground contact” or “early swing”. This could be implemented in NeuRoMod or a similar system as “Boolean scripting”, where rather than setting a time to implement a change in parameters, the scripting process would monitor sensory signals and wait to implement a change until a set of conditions was met. One could even combine time and Boolean scripting, making time one of the signals to watch to determine whether to implement or not, and possibly adding another time to wait *after* the conditions were met before implementing the change. There is a range of experiments that could be done using these methods that is currently not possible in the model, and possibly very difficult to do in the animal.

Script elements such as these could be used to investigate whether the sensitivity of the system to certain parameter changes is dependent on where in the step cycle the changes take place. For instance, is it better to change from walking to turning during late swing or late stance? This scripting might also be used as an ad-hoc addition of further model circuit elements. A Boolean script element essentially creates a new virtual neural

element whose activation requirements and effects are quickly configurable. The conceptual mapping to biological systems here is not guaranteed to be clear, however.

9.2.1.2 Sensor Signal Modification

Also on the theme of messing with the system signals during operation, it may be useful to change how sensory signals are processed on the fly, rather than simply allowing disconnection/reconnection/threshold changes as is currently the case with scripting. If the sensory signal processing additions mentioned in 9.2.2.8 are made SCCILS-configurable, this capability will result.

It is possible to effectively turn a sensor on/off/up/down using current scripting tools, but this would require simultaneous changes in thresholds or gain parameters wherever the sensor value is used, including transition rules, muscle activation functions, and so on. This is not easy to keep track of. A possibly useful interim step would be the ability to turn the actual "sensing" of a sensor on and off, combined with the ability to set the value for a disabled sensor. One should be careful, however, to insure that physical sensory variables used in the simulation of physics, e.g. in muscle model calculations, are not modified by changes intended to model neural modulation or disconnection.

9.2.2 Improvements in Model-Animal Parity and Model-Model Equivalence

In theory, "improvements in model-animal parity" could take place both in the construction and use of the model, as well as in modification of animal handling or experimental apparatus and protocols. Although the thoughts shared here are primarily aimed at modifying the model, *the possibility of modifying animal experimentation for this purpose should neither be ignored nor forgotten!*

"Model-model equivalence" here refers to the ability to get the same results from running an experimental script on different models, here taken to mean the "engine" as described in SCCILS. This is particularly tricky when trying to compare, say, a robotic engine to a purely simulated engine. Given the extent to which certain elements of NeuRoMod are open-loop control, however, there is considerable likelihood that the stick insect robot at Case Western and the supposedly identical robot at the University of Cologne will not only give different results for the same script, but may require somewhat different sensory event thresholds or muscle model parameters to get even moderately similar stepping behavior.

9.2.2.1 Quantification and Improvement of Behavioral Match

So far, the reproduction of behavior by the model has been analyzed only at the qualitative level. "Walking" and "turning" are defined by the gross coordination of joint movement and foot direction during stance. There is, however, considerable quantitative animal data on leg kinematics. It would be very valuable to quantitatively compare the motion of the model to that in the animal, both in the form of comparisons of joint angle data and in comparisons of foot position data. Although ideally comparison of these two types of data would be equivalent, the associated implicit assumption of perfect model kinematics should not be made.

Once a method of quantitatively comparing motion is chosen (mean squared error would probably be a good start), it should be possible to improve the behavioral match between systems. In the first stage this could probably be done by interactive parameter tuning; in a second stage it may be useful to implement a SCCILS "interface" which runs hardware-in-the-loop optimization on the SCASM control parameters. Note that this can

be an extremely high-dimensional parameter space, so it would be wise to spend some time choosing a fairly efficient optimization scheme. A matter of some technical interest would be developing heuristics regarding which parameters are most likely to be involved in the improvement of matches in which of the output variables (i.e., thresholds for "ThC Angle Retracted" are probably important for fixing mis-matches of foot x position at the end of stance).

9.2.2.2 Torque Control Calibration

A basic calibration of the torque control method described in Section 6.2.1.1 would substantially improve capability for both model-animal parity and model-model equivalence. Currently, the Δ_{servo} command is the output of the muscle modeling and input to the motor torque control. This signal, however, has no inherent or obvious quantitative physical meaning. Higher numbers mean generally higher torque, but other than that it is not clear. Not only do we not know the physical magnitude of these torques, but it is unlikely that there is actually a linear relationship between Δ_{servo} and joint output torque, which is currently implicitly assumed.

It would be both more generally applicable and more meaningful if the outputs of the joint calculations was actually torque. Another, more engine-specific part of the program would then apply a calibration between this torque command and low-level actuator commands.

9.2.2.3 Torque-Velocity Characterization

It should be noted that the above leaves open the question of what variables are included in the torque calibration. A static measurement of motor torque over a range of

Δ_{servo} values would be a big improvement over the current situation. This might not, however, take into account the inherent torque-velocity relationship of the motors (or more generally, most actuators). This is a substantial part of the dynamics of the robotic models described in Chapter 6. A sensible force-velocity relationship can go a long way towards smoothing and stabilizing the behavioral output, and it is likely that without the natural torque-velocity characteristics of the actuators it would have been necessary to actually configure force-velocity characteristics for the model muscles. As for the basic torque calibration, understanding and taking account of these dynamics would be a considerable help in allowing comparison of behavior between model and animal, and between models.

Note that there are a lot of data necessary for this calibration; it involves torque measurements at various angular velocities for a range of Δ_{servo} values. For implementation, it would be advisable to use a fast look-up table, such as the TableLookup function in the utility.c file for the RUI control software (used for Robot V and Puppy).

9.2.2.4 Gravity Compensation

Insects operate on a different dynamic scale than humans; damping and friction are vastly more important in their world than in ours, and inertia and particularly gravity vastly less so. A trained mechanical engineer should have some feel for these scaling effects; if you would like some convincing it might be good to start with (Hooper, et al., 2009).

Unlike the insects and computational models, currently implemented robotic models are an order of magnitude or so larger than the insect, and relatively much more affected by gravity. Given the pose and mass distribution in a robotic leg, it is possible to calculate the resultant gravitational torque at each joint (see, e.g., methods inAsada & Slotine, 1986). With sufficiently high-fidelity torque and torque-velocity characterizations, it would then be possible to add the opposing torques, thereby largely removing gravity from the robot dynamics as is the case in the insects. Of course, gravity does still exist at the smaller scale, and it may eventually be sensible to take more carful account of just how much gravity should be "removed". It is likely, however, that taking it away as completely as possible will be an excellent first approximation.

9.2.2.5 Automatic Dynamic Scaling

This one is probably pretty low-priority, but particularly when dealing with different models of the same system, which may operate at different sizes, it would make some sense to make the dynamic scaling of the system automatic. This would automatically scale the script times and any delays or update rates appropriately for the size of the model, and provide model time scaled back to the target (e.g. insect) time scale with any recorded data. It would make sense also for on-line UI signal displays to display either real or "scaled model" time on graphs, for sensory transmission delays, and other time-based configuration parameters.

9.2.2.6 Body Motion

The current robotic models are attached at the thorax-coxa joint at a fixed location and orientation with respect to the ground. In freely walking insects, of course, the body both rotates and translates during walking. Many of the stick insect experiments upon which

the control circuitry in these models was based did take place with the body fixed with respect to ground; however since the cockroach model in particular is now being used in parallel with flexibly tethered or freely walking animals, it may be important to take account of the accompanying body motion.

Initially, the plan was to allow at least some dynamic support of simulated body weight in the robotic models. A vertical linear bearing, with the weight of the leg supported by springs, might allow at least some vertical dynamic loading similar to the take-up and release of body load during stance. This turned out to be a pretty serious pain in the neck, however. Since the downward force of the foot is applied at some distance from the body, there is significant torque applied at the thorax-coxa joint. This torque on the joint requires reactive torque in the slider or linear bearing element attached to the thorax side of the joint, which will bind and render immobile all but perhaps the fanciest linear bearings. For some “fun”, you might ask Bryan Taylor about this, since he wrestled with this problem for several weeks at the beginning of his involvement with the project.

The suggested, more workable method would be to use a four-bar linkage to allow body-like motion. One might even incorporate measured body rotation into the design of the mechanism, though a first shot approximating vertical-only motion would likely be an improvement over the current design.

9.2.2.7 Improved Physical Sensors

Since sensory information is so critical to the operation of a SCASM system, good sensory signals contribute substantially to good system behavior. Some deficiencies can be treated with sensory processing, but the best way to improve sensory signals is to have better, and perhaps more, sensors.

Force in the robotic models is currently represented using motor current signals. These are low-resolution, noisy, and subject to systemic errors introduced by the inertia of the motor, transmission, and leg itself; none of which are likely to be important in the insect. Actual force or strain sensors and contact sensors would provide much more reliable and structurally accurate sensory data. Force sensation in the cockroach robot is a particular problem; see section 7.4.5.2 . Similar problems were encountered when trying to use motor current as a signal to detect obstacle collision in the stick insect leg.

A particularly sexy possibility here is the addition of a set of strain rosettes or a six-axis load cell in the trochanter of the model leg. This full load information would allow the calculation of strain in any direction at that point, which could in turn be used to simulate the sensory response of virtual fields of campaniform sensilla. These could then be used for the detection of sensory events, in magnitude influences, or even just in observation of force traces difficult to obtain from the animal. Gravity compensation could be added to these signals, as well.

The range of sensors one might wish to add is really quite broad, but one other specific signal used in the stick insect, which is not readily available in the model, is joint velocity. Differentiation of low-resolution position data is noisy & involves delays, so is not currently used. This could be addressed with high-resolution position data or true joint velocity sensors.

9.2.2.8 Sensory Signal Processing

Biological sensory processing and filtering is generally highly capable and can be very complicated. Adding capabilities in the model for some basic general signal processing

could be very helpful in making some sensory processing more realistic. Examples of possible desired processing blocks:

1) differentiation/integration
2) sums/products
3) more configurable infinite impulse response (IIR) filtering- currently used hard-coded for the torque signal

In terms of the SCCILS configuration, it would probably make sense for the output of each of these blocks/functions to be another, separate "sensor" HDSO value; possibly one for each "wire" in the sensory processing block diagram.

9.2.2.9 Muscle Activation Function Dynamics

An important part of complete muscle function that is currently nonexistent in the robotic models is the dynamics of muscle activation. Current implementation is "null" dynamics, where the activation equals the input command. In the modeled insect systems, muscle activation follows a time course in reaction to a motor neuron spike train. Marcus Blümel has done extensive and highly detailed (spike-by-spike) work on the modeling of activation dynamics in recent work, however it should be possible to improve both structural accuracy and behavioral match of the model even using smoothly varying "command activation" numbers for motor neuron activity, and relatively simple dynamics governing the response of the actual muscle activation to those commands.

9.2.2.10 Joint Geometry Modeling

The current muscle model implementation has only the simplest joint geometry implemented- that where the muscle insertion is on a constant-radius "pulley", and the

effective radius of the muscle force is constant. This is usually not at all the case in insect legs, however; the apodeme often inserts in a hinge-like manner in the exoskeleton of the distal segment, and the effective radius of force application may go to zero and even change sign as joint angle changes. This is vaguely represented in Figure 4-3 showing elements of a neuromechanical insect joint.

It might be a substantial improvement in structural accuracy of the model to implement simulation of this kind of joint geometry. It requires trigonometric calculations, but if that is too computationally expensive the use of look-up tables might suffice. Eventually, it might be nice to allow modeling of muscle origins and insertions in three dimensions, and across multiple joint degrees of freedom. This would particularly help structural accuracy of models of the thorax-coxa joint.

9.2.2.11 Better Cockroach Kinematics

The cockroach leg kinematics presented in Chapter 5 have been used as a basis for finding more accurate kinematic data. Programs written by John Bender and presented in (Bender, Rutter, et al., 2008; Bender, et al., 2009 (In Prep)) bootstrap a kinematic model to more accurately represent the motion observed in 3D video of walking cockroaches. The data thus obtained on relative orientation of joint axes could be used to improve the physical robot or later models; currently the planar leg approximation is used.

9.2.2.12 Trochanter-Femur Joint

Current biological observation establishes that in straight walking and turning, the TrF joint is active and critical, perhaps even more so than the ThC joint (Bender, et al., 2009

(In Prep)). Inclusion of this joint in robots is therefore highly desirable just for walking and turning behaviors in addition to the climbing behaviors in which its importance was previously known (James T. Watson, et al., 2002). This joint is included in the next revision of the physical model, built by the Biorobotics Team Research class in the spring of 2009.

This of course requires a module to control that joint; my current first suggestion is to move and adapt the module used for ThC1 in this dissertation. It should be kept in mind as well that even in Bender's analysis the TrF joint axis is difficult to accurately find, and appears to be variable. I think the precise description of the kinematic properties of this joint in behavior would be a substantial and worthwhile project for a mechanical engineer to address.

9.2.2.13 Greater Use of the Thorax-Coxa Joint in the Cockroach Model

It is also expected that both degrees of freedom at the ThC joint (Mu, 2007; Rutter, Taylor, et al., 2007) will be critical for climbing behaviors. Current leg SCASM configurations essentially ignore the ThC2 DOF and hold it constant; investigating and modeling the control here will be new territory.

ThC2 is apparently also important in turning (Mu, 2007), though it was not used for that in this work. Though it has not been implemented, this control might be produced by adding a module for ThC2; if only tonic postural changes are desired it could even be done with only one action. The tonic postural pose at least, if not phasic activity, of this sub-joint is probably also critical to the initiation of the reflex cascade proposed in (Mu & Ritzmann, 2008b).

9.2.2.14 Feet

When considering any legged locomotion, the actual interaction with the substrate is primarily through feet. For whatever reason, however, feet appear to be the part of the system most thoroughly ignored when modeling or building legged systems. The current robotic models are quite footless; a smooth round ball is placed at the end of the tibia. Even adding a spring-biased passive pretarsus would probably be an improvement in model parity with the animal. More complicated designs are possible and, if allowed sufficient investigation and effort, certainly buildable. The tarsus on Matt Birch's cockroach front leg model was an excellent first try. The tarsus made in spring '08 by Dorothy for the Biorobotics Team Research cockroach middle leg model is of considerably simpler construction and is probably at least as good of a model. This version should be copied or (through direct reference to the insect) improved for later attempts. In particular, the two muscles allowing actuated sideways motion of the pretarsus relative to the tibia are not modeled in that design.

The more gradual change between swing and stance that a compliant foot affords is likely to change the sensing of ground contact by force. It would also, however, make these transitions less technically challenging from a control point of view, since the more gradual change in mechanical dynamics allows a broader window of time in which changes in control can effectively take place.

9.2.3 General System Improvements

These are basic improvements to modeling or robotic capabilities that don't fit in any other section.

9.2.3.1 Models with parts we can replace

Megarobotics, the company that made the AIMotor actuators used in the robotic models described here, apparently went out of business. Therefore, it is difficult or impossible to find new AImotor servos, which puts a pretty limited lifespan on these robots. The transmissions are mostly plastic and a bit under-engineered. For those interested in using SCASM for robotic modeling, it will be necessary to build new models using parts that can be replaced.

Some efforts at this have already been made; in particular the *B. discoidalis* middle legs built by the Biorobotics Team Research class at Case Western. SCASM control has yet to be implemented on them, however. Partly I think this is due to some underestimation of the difficulty of low-level basic control. Partly there appears to have been some miscommunication regarding precisely what signals are necessary for servo delta control. The AI Motor actuators had a number of things that helped with these considerably.

To implement SCASM more or less as described for the stick insect and cockroach systems in Chapter 4, one needs the following:

1) angle sensing
2) force, torque, or contact sensing
3) something vaguely resembling open-loop force control. It can be pretty vague, as first implemented in the BILL_ANT robot (Lewinger, et al., 2006). For biological modeling, however, it is likely that something closer, like direct current control or servo delta control, will be necessary.

To implement servo delta control, you MUST have a signal that reports the ACTUAL angular position of the joint, NOT the commanded servo position. Yes, the servo is supposed to just go to its commanded position. The fact that it does not do this under sufficient load, however, is how servo delta control works.

This also means that if your underlying servos are so strong and tightly controlled that they do reach the commanded position under the forces seen during the behavior you or modeling, servo delta control may NOT work. It is likely that the servos will need to be at least a little "under-torqued", from the more classical servo control point of view.

9.2.3.2 Basic Cleanup or Re-Implementation of the NeuRoMod Engine

This one is a doozy, and a complete re-implementation in particular should be considered very carefully before making a commitment to do it. The basic function of a SCASM system with constant parameters and minimal muscle modeling is indeed quite programmatically simple. If, however, you wan to record data, allow loading and validating of configuration files, and allow file-based configuration of different robotic models, it becomes much more complicated. This is especially so if you are doing it within the confines of a hard real-time system. Add the ability to modify control parameters online via user interface or experimental scripts, and you're getting to the point where it's a real headache to keep track of everything.

Now, the current NeuRoMod engine has some specific advantages and disadvantages, with respect to further development and debugging.

Pros:

1) RTLinux-GPL

a. Stable

b. Hard Real Time

c. Free

2) Modular and Extensible

3) Few limitations on what can be programmed

4) It already mostly works

Cons:

1) RTLinux-GPL

a. Poorly Supported

b. Needs access to brown- or black-belt Linux skillz for installation

2) Shared data management is somewhat barbaric.

3) Implementation is in C, but the program structure wants to be object-oriented.

4) Grew gradually from a moderately configurable leg controller to the current state; there is considerable cruft that assumes leg control or does not yet implement SCCILS-defined functionality.

The highlighted pros may be extremely important to you. A hard real-time programming environment allows you to count on scheduling of events to happen as you say, within hardware performance limitations. With the speed of today's computers it may be possible to get away without this in some cases. If you really care about the timing of your behavior, however, use of a non-RT system will require extra checking and verification that things are happening in the order you desire.

The modularity and extensibility of the NeuRoMod implementation is what allowed it to grow this far. In truth this is a little messy, and some extension or replacement of function will require more work than a “truly” modular and extensible implementation. It is good enough, though, that it may be reasonable to update and remove cruft one piece at a time, rather than starting over from scratch just to get a clean implementation.

The fact that it works to run robotic experiments and record useful model data is a very big plus. No matter how easy your programming language, this takes a lot of work.

The current low level of developer support for RTLinux-GPL is a problem; eventually this might lead to a need to find ancient working computers to install the software. This and the need for in-depth Linux knowledge might be addressed by upgrading to the non-free RTLinux-Pro; it used to be that they had a stated policy of supplying this software free of charge to researchers. I generally felt comfortable enough that I never got around to pursuing this possibility, but it would be a wise thing to look into. Another possibility is using the newer XtratuM/PaRTiKle real-time system (http://www.xtratum.org/), which can supposedly run RTLinux programs with little modification. This would require installing the new OS and probably some search-and-replace in the code, but possibly not much else.

The implementation in C is definitely annoying. The last time I looked it was possible to use C++ in RTLinux, but it was unclear whether objects could be shared between real-time and user-space programs. Nearly all of the quasi-object-oriented data in NeuRoMod is in shared memory, so this would be a deal breaker.

On the other hand, the code is only as object-oriented as it needs to be, and quite efficient. While pseudo-object and memory management have to be done explicitly, there

is no overhead associated with automatic garbage collection, object and function passing, and so on. This allows the programs to run very substantially faster than might be the case in, say, Java or Python.

The cruft and inherited limitations from the original implementation are annoying, but it may be useful to take note of the fact that I was able to add muscle models, script and data server elements, and extensive runtime data interaction to the original controller. At none of these points did I actually consider a complete code rewrite. Most or all of the cruft issues could be addressed without a complete rewrite.

When I began the coding for the controller, MATLAB real-time tools and LabVIEW were too limited or insufficiently hard real-time for the kind of unlimited extension and higher-level processing I wanted to allow. It is possible that these tools have improved sufficiently to allow such flexibility. Diagram-based programming of event-space diagrams would be nice, as long as it didn't carry too many limitations with it. In particular for use in modeling and controller development, the ability to speak SCCILS is a big plus, and can take considerable work. Tested Python and MATLAB code for partial handling of SCCILS XML does exist, however, in the NeuRoMod GUI and in SCCILStoolbox.

9.2.3.3 Treadmill/Treadball

It is a reasonable match to some experiments, but the slippery feet used to simulate the oiled plate experiments do make some parts of stepping more difficult, and sometimes provide a poor behavioral match to freely walking or treadband/treadmill/treadball experiments. As one example, a more grippy foot on a treadmill for the stick insect leg model would have considerably less trouble with erratic behavior arising from poor

detection of ground contact. It is even likely that this setup would make the detection itself more reliable as well.

It is desirable, therefore, to have some sort of tread-surface and feet to interact with it. There is a more in-depth discussion of feet above in section 9.2.2.14 , but the requirements here may not be so involved. In the first round, adding a high-friction rubber ball at the end of the tibia instead of a low-friction plastic one may show sufficient improvement to be interesting.

With such a foot, it should not be too difficult to affix an appropriate small mounting scaffold to the robot treadmill in Glennan 814, and get some treadmill stepping going. Of course, I do not know how well that treadmill itself is actually functioning.

9.2.3.4 Teleoperation Development

There is considerable work left to be done for improving the ability to conduct experiments and otherwise operate the model robots from a different room- be that across campus or across the globe. During model experimentation it is often useful or even required to reposition the robotic leg between trials, and it can be very educational to interact physically with the leg during behavior. Both sound and video monitoring by the teleoperator are highly valuable but do not re-create the full experience.

The low-level position control software exists in NeuRoMod for doing leg repositioning, but there are currently no higher-level HDSO hooks for specifying angle commands, or switching between angle and torque control at a joint. With these tools in place, real tele-experimentation without significant local human supervision becomes a possibility.

Replacing physical interaction during behavior over distance is not reasonably possible, but it might be useful to add a "tweak torque" at each joint, which could be interactively specified through the UI and added to the controller-specified torque. A more difficult but possibly useful task would be to add a virtual "tweak force" at the foot as well; this requires inverse dynamics, however. This is not a simple problem, especially for a highly flexible robot with low-resolution position and force data.

9.2.4 Neural Delays

There is currently only the most rudimentary modeling of anything resembling neural dynamics in the NeuRoMod software. Modeling of neural and synaptic dynamics may eventually be desirable, but that is a huge can of worms, and that's all the treatment I'll give it here. Here are some possibilities for more tractable improvements. Only neural delays are covered here, inherent CPG dynamics and central coupling are discussed in the CPG section (9.4).

As described in section 6.2.1.3 , all delays in all sensory-motor control loops are currently simulated by setting the period of the state transition thread in NeuRoMod. This approach has a number of drawbacks. Firstly, though it does produce a delay in controller signal propagation, it also increases the time period over which events are detected, making it a worse model of a continuous system. A continuous system with delay can detect events at any time, but will not react to them for a little while. This implementation can only detect events at specific points in time. Secondly, this time delay also affects the speed and time continuity of the muscle model dynamics calculations. These calculations are a part of the controller that is pretending it is part of the physical reality. Any such

real-time simulation of continuous-time physics should usually be done at as high a rate as practical, and its fidelity should certainly not be affected by sensory delays.

It might therefore be useful to incorporate a more structurally accurate way to model neural delays, both afferent and efferent. This would require some bit of the program to store recent data for delayed signal lines, and report the appropriately delayed data to processes looking up that line. This could be done either by defining a delay for a sensor/signal variable, or, more generally and correctly, defining a delay for each of the various recipients of a signal. This might be best accomplished by including a "delay" element in the signal processing blocks described in section 9.2.2.8 , and adding as many delay elements as necessary to produce the appropriately delayed signals. Another option might be to augment HDSO specifiers (e.g. used in specifying a Boolean transition rule or a muscle activation function) to optionally include delays. For instance, one might try:

(324 > 101) means "is HDSO 234 greater than HDSO 101?"

(324,0.001 > 101,0.05) means "was HDSO 324 1 ms ago greater than HDSO 101 50 ms ago?"

This would, of course, require some quickly accessible storage for the signal history of each HDSO. This is not usually difficult, but in cases where the HDSO is a calculated value (e.g. another Boolean expression) the requested delay would need to be added to each delay used in that calculation.

9.3 Model-driven & Model Experimentation

Really, the point of a model is to support the process of explanation. In the model improvements section above, many of the improvements were related to specific

biological functions or questions, but this section is intended to outline actual use of these models for experimentation and analysis.

9.3.1 Walking and Turning

A major motivator for the biological modeling in this work was the investigation of the interplay of descending commands and local control circuits; specifically to test the reflex cascade hypothesis. Since the behaviors most used for addressing this question in the Ritzmann lab are walking and turning, it would be sensible for these to be a significant element of further model experimentation.

9.3.1.1 In The Cockroach

All of the cockroach model control presented in chapters 4 and 7 represents hypothetical neural mechanisms, none of which have yet been tested. It would be a really good idea to start testing them, if possible. Some suggestions are given in the conclusion of chapter 7. Additionally, it would be a useful exercise to try to come up with behaviorally equivalent or similar alternate event-space diagrams for the walking and turning behaviors described. If many alternates are found, that indicates a greater input of biological data to select the most likely hypotheses.

An explicit experiment that Bender is planning to run was diagrammed in (Rutter, Bender, Ritzmann, et al., 2009) and is shown in Figure 9-1. Further experimentation would make use of extracellular or electromyogram recordings to help discern between local and higher-level responses.

As a map of the actual neural elements involved in these behaviors is developed, animal experiments similar to the more involved experiments described in section 7.4.2

may become possible. Specific local influences might be reversed, or delay in reversal may be possible.

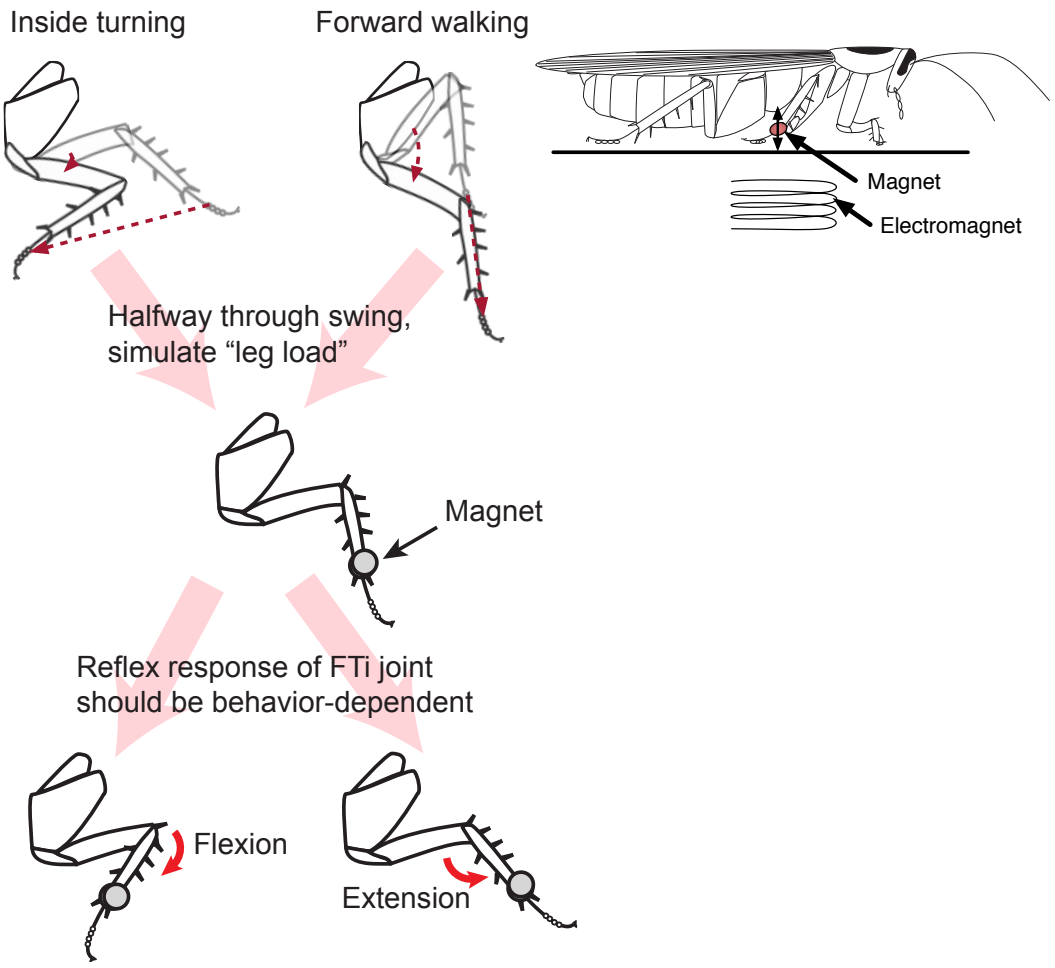


Figure 9-1 A small magnet will be affixed to the end of the tibia of the middle leg. An electromagnet will then be used to modulate the force applied at the tibia. This experiment should be used to test the hypothetical reversal of load influences at the FTi joint controller between inside turning and forward walking.

It should be useful to incorporate known cockroach magnitude influences into the design of the SCASM control configurations for walking and inside turning. Once functional in straight walking and inside turning, these and further hypothesized

magnitude influences might be sufficient to test the specific reflex cascade hypothesis presented in (Mu & Ritzmann, 2008b)

Investigate whether the "outside turning, backward" qualitative behavior shown in Figure 9-2 appears to happen in the animal. Probably it does, but under what circumstances? Actually in turning? The global state diagram for this behavior could be translated to an event-space diagram, implemented in the model, and used to generate testable hypotheses.

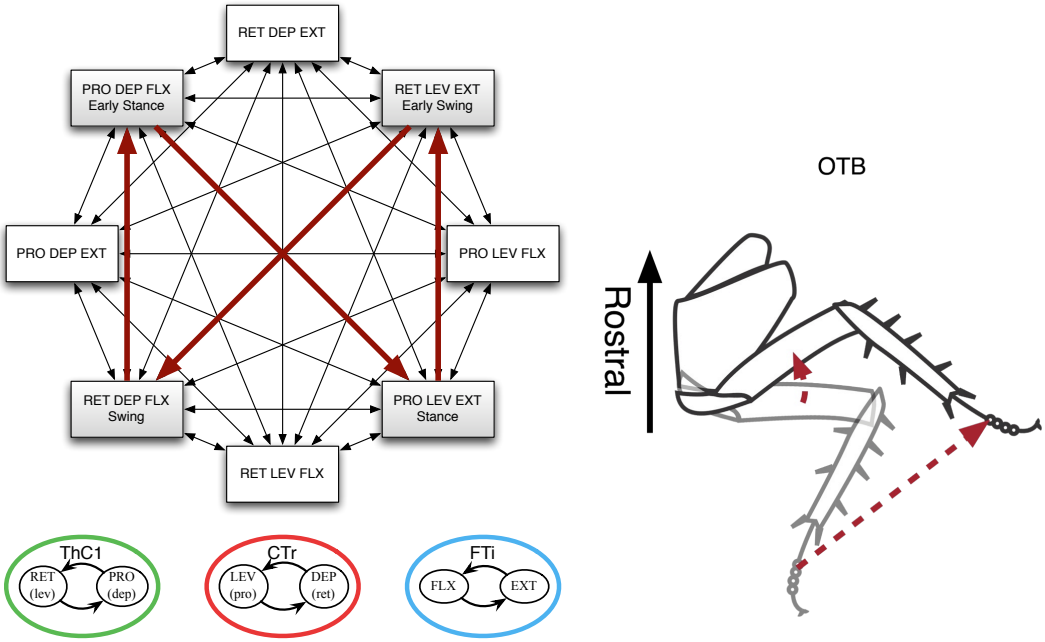


Figure 9-2 Global state diagram and behavioral sketch of the "outside turn, backward" behavior.

9.3.1.2 In the Stick Insect

There is increasingly available data on stick insect turning, and it would be a great idea to use a stick insect model leg to investigate and/or verify some of the mechanisms involved in turning and transitions between walking and turning in this system. Ideally this would include collaboration with Matthias Gruhn, who has done stick insect turning work in Cologne (Gruhn, Hoffmann, Dübbert, et al., 2006), and Bill Lewinger, who has

implemented turning in a hexapod robot using SCASM (Lewinger & Quinn, 2009), and who is now working on a stick insect model hexapod in Edinburgh.

9.3.2 Other Behaviors

9.3.2.1 Searching

The cockroach automatically switches from walking to searching upon removal of substrate; the model does not. Investigate whether reasonable additions such as magnitude-influences suggested in (Mu & Ritzmann, 2008b), or perhaps inherent CPG dynamics, can cause the controller to automatically generate this behavior locally in SCASM. If not, is some higher-level (possibly still in the ganglion) processing easier to implement? Once the model exhibits the behavior, suggest biological experiments to test the implemented hypotheses.

9.3.2.2 Climbing

With operational ThC and TrF joints, and possibly feet and tibial spines, it will be reasonable to generate control configurations for climbing, and investigate transitions between these and walking in a manner similar to that done for turning in this work.

9.3.2.3 Higher-Level Behavior.

Once a model insect capable of locomotion is constructed, and the leg-level behaviors and transitions between them are sufficiently understood, it will be possible to begin investigating the generation of higher-level behaviors that actually make use of descending commands, such as navigation to a goal. Though more abstract in its biological basis, Lewinger has already begun this (Lewinger, Rutter, & Quinn, 2008).

9.3.3 Speed analysis

9.3.3.1 Further Speed Control Characterization

As discussed in section 7.3, changes to gated tonic drive of muscles can change speed in stick insect and cockroach robot models. This data is very preliminary, and further description and sensitivity analysis could be useful; this might guide experimentation in the animal. Look at prior motor neuron activity descriptions (Gabriel & Büschges, 2007; Watson & Ritzmann, 1998b), and investigate whether changes in magnitude influences might control speed as well as or instead of such gated tonic drive. Mention method, and tests in both. Do experiments in the model to determine which muscle activations may be most critical for speed modulation; see if experiments can be done in the animal to simulate or block such control.

9.3.3.2 Analysis of effects of speed on step variance

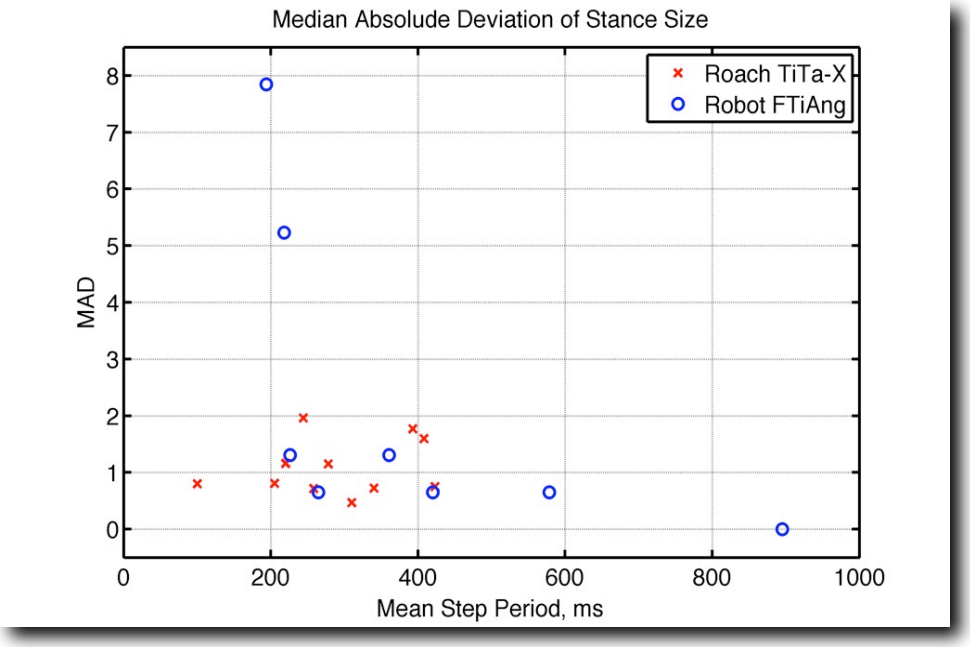


Figure 9-3 Median Absolute Dispersions of stance excursions for x-position of the tibia-tarsus joint in the cockroach, and for the closely related femur-tibia angle in the robot. The robot is not capable of periods smaller than about 200 ms, due to limits on motor speed.

Testing either hypothesis for complete sufficiency in the animal is difficult at best. It therefore makes sense to go a step further in the model. When and how does it fail to reproduce animal behavior? In the preliminary results for the tonic drive hypothesis here, high stepping speeds give rise to greater dispersion in the physical extent of stance in the model system, but not in the cockroach itself.

Looking more closely at the two graphs together; higher muscle activations reach a point where they slow down stepping, but the over-driven example has a lower dispersion than the two trials that were faster than it. This suggests speed may be more important than drive for variability in sensory pattern generation. A more full analysis of variance with speed has been done in (Bender, Simpson, & Ritzmann, 2008), and may be presented in (Bender, et al., 2009 (In Prep)).

9.3.4 Muscle models

9.3.4.1 Impact of Muscle Models on step variance

Initial investigations were for one joint only; it would be interesting to do a full test of combinations of muscle models in both cockroach and stick insect robots. It is likely that improving muscle models at some joints fixes specific problems in behavior (e.g., FTi and the swing-stance transition); look into these. This investigation is quite interesting from the engineering point of view; it could also be useful in finding which muscle properties may be most critical in the animal for different behaviors. One might then hypothesize that muscles whose properties or activity is most critical would have the most tightly controlled or constructed properties. Measurement of variance in animal muscle properties may be difficult, but some experiments could be tried.

9.3.4.2 More Sophisticated Muscle Models

The "muscle models" used in (Rutter, Lewinger, et al., 2007) and described in section 7.2 were extremely simple, but had a significant effect on system behavior. It would be useful to investigate to what extent further complexity in muscle models affects behavior. It would make sense to start with variance tests as before, but it would also be good to measure the effects on performance in dealing with rougher terrain or other environmental perturbations. Either a treadmill or a true walking robot would be necessary for this. It is reasonable to expect improvements in quantitative behavioral match, so it would make sense to test this as well.

9.3.5 Impact of Neural Delays

Once neural delay simulation was implemented, one could observe changes in step variation, ability to deal with environmental changes, and apparent stability as all or

specific neural delays are changed. Look over a range of stepping speeds! Start with comparison of zero delay and unreasonably long delay; find points/regions where the behavior changes. Find specific pathways whose delay affects overall system behavior the most. If joint and/or full-leg central pattern generation are implemented in the model, it would be interesting to test the effects of sensory delays on stability in three different regimes: no central pattern generation, uncoupled joint CPGs, and central coupling for a whole leg CPG.

9.4 Central Pattern Generators

Sensory coupling is a defining element of SCASM systems, and the current models of both cockroach and stick insect legs use only sensory influences for the generation of coordinated stepping. The insect neural system is capable of somewhat more centrally-driven behavior, however, and modeling this capability will allow the investigation of potentially more realistic model behaviors and associated neural function.

9.4.1 Inherent CPG dynamics

In the stick insect neurobiological system, the Action Switching Module for each joint has the capability to act as an oscillating central pattern generator (CPG). The existence of these inherent CPG dynamics is demonstrated in experiments where the thoracic ganglion is excited with pilocarpine, and each joint CPG oscillates slowly, individually, and without sensory input (Büschges, et al., 1995; Ekeberg, et al., 2004).

It is expected that CPGs could improve system stability and robustness. We have already shown (Rutter, Lewinger, et al., 2007) that simple muscle models decrease the variance in stepping behavior in the model. It is suspected that this is at least partially

accomplished by conditioning the system outputs such that the entire leg is more likely to stay within the feasible operation space of any particular SCASM control configuration. Nevertheless, it is possible in the absence of CPG dynamics to get "stuck" in a state where a sensory signal critical to the transition to the next phase of behavior does not occur. (This is particularly likely when the SCASM configuration is "not optimal"). In these cases, internal CPG dynamics would eventually trigger a transition themselves, thereby allowing for continued system operation.

9.4.1.1 State and/or Module/CPG inherent dynamics

Within the context of modeling these legged systems, each module of the SCASM system representation is sometimes referred to as a central pattern generator (CPG). This is due to the fact that each of them is modeling an element which, in the stick insect, can produce alternating activity without sensory feedback, at least under excitation induced by pilocarpine. The models of these circuits so far tested, from (Ekeberg, et al., 2004) to here, have modeled these PGs entirely as bi-stable state selectors, without any inherent switching dynamics.

The biological circuits do possess the ability to exhibit such dynamics, at least under some conditions, and it would sometimes even be sensible for them to do so. In particular, it is possible that properly tuned CPG dynamics would increase overall system reliability. If each CPG is tuned to switch at a rate slower than the currently exhibited sensory-driven switching, it will have no effect on the system under "normal" operation. If, however, the leg is exposed to some perturbation that sufficiently delays the normal action switching, or somehow a sensor fails to detect a switching event, the inherent dynamics of this CPG would have an effect, kicking in and continuing the progression of

local events likely to produce the desired behavior. This possibility is particularly obvious in the robotic models, where poor sensory signals are often responsible for a disruption or ceasing of the current behavior.

Ideally, one would tune these inherent CPG dynamics so that they were continually a little slower than the current stepping rate; this could be done with some leaky-integrator signal for CPG frequency, which would be integrating either sensory switching events or the module switching itself.

As it happens, there is some initial data available (Borgmann, Hellekes, & Büschges, 2009) which shows exactly this effect in the stick insect- a sensory signal which stimulates alternating motor neuron activity, whose frequency decays to zero over a few cycles. It is suggested that this could be a very interesting area for modeling and combined model/biological investigation.

One way to implement CPG dynamics would be to have each action keep track of how long it has been active via some decay variable. The simplest implementation would just have a timer for when to switch to the next action. For multiple-action modules, there would be a timer for each other action, and the action switched to would be the one whose timer expires first. This implementation is equivalent to setting a switching threshold on a time decay variable that decays linearly. A somewhat more capable implementation would include a refractory period as well, in which switching to other states is not possible; this is in fact already implemented and configurable as the SCCILS dwell_time element for each state. This was added to allow sensible response to obstacles and holes in the stick insect leg robot, as described in (Lewinger, et al., 2008).

A more biologically plausible implementation might involve an exponential or possibly power-function decay of the state time variable; events depending on this variable could be triggered by setting thresholds for each such event; e.g., each other state in the module would have a threshold on that decay variable as part of its transition function.

9.4.1.2 Experiments in Joint CPG Dynamics

Useful experiments and observations here are similar to those recommended for investigating sensory delays. Investigate how close to nominal stepping speed the CPG dynamics can get before they modify the behavior. Investigate how CPG dynamics modify how quickly the system can change speed. Specific testable behavioral hypotheses are possible here regarding whether there is active inhibition of the CPG dynamics when an animal slows down.

9.4.1.3 Central Coupling of Joint CPGs

For a system to be represented as SCASM, it must be the case that sensory coordinating signals can override central coupling and dynamics. It is not necessarily the case, however, that these central connections do not exist or serve no purpose, in fact there is evidence for at least some central coupling during swing-stance transitions in stick insect walking (Büschges, et al., 1995). Weak central coupling at low speeds might be useful in increasing stability, acting as a fallback if sensory coordination somehow fails. Stronger central coupling may become very helpful or necessary at high speeds, particularly in the cockroach- this is discussed in more detail in section 9.3.3.2 .

It will be fairly conceptually clear to add overrideable central coupling to the NeuRoMod controller. The easiest way to do this would be for each module/joint to have

a time decay HDSO variable, along with a "current action" variable for that joint. Central coupling could then be added by including an OR dependence on these variables for state transitions. For example, in reference to the minimal SCASM stepper in section 4.4.1, the transition rule for entering ThC protraction might be changed from

ground_contact_sensor < no_contact_threshold

to

((ground_contact_sensor < no_contact_threshold) || ((CTr_state == levation) && (CTr_statedecay < decay_threshold)))

It should be fairly clear that whether this is weak, rarely active central coupling or strong, usually active central coupling depends on whether the sensory or central threshold triggers first. As for internal CPG dynamics, it would be very sensible for decay_threshold to be a function of the speed of behavior, modulated by sensory input and/or central drive. At least one of these is necessary, in fact, for the system to be able to stop stepping when some higher-level drive command is turned down to zero.

Note that although central coupling and joint CPG dynamics are conceptually somewhat different, the requirements for implementation in NeuRoMod are nearly identical.

9.4.2 Speed, Modulation of CPG dynamics with speed

With internal CPG dynamics, it will be necessary to scale these dynamics with "speed" in order to allow the full range of speeds currently available in the model.

9.4.2.1 Impact of Muscle Models

Speed changes can be implemented in the model merely by changing muscle activations; no rearrangement of the local control network is necessary. It is expected that more advanced muscle models, perhaps including the modeling of muscle activation dynamics, would make the response of the local neuromechanical system more forgiving in terms of both the acceptable ranges for descending commands and the necessary timing precision of the coordinating CPG influences, whether sensory or central in nature.

9.4.2.2 Central CPG Coupling at higher speeds

At high enough speeds in a SCASM system, it is feasible that delays in sensory processing would prevent the sensory coordinating influences from arriving quickly enough to produce well-coordinated behavior. There is behavioral evidence (Bender, Simpson, et al., 2008) that central pattern generation may be taking place at least at higher speeds in the cockroach, supplementing or overriding the sensory pattern generation that appears sufficient and adaptive at lower speeds.

9.4.3 Sources, sinks, and function of variability in behavior

An inspection of the distribution of various step metrics during behavior in the robotic models reveals substantial variation that is not easily predictable step-by-step. See section 7.2 for some examples. I initially thought that this was due to some sensitivity of the system to small amounts of sensory noise. This is very likely part of it, however Marcus Blümel reports (pers. comm., June 2008) that the computational simulation also exhibits variability between steps, without reasonably discernable patterns of repetition. This suggests that the SCASM leg coordinating systems, at least as modeled, are inherently

chaotic in nature. The chaotic behavior produces variation between steps, but at least for the most part it does not interfere with the overall generation of behavior.

At least from the engineering standpoint, there are some very interesting questions raised here. It is plausible that some random and/or chaotic variability in behavior is adaptive. The world is often a highly variable and non-constant place, and to some degree variable behavior may improve performance of tasks in such environments in a computationally cheap and robust manner. If you barely missed getting a foothold with this step, just taking another chance might work much of the time, without the need for more complex, directed, or higher-level modification of the behavior. A big question here is: what are the functions and advantages of variability in behavior? Looking at "steady" locomotor behavior may be a useful way to start addressing this question.

Indeed, most biological systems are inherently variable or "noisy". It will be easy for most readers to believe that these systems must have mechanisms to reduce or deal with this variability; "sinks" for the variability. The muscle modeling work above suggests that muscles may be an important variability sink in these stepping systems. Tuning muscle properties, then, could be one way of affecting how much variability actually makes it to the behavioral output.

Another, slightly less usual question is: where does this variability come from? There are plenty of sources of noise in biological systems, but which of these might actually be used in an adaptive way? Are certain elements of the system generating variability "on purpose"? Are systems with some inherent chaos somehow favored over other, possibly non-chaotic systems? Is there such a thing as a non-chaotic biological system?

These are not particularly easy questions to address- however in the models presented here one can at least start looking at the question of sources and sinks. Figuring out what the functions of variability are would be substantially helped by an ability to modify this variability. It would take considerable coding, but it is not conceptually difficult to add variability into various sensory and other signals in a SCASM system. What effects does this have on behavior? It may be possible to artificially inject variability in the animal system as well, via pharmacological means increasing "jitter". As mentioned above, there has already been some work on muscle models as sinks of variability, and the effects of this on behavior. Are there some sources of variability or noise that muscles do not sink? Which muscles are the most responsible for modulating variability from which sensors or internally generated "noise"? What other system elements (e.g. neural signal filters) might act as variability sinks?

With such tools, it should be possible to start investigating what adaptive functions variability may have. Is there some non-zero optimum of behavioral variability or "noise"? Specific questions include: A) on a flat slippery surface, do some types of variability allow for longer periods of operation without failure? B) In more complex terrain, do some increases in variability allow improved ability to overcome some difficulties?

There is biological experimentation to be done here as well. Observation of whether step variance metrics differ between behaviors might be informative. Are there biological behavioral experiments that we can develop in the model that can elucidate the role or existence of more central coupling in the animal? Are there ways we might be able to disrupt central coupling in the animal and see what happens?

9.5 Model User Interface and Data Representation

The experimental work and model improvements described above require a functioning user interface to be doable and useful. The graphical user interface (GUI) included in NeuRoMod is basically sufficient for many tasks, but improvements will be necessary for enhanced experimental utility. This includes improvements to the naming and display of system elements, configuration-specific help regarding the layout and function of the various sensory events and event discriminators.

Data collection and live viewing could also be integrated into the GUI, in order to speed up the experimentation/analysis cycle.

Further improvements could include ability of the model user to load, edit and save model configurations and experiment scripts. A more thoroughly graphical representation of and interaction with the model could really substantially improve and support the representing, interpreting, and demonstrating elements of model usage. Work on these things might be at least an M.S. amount of work in UI design and data representation; investigators at Case Western might want to talk with the gaming lab about it.

The following two sections give examples of things that would definitely be useful; they are by no means a complete description of what might be possible in this regard.

9.5.1 Incorporation of Representational Diagrams Into GUI

Currently the NeuRoMod GUI does not present any truly graphical representation of system configuration or function. A number of the graphical tools and system representations presented in Chapter 4 could be programmatically generated.

9.5.1.1 Generation of event-space diagrams

This is probably one of the more useful and more difficult diagrams to generate. The SCASM elements of this graph and their connections can be generated from a SCCILS configuration in a relatively straightforward manner using the method described in Section 4.5.3. Choosing where to place the modules and how to automatically generate the color and line-style codings is more involved but certainly possible. Automatically generating a leg diagram on which to place the modules is also doable from the Denavit-Hartenburg parameters in a SCCILS jointlist, but this is a lot more work, and may not be as useful to the user as providing an pre-generated background image of the physical system, with defined places for module placement.

9.5.1.2 Boolean State-Space Rule Representations

For modules with compound sensory events, it can be useful to construct a diagram such as Figure 4-10 to help insure there is no overlap between the areas in sensor space that specify each switching event. With the current layout of the UI, it is not even necessarily very easy to avoid this overlap when only a single sensor is involved. The automatic generation of such a diagram requires the capability to construct line, area, and possibly volume inequality graphs. It is no more complicated than that, however.

9.5.1.3 Global State Diagrams

It is difficult to construct a global state diagram without knowledge of which actions tend to lead to which sensory events. With this information (possibly user-supplied), a predicted global state diagram can be constructed from an event-space diagram or a SCCILS configuration. It is a somewhat less difficult programming problem to construct an *actual* global state diagram from data collected during a behavior- this can be useful

both to compare actual behavior to that predicted, and to investigate which non-nominal global states are reached under certain perturbations or behavioral transitions.

9.5.1.4 Diagram Interactivity

All of the above diagrams would be made even more useful by adding the ability to show live data, or generate an animation or plot from recorded data. Additionally, the event space and Boolean sensory space diagrams could be used to input or modify configuration parameters in a conceptually straightforward way. Dragging an arrow from a sensory event to an action transition could make it part of the switching event for that action (though of course a few more parameters must be entered to specify exactly *how* it effects this action). Dragging an inequality boundary on a Boolean sensory space diagram could modify the associated switching event threshold.

9.5.2 GUI User-Friendliness

There are a number of less difficult things that could be done to make the current GUI more broadly useful.

9.5.2.1 Clickable Map of Leg

The ability to bring up the configuration information for a joint by clicking on an image of the model (or insect) would make some user interaction much more quickly learnable. Further advances here might include single muscles and sensors as clickable elements as well. Basically this requires only an appropriate image and a map of which parts of the image should bring up which UI configuration panes; it would probably be possible to add this to the existing Python GUI.

9.5.2.2 More Complete Naming and Help

This is mostly a change only in the XML configuration files, but more complete and clear naming and display of states, joints, and HDSO elements could make some user interaction considerably more self-explanatory. Primarily this consists of completely and appropriately setting the “name” and “fullname” XML attributes on SCCILS joints, states, sensors, HDSOs, etc. Many SCCILS elements also contain a “description” element, in which description of element function and help for the user can be placed for display at runtime.

9.5.2.3 Move From HDSO ID to HDSO Name in GUI and SCCILS

The use of HDSO ID numbers to specify data in the configuration files and UI does make the engine and translator programming easier, but it introduces an extra layer of obfuscation for the user. Moving towards a primarily name-based SCCILS implementation requires more string processing but would substantially improve user clarity. In the NeuRoMod engine this would require an additional HDSO_get_byname function (at least) in the parsing code, in addition to the changes necessary in the HDSO and jointlist DTDs.

9.5.2.4 Improved GUI File Handling and Editing

Look at my notes and requests in conversations with John Bender. Basically, it doesn’t quite work, and it would be nice if it did. It would take some work, but it might be worthwhile to also reduce the current duplication of SCCILS translator-type functions, which are implemented both in the engine and in the Python translator, in two different languages, by two different people.

9.5.2.5 Integrated and Live Data

Better display of data, and better handling of data logging command would be a big plus. Live data traces that redrew nicely would be very good, implementation of doing this by a specification of data rate feeding rather than polling would be great. Incorporation of SCCILStoolbox plot elements into the UI could be helpful.

9.5.2.6 Higher-Level Control Parameters

The ability to configure higher-level control variables might be good. A relatively easy example would be an HDSO parameter (with associated GUI slider) for "speed", which would set specified changes in a set of muscle activation parameters as it slid. This could be done using the data in Figure 7-11, for example.

9.6 Theoretic Understanding and Development of SCASM

9.6.1 Formal mathematical definition of SCASM

One of the most useful tools for theoretical understanding, development, and comparison of SCASM to other systems will be a formal mathematical description. It is possible this will require two parts or stages; one for the mathematical description of system and signal connections (quite probably the event space diagram or some very similar representation), and another for the description of what precisely a module may consist of, and how it may switch between actions. It is possible that a considerable amount of use could be gotten out of the first stage only. It is my feeling that the first stage at least should not be particularly difficult or involved for somebody who is moderately proficient with mathematical descriptions of connection graphs and trees.

An alternative and possibly more quickly fruitful method for formal definition of SCASM would be to derive such definition from the SCCILS XML language definition. In order to make this complete and correct, it would be necessary to include true XML schema in the language definition, to implement limitations which cannot be expressed in the DTD language but are mentioned in the comments of the DTDs or in the descriptions in Chapter 4.

9.6.2 Mapping of SCASM to understood/previous control representations.

There is a very substantial body of work in the formal description and analysis of discrete-event and hybrid systems. Even if it turns out that SCASM systems cannot always be mapped to some alternate method of description such as Petri nets ("Petri nets," http://www.petrinets.info), it is likely that limited mapping or mapping of a constrained SCASM system would still be useful in analyzing and predicting system behavior.

If it does turn out that SCASM systems can be fully represented using some already-existing tools, this is likely to be a good thing for helping both biological modeling and engineering synthesis. It seems likely, however, that some of the representational and synthesis tools presented here would still prove useful, both for ease of mapping to biological systems and for straightforwardness of controller synthesis. If such systems were already clearly presented and well understood, it is hard to understand why they would not be in much broader use for this purpose at this time. It would still be worthwhile to spend some effort looking for them, however.

9.6.3 Description of abstract system types in which SCASM can be used

A broad suggestion is given in the conclusion: “any multi-dimensional control system that needs to go through a coordinated set of states in concert with the environment”. This would be done implicitly in any formal description, however it would be useful to more thoroughly describe systems and system types which can and cannot be described by or implemented as SCASM. With sufficiently open restrictions on what an action may do, of course, it may be possible to represent any dynamic system “as SCASM”, if one includes the degenerate single-module case and the even more degenerate single-action case. There will be limits regarding to what extent this will be instructive or useful, however.

9.6.4 Further Methods for Generating Modules and Coupling Influences for Desired Behaviors

This is already partly done in the synthesis description of section 4.4.1, however there are a number of limitations and design trade-offs that are not formally addressed. Formal description of the full system including mechanics as a hybrid dynamic system might allow more explicit accounting for mechanical coupling between modules, for instance.

9.6.5 Methods for generating Modules and Potential Couplings for a Desired Range of Behaviors

This is not exactly the same as above. Whereas that is a per-behavior generation and related to a specific event-space diagram such as Figure 4-4, this one would be more like generating all locally necessary nervous pathways; not just those active in a particular behavior. This is to some degree the information contained in a sensory-flow diagram such as that in Figure 4-2.

9.6.6 Stability Analysis

The formal analysis of stability here may be quite difficult; however it may be possible to represent modules, SCASM systems, and hierarchical SCASM systems in such a way that previously developed methods for analyzing hybrid systems would apply. A good deal can be done with the implicit testing of stability involved in interactively developing robot controllers. There are a number of applications for which this is not sufficient, however.

9.6.7 Implementability of Hierarchical Sensory Coupled Action Switching Modules

Possibly, just try it and see what the problems are. Including "A SCASM system" recursively in the formal definition of what a module may consist of would take care of this from a theoretical standpoint, but might be a particularly tricky problem.

9.7 SCCILS Generalization

Although those of us working on this have made the argument that SCASM can be useful in the description and design of a fairly broad range of systems, the current SCCILS language has a number of things that limit it. Firstly, there are limitations related to hardware-specific assumptions and elements; primarily these are in the aimotor and sensor elements. Secondly, there are limitations related to the assumed description of a leg actuated by simulated muscles. Generalizing the description of modules and actuators should certainly be doable. Note that NeuRoMod need not necessarily implement more general control in order for this to be useful; a more general SCCILS definition might be useful just to allow easier use of the SCCILStoolbox data handling and representation functions by a broader range of systems.

9.8 Use of SCASM in Other Systems

Formalization of SCASM and generalization of SCCILS have use in their own right. In my opinion, however, it could be very useful and instructive, as discussed below, to actually apply these to other systems. To some extent this is already being done. Broader application in biorobotics could include swimming and flight as well; one place to look for possible applications is anywhere anybody has tried to use centrally coupled oscillators.

9.8.1 What's Been Done So Far

Of course there was the original simulation done by (Ekeberg, et al., 2004), and the autonomous two-leg test platform of (Lewinger, et al., 2006), shown in Figure 9-4. Lewinger has also applied SCASM to the control of a small autonomous hexapod (Lewinger & Quinn, 2009), and is currently working on an 18.1:1 scale stick insect hexapod (pers. comm.). Also, Brian Taylor is making the first attempts at applying SCASM to a non-leg control problem; (Taylor, Rutter, & Quinn, 2009) work on plume tracking.

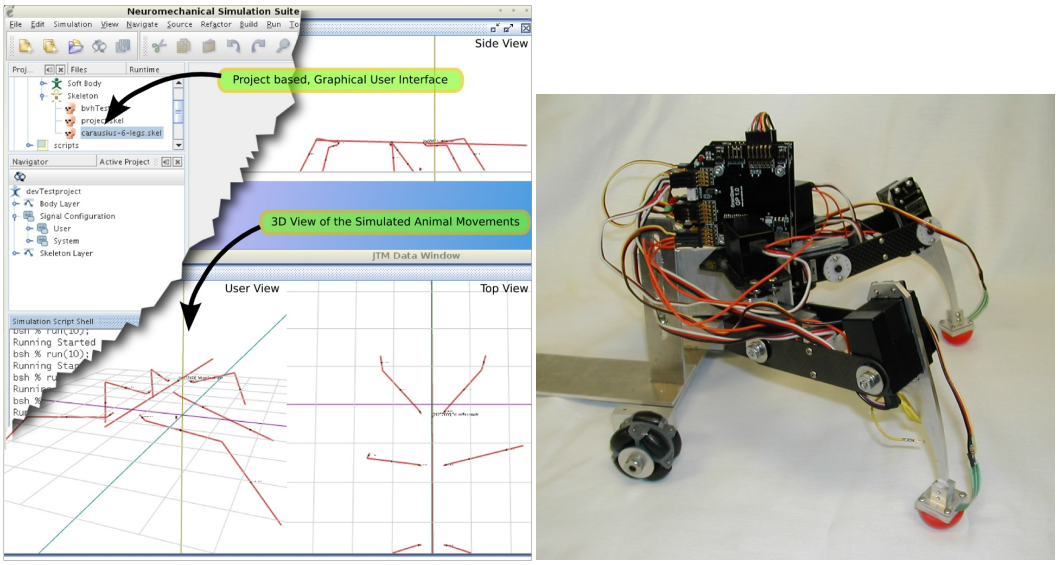


Figure 9-4 Left: a version of Marcus Blümel's stick insect simulation. Right: Bill Lewinger's two-leg test platform. Both from (Rutter, et al., 2006).

9.8.2 Make it Easy

One of the problems with SCASM, and it appears more generally with "smart" motion control, is that somebody who is building a robot often has a very strong desire (or, at least as often, outside pressure) to just get the bloody thing to move. It is very common for potentially highly capable robots to therefore have some "simple puppet-mode" position control applied for this reason. The problem with this is that implementing puppet mode position control is rather a lot of work in itself, particularly for systems with many degrees of freedom. Then, once the robot is moving more or less in a way that looks reasonable, the temptation is high to hand-tune the puppet files to produce a more functional behavior under some limitations. THIS TAKES A LOT OF WORK AND ATTENTION, WHICH SHOULD NOT BE IGNORED OR UNDERESTIMATED. If this effort were applied to implementing or applying a more intelligent motion control method, of which SCASM is an example, it is likely that very substantially better results would be obtained in the end.

Why is this not done? “Milestones” is one answer, but the real or perceived complexity or implementation difficulty of smarter motion control is a big part of it as well. Often, such things require or greatly benefit from another person working on the project; in the graduate lab setting this is unfortunately often unaffordable, undesired, or unreliable.

It might therefore be a very substantial contribution to produce a set of representations, code snippets, and programs that were well-known and understandable enough that more people would skip the very limited puppet-control stage of development. There was no such stage in the development of the models presented here, and as far as I can tell it would have been substantially a waste of time and resources.

9.8.3 Implement SCASM with Artificial Muscles

A lot of muscle modeling is moved to hardware in this case; controller code becomes even more lightweight. Finishing/improving Michael Cohen’s cockroach leg model (2006 senior project in the Case biorobotics lab) and implementing SCASM on it would be pretty neat. At least for locomotion behavior, this also has the possibility of doing away with a lot of the position-control difficulties we have had with braided pneumatic actuator robots. You want to put muscle-like actuators to good use? SCASM represents one of the ways it is actually and successfully done.

9.8.4 SCASM for Higher-Level or Synthetic Behaviors

Higher-level cyclic behaviors executed in interaction with the environment might also benefit from the use of SCASM. As mentioned above, this is already being done for odor tracking. One could even represent behaviors such as driving on a road that has stop signs and stop lights as a SCASM system, though in order for this to be useful it is probably

necessary to be able to break the behavior's high-level "virtual actuators" into control by more than one module. An intermediate-level behavior that occurs to me here is skiing a slalom course; one might break this down into three modules; one controlling body twist, another controlling the amount of crouch and ski roll, and the third controlling arm position. Candidates for important sensory events here include ground contact force thresholds, contact of the forearm with a flag, and the next flag passing a threshold bearing in the visual field. This is actually a pretty physically complicated behavior, but it could be fun to really try applying a SCASM description to it.

SCASM might also be applied to synthetic behaviors such as pick-and-place. In this example the selection of proper sensors and sensory events is critical. Once that is done, this might be a good way to generate fast, lightweight and reliable control for a number of industrial robot applications.

10 Appendices

10.1 NeuRoMod Engine Code Documentation

Romod Engine Code Overview

Brandon Rutter
1/4/10 12:44 AM

Table of Contents

NeuRoMod Engine Documentation

Print-Edited Introduction

This document, other than this paragraph, is a subset of the Doxygen-generated documentation for the NeuRoMod engine collection of programs and libraries. Most serious users will find the complete HTML version of this documentation much more useful; it has been maintained as part of the doc-romod section of the consolation code repository: svn://consolation.cwru.edu/releg/doc-romod/trunk/engine/html . What is included here is the main introduction page, full Todo and bug lists, and full indexes of data structures and files. Only the brief data structure documentation is included; there is further detailed documentation available for many of the struct elements. Brief documentation for a few example files is also included. It should be noted that a great deal of the actual function of these programs is contained in the library functions defined in the other files; particularly utility, userio, HDSO, HDSOio, parsebool, boolio, and the HDSO Boolean parsing files. Including the detailed documentation for all of these files brings this document up to 180 pages, which is just killing too many trees given that it is available in digital browseable HTML form.

What follows is not beautiful; I am not a Doxygen wiz, and there are a number of poorly formatted or incorrectly tagged items. It should serve, however, to give a faster basic introduction to what exists in the code than the code itself.

Introduction

This set of pages attempts to document the "engine" portion of the "romod" project, an implementation of SCASM control for robots and robotic models. The "engine" means the underlying code which directly controls the robot, as described in the SCILLS system description.

There are a lot of source files here, not all of which will make sense at first, if at all. I will attempt to provide a basic description of the main and mostly interesting programs here, however. Of note is that these programs and threads communicate with one another both through shared memory and real-time FIFO. In general, the shared memory has to be initialized in a sane matter from user space before starting the realtime module which depends on said shared memory to operate.

Related Projects

- Romod/SCASM analysis toolbox
- SCCILS (Actual link is available on that page; direct linking somehow doesn't work).
- Python SCCILS GUI

Engine Module: Enginemod.c

enginemod.c is the most basic set of realtime functionality for running a SCASM robot, consisting of several threads which handle state transitions, activation and muscle modeling, and

low-level motor commands. The modularity of these threads should be made better- really we should be able to do any of these things from a completely different module, and it isn't currently well suported.

Engine Initializer: Engineprog.c

engineprog.c is where the initialization mentioned above happens for enginemod.c. In fact, this program is used both to initialize and load the module, as well as to unload it.

Engine Command Line Interface: Enginecli.c

It isn't much of an interface, but enginecli.c is a command line tool for running the robot as-is, or running an HDSOscript. It also handles some of the log file post-processing.

Data Logger and Monitor: Monitor.c

monitor.c is a user-space program that handles tempoarary logging of the data streams coming in on the RT-FIFOs; needs to be started at a VERY high priority (usually nice -19 or so) by whoever wants it to do logging. Enginecli starts and stops monitor, for instance.

Script Handling Module: Scriptmod.c

scriptmod.c handles timed changes to HDSO data stored in shared memory, via hdsoscript-styled entries. It doesn't have to be an actual script; it can just be one-by-one changes as long as script handling is enabled.

Todo List

Global aimotor::offset
specify maximum offset, or maximum offset for each speed, for safety

Global cdata_msg_struct::data [20]
re-kluged pending translator parsing fix [MAXDATAELTS];

Global contr_joint::basestate
change this so it doesn't require a particular state order

Global fdata_msg_struct::data [20]
re-kluged pending translator parsing fix [MAXDATAELTS];

Global muscle::lo
put better geometry stuff here, as in "actuators" for llc
make it 3D resting length of active section.

Global sensor::correct_overflow
Make a specification of the input sources, instead of doing that all in code. flag indicating whether to do overflow correction

Global FIFO_ENUM
Actually modify monitor et al. so this works!

Global main
figure out which of the following actually needs to be shared/kept
make workingdirname an absolute path; more portable?

File enginecli.c
check for failure of system calls to other programs

File enginecli.c
reduce code duplication between this and datalogging.c

Global main
figure out which of the following actually needs to be shared/kept
make workingdirname an absolute path; more portable?

Global main
format this better, and get the year/century to show up properly

Global init_module
make this depend on the defines and initializers in control.h

Global motor_action
THE CONTROL LOOP DELAY WILL VARY WITH NUMBER OF MOTORS. The comm delay is long enough this is probably insignificant, but really it should be fixed.

Global motor_action
merge this with clock_gettime below

Global motor_action
update the following to report robot_dead or something under normal runtime environment

Global motor_conversion
unify the signal flow to be input->processing->overflow comp->assignment

Global motor_conversion
put bounds checking here!

Global muscle_conversion
only do this if the joint is in muscle mode!

Global state_transition
make this configurable on-the-fly

File engineprog.c
update this for new structure

Global main
move this initializer to a DEFINE somewhere
make this so that it restores the serial port charactaristics on unload
(in a perfect world) make it so we can actually have non-rt serial devices as well

Global main
make this so it will work on a system with any number of serial ports

Global main
make this as clean as the other file parsing

Global main
put this somewhere else?

Global main
fix this horrible kluge for cockroach leg!!

Global hdso_get
make HDSO_INVALID_IDNUM also set errno,or return -EINVAL, but that would need to be caught by all the callers!

Global hdso_write_bool
Make a name-based fancy output function

Global hdso_parse_bool
make the self-reference checking deeper

Global hdso_init_data
also get this to use HDSOmods in a document?

Global hdso_parse_datafield
weed out unnecessary booleans below

Global hdso_parse_datafield
Do something more intelligent with retval here

Global hdso_parse_datafield
make the following handle both numerical and string char data

Global hdso_write_datafield
update hdso_parse_datafield, and make this work more intelligently with it.

Global StrToDataType
make this actually use the enum type properly

Global main
make there be a way to quit monitor in 'i' mode - now done with datalogging program?

Global main
some error checking here, and for hdsolog

Global rt_write_bool
update this description

Global command_handler
decide whether to also flush the HDSO data fifo when logging is stopped.

Global HDSOserver_code
change this behavior, ignoring engine running (requres a reworking of time tracking) and dumping requests if HDSOserver is not enabled. (?)

Global HDSOserver_code
double-check the following line?

Global HDSOserver_code
make hdso_GetDataSize faster; initialize a static data size member of the hdso struct

Global init_module
have hdso_maxdatasize automatically handled and updated when inserting new HDSOs

Global script_handler
put the "if enabled" crap here, or determine if this todo is stale

Global script_handler_code
handle kill-type signals during clock_nanosleep above, rather than doing this second check on script_enable

Global script_handler_code
somehow indicate whether an error occurred, and what actually happened (which is what is recorded on these FIFOS) is NOT what was desired.

Global getinteger
Say who Robert is.

Global InvParseScriptXML
put the DTD information in a DEFINE in hdso(io).h

Global InvParseScriptXML
maybe set this to "DataScript" instead?

Global InvParseScriptXML
make the description more specific; i.e. different between ScriptLog and HDSOlog ?

Global InvParseScriptXML
instead of having this here, make an "hdso_printdata" function; start with the relevant section of hdso_print_general

Global ParseJointFile
Add more contextual information to the debug error messages.

Global ParseJointFile
actually put the senseory organs in (do we care?)

Global ParseJoints
Add more contextual information to the debug error messages.

Global ParseJoints
better bounds/input checking?

Global ParseJoints
check to make sure the start state is actually one of the contained states

Global ParseMuscles
make this so it doesn't require a specific order for the usages, and provides default null muscle model behavior

Global ParseScriptXML
make this warning work for xml-translated abstract types

Global ParseSensOrgans
Add overflow checking for string arguments!!

Global ParseStates
clean up the repeated retcodes!

Global ParseStates
the indexing here?

Global ParseStates
handle more than one HDSO parameter?

Global ParseStates
check to see that these correspond to either sensors or params here defined

Global ParseStates
roll this proccessing into the standard initialization functions

Global ParseStates
put the following in a function?

Global ParseStates
figure out why the heck the following lines cause "double free or corruption" errors

Global PrintJoints
better-ify sense organ handling (?)

Global PrintJoints
fix the misnomer? also prints aimotors

Global PrintStates
print more names & values, fewer indices

Global safeOpenWrite
update this description

Global StrToActFcnType
make this actually use the enum type properly

Global StrToFcnType
make this actually use the enum type properly

Global StrToSensorType
make this actually use the enum type properly

Global ToUpper
Make this behave properly for strings shorter than length.

Global const_activation
investigate wether all these lookups are too slow?

Global generic_function
document this

Global generic_function
set errno instead;

Global linear_activation
switch this to using RTValuePTR?

Global rt_memcpy
fix or remove StateToBool

Global RADFACT
make this dependent on compile environment

Bug List

Global state_transition
allow the following to handle other than two muscles per joint

Global HDSOserver_code
?maybe a bug? In the following usage, it is possible that the data written to HDSO_DATA_FIFO and HDSO_DATA2_FIFO would differ, even though the timestamps provided to those FIFOS are the same. Putting the data in a temporary holding place may be more strictly correct, but would also take more processing time.

Global getinteger
The code for this sscanf line says still buggy, but I don't know what the bug is.

Data Structure Index

Data Structures

Here are the data structures with brief descriptions:

File Index

File List

Here is a list of all files with brief descriptions:

Data Structure Documentation

aimotor Struct Reference

Struct for describing a aimotor servo motor.

`#include <aimotor.h>`

Data Fields

- int idNum
  idNum for xml/UI use
- volatile unsigned char enable
  1 (enable) or 0 (disable) force control/data collection
- unsigned char id
  motor id, 0-30
- unsigned char mode
  position or "force" control
- char dir
  1 (positive posn.=flexed) or -1 (" = extended)
- volatile unsigned char speed
  speed of motor position command, 0-4
- volatile char offset
- volatile unsigned char commandpos
  commanded position (0-254)
- unsigned char old_commandpos
  for keeping track of current dir
- unsigned char position
  latest position data
- unsigned char old_position
  for keeping track of velocity
- float velocity
  filtered velocity data
- unsigned char cur
  latest current data (unsigned)

- float filtcur
  filtered, directionalized current
- float calibration [2]
  *angle calibration; raw = [0] + [1]*angle_in_degrees*
- int numsensors
  number of associated sensors
- int sensorIDs [AIMOTOR_MAX_SENSORS]
  HDSO/sensor ids of associated sensors.
- int sensorindices [AIMOTOR_MAX_SENSORS]
  associated indices into the sensor array
- int sensorTypes [AIMOTOR_MAX_SENSORS]
  type of sensory data to be set

Detailed Description

Struct for describing a aimotor servo motor.

This includes current state, mode, and communication info

Definition at line 41 of file aimotor.h.

---

cdata_msg_struct Struct Reference

Struct for carrying char data from a task via FIFO.

`#include <control.h>`

Data Fields

- hrtime_t timestamp
  timestamp usually represents when the data was gathered, in nanoseconds
- int task
  task is one of the defined integer task id's, e.g. AD_TASK
- unsigned char data [20]
  data is an array of char data, the meaning of which is usually implicitly defined by the task it comes from.

Detailed Description

Struct for carrying char data from a task via FIFO.

The data message struct is used for packaging data, usually for the purpose of logging to a user-space program which would write the data to a log file

Definition at line 222 of file control.h.

---

command_msg_struct Struct Reference

Struct for carrying command information via FIFO.

`#include <control.h>`

Data Fields

- int command
  command is one of the defined integer messages, e.g. START_TASK
- int values [5]
  The contents of "values" are parsed according to what "command" is.

Detailed Description

Struct for carrying command information via FIFO.

The command message struct is used for packaging command information for sending commands from one place to another (only for sending commands to LLCmodule tasks).

Definition at line 162 of file control.h.

---

contr_joint Struct Reference

Struct for joint control.

`#include <legdef.h>`

Data Fields

- volatile char name [LEGDEF_NAMELENGTH]
  A char name for UI usefulness.
- volatile int idNum
- volatile unsigned char statetrans_enable
  enable state transition processing
- volatile unsigned char activation_enable
  enable activation function processing
- volatile unsigned char numstates
  Number of possible states for the joint.
- volatile unsigned basestate
- volatile int curstate
  Index into current joint state in state array (negative means invalid).
- volatile unsigned char numsenseorg
  Number of sensory organs associated with this joint.
- volatile unsigned senseorgans [10]
  Indices into sensory organ array.
- volatile unsigned char nummuscles
- volatile unsigned muscles [MAXMUSCLES]
  Indices into the muscle array.

Detailed Description

Struct for joint control.

In practice most state and sensor data should be accessed via a joint, at least in the controller.

nubmer stored in contr_jointnum; array stored in contr_joints;

Definition at line 25 of file legdef.h.

---

contr_state Struct Reference

Struct for describing control state transitions and activations.

`#include <legdef.h>`

Data Fields

- char name [LEGDEF_NAMELENGTH]
- int idNum
- hrtime_t starttime
  runtime variable for storing start time of the active state
- int hdso_dwelltime
- unsigned char nummuscles
  number of muscles (later we could combine muscle forces at a joint more flexibly)
- volatile unsigned int muscles [MAXMUSCLES]
  Indices into the muscle array.
- int activation [MAXMUSCLES]
  activation function which sets muscle activation levels for given muscle
- unsigned char numasens [MAXMUSCLES]
  sensors needed for the activation functions
- int hdsoasensors [MAXMUSCLES][10]
- unsigned char numaparams [MAXMUSCLES]
  parameters for activation function
- int hdsoaparams [MAXMUSCLES][10]
- int transitionrule
  HDSO idNum of the boolstring transition rule.

Detailed Description

Struct for describing control state transitions and activations.

The contr_state struct consists of essentially four things: 1) a name for the state 2) a pointer containing the function describing the transition to the state 3) a pointer containing the activation function for this state 4) the sensors needed for the transition/activaiton functions

number stored in contr_statenum; array stored in contr_states

Definition at line 105 of file legdef.h.

---

fdata_msg_struct Struct Reference

Struct for carrying float data from a task via FIFO.

`#include <control.h>`

Data Fields

- hrtime_t timestamp
  timestamp usually represents when the data was gathered, in nanoseconds
- int task
  task is one of the defined integer task id's, e.g. AD_TASK
- float data [20]
  data is an array of float data, the meaning of which is usually implicitly defined by the task it comes from.

Detailed Description

Struct for carrying float data from a task via FIFO.

The data message struct is used for packaging data, usually for the purpose of logging to a user-space program which would write the data to a log file

Definition at line 207 of file control.h.

HDSO Struct Reference

`#include <hdso.h>`

Data Fields

- unsigned int idNum
- char name [HDSO_MAXNAMELEN]
- unsigned char abstractType
- char type
- char xmltype
- int numdims
- int dimensions [HDSO_MAXDIMS]
- size_t data
  offset of the data, relative to base of data storage block

Detailed Description

Definition at line 23 of file hdso.h.

---

info_msg_struct Struct Reference

Struct for carrying info on task operation via FIFO.

`#include <control.h>`

Data Fields

- hrtime_t timestamp
  timestamp is usually that; system time in nanoseconds
- int task

task is one of the defined integer task id's, e.g. AD_TASK

- int msg
  msg is an integer representing a status message, e.g. START_TASK, which in this case would usually be interpreted as "This task has started"

Detailed Description

Struct for carrying info on task operation via FIFO.

The info message struct is used for packaging information for sending operation data, like a command receipt acknowledgement, from one task to another.

The original usage intention was for just that, and that is the usage on info_fifo and debug_fifo, but info_msg_struct is used for a number of other things now, including:

1) script_fifo & scriptlog_fifo; here, .timestamp is the time in ns since the start of the engine to implement the change, .task is the HDSO ID, and .msg is the number of bytes to be written. Following each such formatted info_msg_struct should be data equal to the number of bytes specified. In the case of scriptlog_fifo, the .timestamp and .msg indicate the actual time and number of bytes written.

2) hdso_request_fifo & hdso_data_fifo are similar to script, though .timestamp is irrelevant in the hdso_request_fifo and .msg is the number of bytes to be read, which need not equal the entire data for that HDSO.

Definition at line 191 of file control.h.

---

muscle Struct Reference

Struct for containing muscle data, incl. muscle model & geometry.

`#include <legdef.h>`

Data Fields

- char name [LEGDEF_NAMELENGTH]
- int idNum
- float lo
- float r
  *insertion radius, defined s.t. force*r gives positive torqe for stance muscles*
- float alpha
  neutral angle, in degrees
- int functype [NUM_MUSCLE_FUNCTIONS]
  generic-function-types for the various muscle model parameters
- int hdsoID [NUM_MUSCLE_FUNCTIONS]
  the HDSO parameter IDs for each of the functions
- float * rtparams [NUM_MUSCLE_FUNCTIONS]
  pointers to the actual data storage for use in RT-space. Necessary?
- float fmax
  Type of passive stiffness model.
- float current_force

force currently being exerted by the muscle

- float current_length
  current length of the muscle active section
- float current_velocity
  current velocity of the muscle contraction
- volatile float activation
  activation level for this muscle

Detailed Description

Struct for containing muscle data, incl. muscle model & geometry.

The muscle structure contains data describing the muscle model used for the associated muscle, as well as (hopefully) the appropriate joint geometry.

Definition at line 54 of file legdef.h.

---

senseorgan Struct Reference

Struct for describing a "Biological" sense organ.

`#include <legdef.h>`

Data Fields

- char name [LEGDEF_NAMELENGTH]
- int idNum
- unsigned numsensors
  number of actual sensors that have something to do with this organ
- unsigned sensors [10]
  array of indices into these sensory signals

Detailed Description

Struct for describing a "Biological" sense organ.

The purpose of this struct is primarily to clarify the relationships between biological sense organs and the actual sensory signals used in the controller.

number stored in sensorgannum; array in sensorgans

Definition at line 147 of file legdef.h.

---

sensor Struct Reference

Struct for describing & carrying data for a sensory singal.

`#include <legdef.h>`

Data Fields

- char name [LEGDEF_NAMELENGTH]

- unsigned numorgans
  number of sense organs associated with this sensor
- unsigned senseorgans [10]
  array of indices to the sense organs
- volatile float * RTValuePtr
  RT-context-valid direct pointer to value. NOT FOR USER SPACE!
- volatile int valueID
  ID of the HDSO actually storing the data, AND the ID identifier for THIS sensor;.
- unsigned char correct_overflow
- unsigned char overflow_initialized
- unsigned char overflows
- int oldinput
- int input
- int limits [2]
  limits of input data for overflow correction [low high]

Detailed Description

Struct for describing & carrying data for a sensory singal.

This struct carries data for a sensory signal, and describes which sensory organs it has something to do with.

number stored in sensornum; array in sensors

Definition at line 164 of file legdef.h.

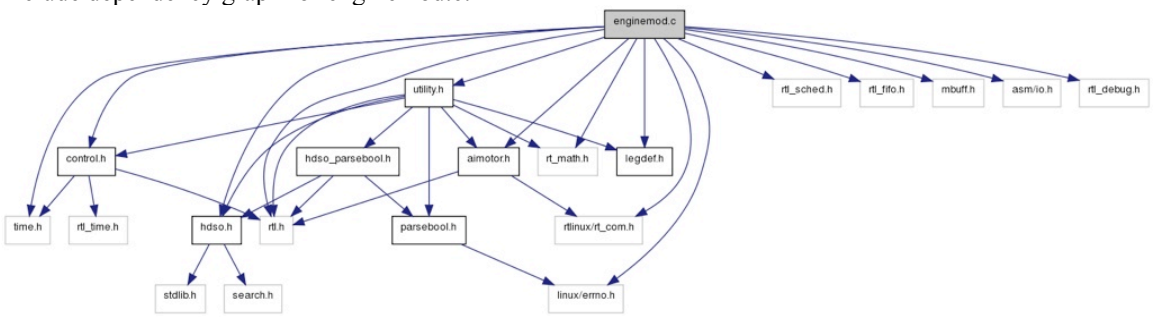
File Documentation

enginemod.c File Reference

Kernel-space realtime module for basic SASM robot control.

```
#include <linux/errno.h>
#include <rtl.h>
#include <time.h>
#include <rt_math.h>
#include <rtl_sched.h>
#include <rtl_fifo.h>
#include <mbuff.h>
#include <asm/io.h>
#include <rtl_debug.h>
#include <rtlinux/rt_com.h>
#include "legdef.h"
#include "control.h"
#include "aimotor.h"
#include "utility.h"
#include "hdso.h"
```

Include dependency graph for enginemod.c:



Defines

- #define ENGINEMOD_TASKNO 4
- #define DEBUG 2

Functions

- float actfunctions (int, sensor *, contr_state *, contr_joint *, HDSO *, unsigned int, void *)
- void * motor_action (void *arg)
- void * motor_conversion (void *arg)
- void * muscle_conversion (void *arg)
- void * state_transition (void *arg)
- int my_handler (unsigned int fifo)
- int init_module (void)
- void cleanup_module (void)

Variables

- pthread_t tasks [ENGINEMOD_TASKNO]
- int tasks_enable [ENGINEMOD_TASKNO]
- int conversions_waiting = 0
- int muscle_waiting = 0
- unsigned int * contr_jointnum
- unsigned int * contr_statenum
- unsigned int * sensorgannum
- unsigned int * sensornum
- unsigned int * musclenum
- unsigned int * hdsonum
- size_t * hdsodatasize
- hrtime_t * enginestarttime
- volatile char * cmode
- volatile unsigned char * puppetdone
- unsigned char * puppetnum
- unsigned char * puppetjoints
- int * puppetlines
- hrtime_t * puppettimes
- float * puppetdata
- volatile int * motor_speed
  Pointer to the HDSO-enclosed motor speed variable.
- volatile long long int * motor_control_delay_ns
  Pointer to the HDSO-enclosed motor control loop delay parameter.

- volatile long long int * statetrans_period_ns
  Pointer to the HDSO-enclosed state transition loop period.
- volatile aimotor * motors
- volatile contr_joint * contr_joints
- volatile contr_state * contr_states
- volatile senseorgan * sensorgans
- volatile sensor * sensors
- volatile muscle * muscles
- volatile HDSO * hdsos
- volatile void * hdsodata

Detailed Description

Kernel-space realtime module for basic SASM robot control.

Programmer: Brandon Rutter

This module containes threads which run low-level servomotor controls as well as some for data processing and state transition control in a reflex-based leg controller. This is the realtime part of the "engine" for this reflex-based robot model of neuromechanical systems.

The module assumes that rt_math.o is loaded, to provide rt-safe math library symbols.

It is also supposed that certain shared memory values have already been set when init_modules is called.

Threads are all initiated, enabled, disabled, and destroyed through commands sent from another process to the command FIFO.

If DEBUG is set, additional data are (supposedly) sent from each thread to a data logging FIFO.

Definition in file enginemod.c.

Define Documentation

#define DEBUG 2

Definition at line 42 of file enginemod.c.

#define ENGINEMOD_TASKNO 4

Definition at line 22 of file enginemod.c.

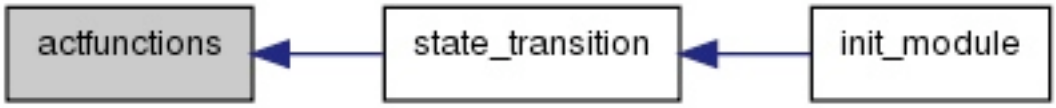
Referenced by motor_action().

Function Documentation

float actfunctions (int, sensor *, contr_state *, contr_joint *, HDSO *, unsigned *int*, void *)

Referenced by state_transition().

Here is the caller graph for this function:

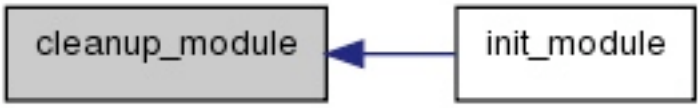


void cleanup_module (void)

Definition at line 1093 of file enginemod.c.

Referenced by init_module().

Here is the caller graph for this function:



int init_module (void)

Todo:

make this depend on the defines and initializers in control.h

Definition at line 872 of file enginemod.c.

void* motor_action (void * *arg*)

< I hate hard-coding this, but it crashes otherwise

Todo:

THE CONTROL LOOP DELAY WILL VARY WITH NUMBER OF MOTORS. The comm delay is long enough this is probably insignificant, but really it should be fixed.

Todo:

merge this with clock_gettime below

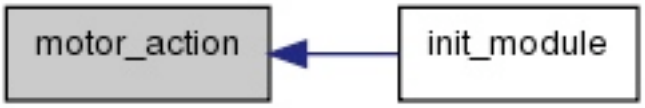
Todo:

update the following to report robot_dead or something under normal runtime environment

Definition at line 100 of file enginemod.c.

Referenced by init_module().

Here is the caller graph for this function:



void* motor_conversion (void * *arg*)

Todo:

unify the signal flow to be input->processing->overflow comp->assignment

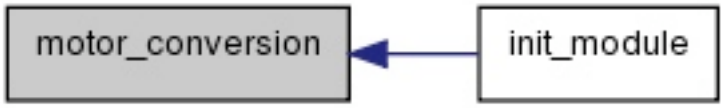
Todo:

put bounds checking here!

Definition at line 229 of file enginemod.c.

Referenced by init_module().

Here is the caller graph for this function:



void* muscle_conversion (void * *arg*)

This thread is another "conversion daemon" which converts muscle activations to motor commands, when kicked by the state transition/activation setting task.

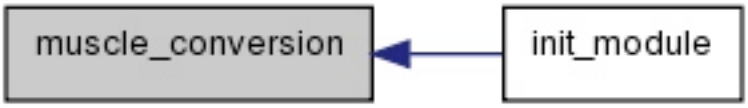
Todo:

only do this if the joint is in muscle mode!

Definition at line 432 of file enginemod.c.

Referenced by init_module().

Here is the caller graph for this function:



int my_handler (unsigned int *fifo*)

Definition at line 719 of file enginemod.c.

Referenced by init_module().

Here is the caller graph for this function:

void* state_transition (void * *arg*)

This thread keeps track of the state transitions for each joint, then sets the corresponding muscle activations. Should be a periodic thread.

set to execute periodically Todo:

make this configurable on-the-fly

todo consolidate gethrtime calls;

todo error checking here?

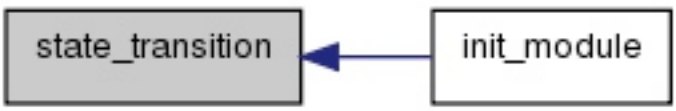
Bug:

allow the following to handle other than two muscles per joint

Definition at line 578 of file enginemod.c.

Referenced by init_module().

Here is the caller graph for this function:

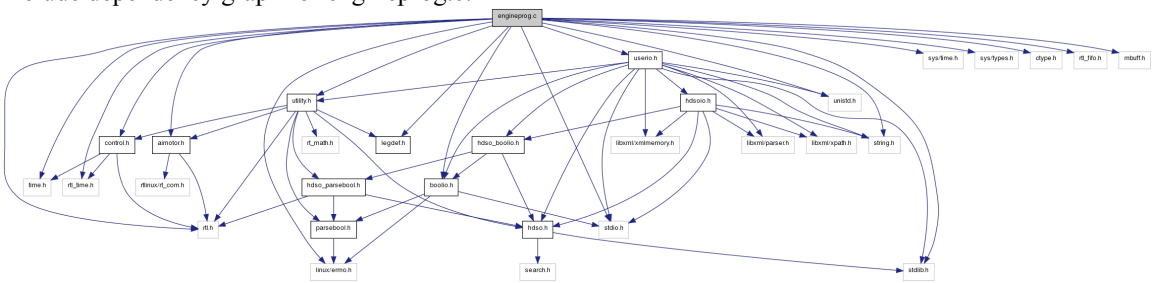


engineprog.c File Reference

User-space program for loading/unloading configuration files and the enginemod realtime kernel module.

```
#include <linux/errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <ctype.h>
#include <string.h>
#include <rtl.h>
#include <rtl_time.h>
#include <rtl_fifo.h>
#include <mbuff.h>
#include "control.h"
#include "userio.h"
#include "utility.h"
#include "aimotor.h"
#include "legdef.h"
#include "boolio.h"
```

Include dependency graph for engineprog.c:



Defines

- #define LOAD 1
- #define UNLOAD 2
- #define DEBUG 2

Functions

- void usage (void)
  Prints basic usage/argument information.
- void help (void)
  Just prints more detailed usage information.
- int main (int argc, char **argv)

Detailed Description

User-space program for loading/unloading configuration files and the enginemod realtime kernel module.

User-level engine program; primary usage is loading configuration files and initializing the RT portion of the engine, and the unloading of the RT engine when called with that argument. Returns zero for no errors, number of errors for non-fatal errors, or a negative number if a fatal error occurred.

Usage: ./engineprog [V*|Q*]L [config files] or ./engineprog [V*|Q*]U or enter "./engineprog help" for more info.

Code Overview

The main function goes approximately like this: Convert input to all upper case for easier processing. Set verbosity, call help and exit if appropriate Set load or unload action.

If load: set the serial uarts ttys0 and ttys1 to "none" to allow rt_com to load start rtlinux Verify that the configuration files are present and parseable, read in data sizes. If a file can't be opened; set fatal error number and exit. allocate mbuff size variables for shared config/parameter storage initialize mbuff size variables using data from config files allocate shared config/parameter storage based on data now in size variables Do additional inital parsing of HDSOlists (parameters.xml and sensors.xml) determine size of HDSO storage block using HDSO size data. Allocate and initialize the HDSO data storage from the HDSOlist files. Verify that global HDSO engine parameters exist and are sensible. Parse the joints file, finishing the population of configuration data. Depending on verbosity, print out various levels of now-loaded config data: verbosity > 1 : print joint, muscle, state, and sensor configuration verbosity > 2 : also print all parameter and sensor HDSOs with data. Insert enginemod.o report probable success detach from shared memory

If unload: call "rtlinux stop enginemod"

then return the "errors" variable

Todo:

update this for new structure

Definition in file engineprog.c.

Define Documentation

#define DEBUG 2

Definition at line 91 of file engineprog.c.

#define LOAD 1

Definition at line 88 of file engineprog.c.

Referenced by main(), motor_conversion(), and ParseJoints().

#define UNLOAD 2

Definition at line 89 of file engineprog.c.

Referenced by main().

Function Documentation

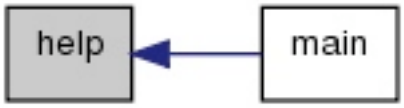
void help (void)

Just prints more detailed usage information.

Definition at line 727 of file engineprog.c.

Referenced by main().

Here is the caller graph for this function:



int main (int *argc*, char ** *argv*)

Todo:
move this initializer to a DEFINE somewhere
make this so that it restores the serial port charactaristics on unload
(in a perfect world) make it so we can actually have non-rt serial devices as well

<number of joints

<number of joint states

<number "biological" sense organs

<number of sensory signals

<number of virtual/real muscles

<number of HDSO parameters

<size of HDSO parameter data storage (in bytes)

<number of total HDSOs (parameters+sensors)

<size of cumulative HDSO data storage

<size of HDSO sensor data storage in bytes

Todo:
make this so it will work on a system with any number of serial ports

Todo:
make this as clean as the other file parsing

Todo:
put this somewhere else?

Todo:
fix this horrible kluge for cockroach leg!!

Definition at line 93 of file engineprog.c.

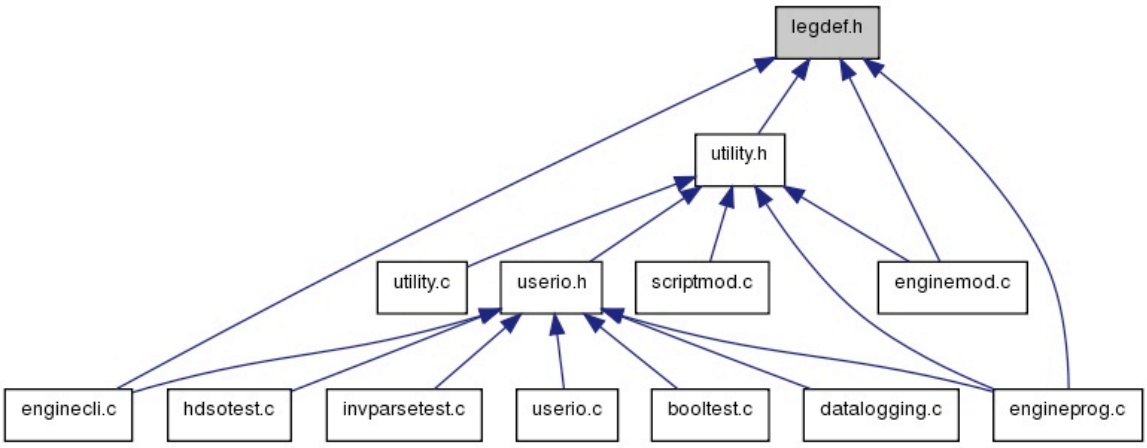
void usage (void)

Prints basic usage/argument information.

---

legdef.h File Reference

This graph shows which files directly or indirectly include this file:



Data Structures

- struct contr_joint
  Struct for joint control.
- struct muscle
  Struct for containing muscle data, incl. muscle model & geometry.
- struct contr_state
  Struct for describing control state transitions and activations.
- struct senseorgan
  Struct for describing a "Biological" sense organ.
- struct sensor
  Struct for describing & carrying data for a sensory singal.

Defines

- #define LEGDEF_NAMELENGTH 128
- #define MAXMUSCLES 4
- #define NUM_MUSCLE_FUNCTIONS 3
- #define MUSCLE_FUNCTION_NAME_INITIALIZER
  [NUM_MUSCLE_FUNCTIONS][MAXNAMELEN]={"passive_stiffness", "force-length", "force-velocity"}

Enumerations

- enum MUSCLE_FUNCTION_ENUM { PASSIVE_STIFFNESS = 0, FORCE_LENGTH, FORCE_VELOCITY }
  function uses and enumerations for muscle models

Detailed Description

This file should contain the structure definitions and function prototypes necessary for the definition of reflex-controlled legs in a biology- friendly manner.

Definition in file legdef.h.

Define Documentation

#define LEGDEF_NAMELENGTH 128

Definition at line 10 of file legdef.h.

Referenced by ParseJoints(), ParseMuscles(), ParseSensOrgans(), ParseSensors(), ParseSensorsOld(), and ParseStates().

#define MAXMUSCLES 4

Definition at line 11 of file legdef.h.

Referenced by ParseStates().

#define MUSCLE_FUNCTION_NAME_INITIALIZER [NUM_MUSCLE_FUNCTIONS][MAXNAMELEN]={"passive_stiffness", "force-length", "force-velocity"}

Definition at line 16 of file legdef.h.

#define NUM_MUSCLE_FUNCTIONS 3

Definition at line 15 of file legdef.h.

Referenced by ParseMuscles().

Enumeration Type Documentation

enum MUSCLE_FUNCTION_ENUM

function uses and enumerations for muscle models

Enumerator:

PASSIVE_STIFFNESS

FORCE_LENGTH
FORCE_VELOCITY

Definition at line 14 of file legdef.h.

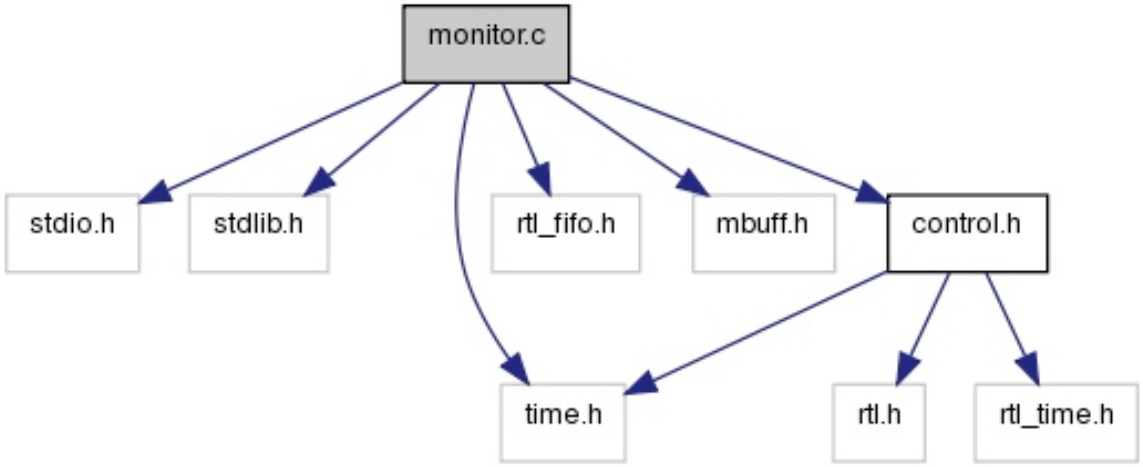
---

monitor.c File Reference

Monitors/logs data and information from realtime processes.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <rtl_fifo.h>
#include <mbuff.h>
#include "control.h"
```

Include dependency graph for monitor.c:



Defines

- #define DEBUG 1

Functions

- int main (int argc, char **argv)

Detailed Description

Monitors/logs data and information from realtime processes.

A program to monitor (to screen) and log (to files) data from rt processes, while a seaparate program may provide user interaction.

Definition in file monitor.c.

Define Documentation

#define DEBUG 1

Definition at line 20 of file monitor.c.

Function Documentation

int main (int *argc*, char ** *argv*)

Todo:
make there be a way to quit monitor in 'i' mode - now done with datalogging program?

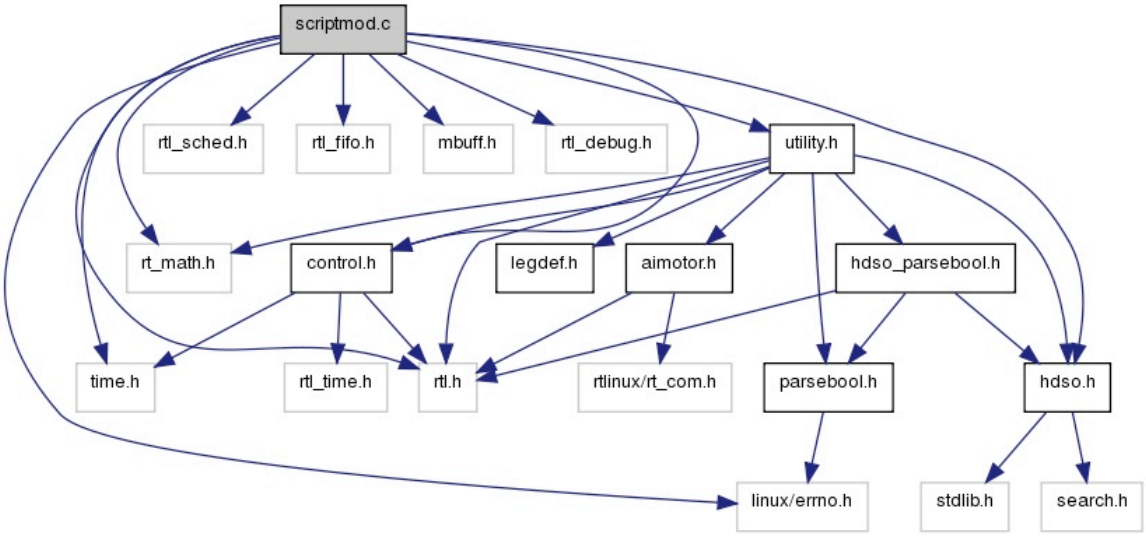
Todo:
some error checking here, and for hdsolog

Definition at line 22 of file monitor.c.

scriptmod.c File Reference

```
#include <linux/errno.h>
#include <rtl.h>
#include <time.h>
#include <rt_math.h>
#include <rtl_sched.h>
#include <rtl_fifo.h>
#include <mbuff.h>
#include <rtl_debug.h>
#include "control.h"
#include "utility.h"
#include "hdso.h"
```

Include dependency graph for scriptmod.c:



Defines

- #define DEBUG 1

Functions

- void cleanup_module (void)
- int command_handler (unsigned int fifo)
  FIFO handler for script_command FIFO.
- int script_handler (unsigned int fifo)
- void * script_handler_code (void *arg)
- int HDSOrequest_handler (unsigned int fifo)
- void * HDSOserver_code (void *arg)
- int init_module (void)

Variables

- pthread_t script_task
- pthread_t HDSOserver_task
- int script_enable
- int HDSOserver_enable
- int HDSOserver_done
- unsigned int * hdsonum
  number of HDSO parameters/sensors/etc.
- unsigned int * hdsodatasize
  size of cumulative HDSO data storage, in bytes
- size_t * hdso_maxdatasize
  maximum size of data "in" any single HDSO
- volatile hrtime_t * enginestarttime
  time of engine start, for scripting and other purposes
- volatile unsigned char * scriptdone
- volatile HDSO * hdsos
- volatile void * hdsodata

- volatile char * scriptmod_status
- void * scriptdata
- void * hdso_reqdata

Detailed Description

This module modifies the data in an HDSO data store at specified times, relative to a shared memory start time. Mostly intended for scripting purposes.

It is supposed that certain shared memory values (number of HDSO entries and size of data store) have already been set when init_module is called.

The rt_math module needs to be loaded in order for this to run correctly; needed for some of the hrtime_t mathematics.

Definition in file scriptmod.c.

Define Documentation

#define DEBUG 1

Definition at line 34 of file scriptmod.c.

Function Documentation

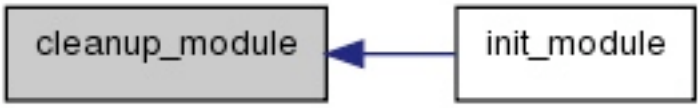
void cleanup_module (void)

free all allocated resources

Definition at line 1093 of file enginemod.c.

Referenced by init_module().

Here is the caller graph for this function:



int command_handler (unsigned int *fifo*)

FIFO handler for script_command FIFO.

Currently just takes commands to enable or disable script handling. Sending LOG_DATA or STOP_LOGGING to this task turn the HDSO data server on or off, respectively. Later implementations may use command message parameters to actually turn some true logging on or off as well.

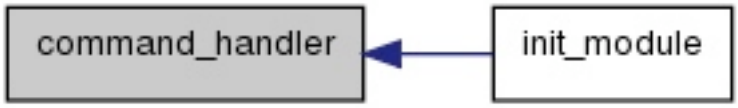
Todo:

decide whether to also flush the HDSO data fifo when logging is stopped.

Definition at line 71 of file scriptmod.c.

Referenced by init_module().

Here is the caller graph for this function:



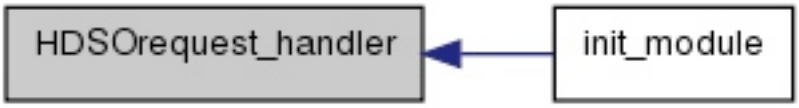
int HDSOrequest_handler (unsigned int *fifo*)

yup, this is pretty lame. Done this way just so we can give this task a priority

Definition at line 322 of file scriptmod.c.

Referenced by init_module().

Here is the caller graph for this function:



void* HDSOserver_code (void * *arg*)

This takes info_msg_structs formatted similarly to those for the script handler, interpreting them as requests for data rather than instructions on what to modify. .timestamp is irrelevant, .task is the HDSO ID, and .msg is the number of bytes to be read (i.e. written to HDSO_DATA_FIFO).

Todo:

change this behavior, ignoring engine running (requres a reworking of time tracking) and dumping requests if HDSOserver is not enabled. (?)

Todo:

double-check the following line?

Todo:

make hdso_GetDataSize faster; initialize a static data size member of the hdso struct

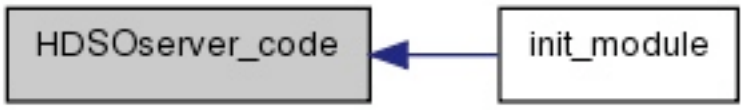
Bug:

?maybe a bug? In the following usage, it is possible that the data written to HDSO_DATA_FIFO and HDSO_DATA2_FIFO would differ, even though the timestamps provided to those FIFOS are the same. Putting the data in a temporary holding place may be more strictly correct, but would also take more processing time.

Definition at line 342 of file scriptmod.c.

Referenced by init_module().

Here is the caller graph for this function:



int init_module (void)

Todo:

have hdso_maxdatasize automatically handled and updated when inserting new HDSOs

Definition at line 510 of file scriptmod.c.

int script_handler (unsigned int *fifo*)

yup, this is pretty lame. Done this way just so we can give the scripting a priority

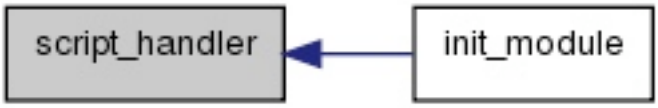
Todo:

put the "if enabled" crap here, or determine if this todo is stale

Definition at line 157 of file scriptmod.c.

Referenced by init_module().

Here is the caller graph for this function:



void* script_handler_code (void * *arg*)

This takes info_msg_structs and uses them to modify data in the "hdsos" HDSO data store. in the struct, .timestamp is the time in ns since the start of the engine to implement the change, .task is the HDSO ID, and .msg is the number of bytes to be written. Following each such formatted info_msg_struct should be data equal to the number of bytes specified.

Todo:

handle kill-type signals during clock_nanosleep above, rather than doing this second check on script_enable

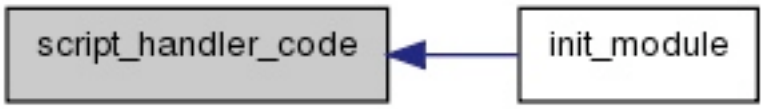
Todo:

somehow indicate whether an error occurred, and what actually happened (which is what is recorded on these FIFOS) is NOT what was desired.

Definition at line 179 of file scriptmod.c.

Referenced by init_module().

Here is the caller graph for this function:



10.2 SCCILS 0.1

The SCASM Command and Configuration Interface Language Specification

version 0.1

BRANDON LEWIS RUTTER

in Collaboration With

John Bender and Marcus Blümel

Department of Mechanical and Aerospace Engineering

CASE WESTERN RESERVE UNIVERSITY

January 2010

Table of Contents

1 Introduction

The SCASM Command and Configuration Interface Language Specification (SCCILS) is an XML specification that formally defines a language for defining and controlling SCASM-controlled systems. It is definitely still in development and pre-alpha in terms of maturity, but it is used in the NeuRoMod control suite.

Although those of us working on this have made the argument that SCASM can be useful in the description and design of a fairly broad range of systems, the current SCCILS language has a number of things that limit it. Firstly, there are limitations related to hardware-specific assumptions and elements; primarily these are in the aimotor and sensor elements. Secondly, there are limitations related to the assumed description of a leg actuated by simulated muscles. Generalizing the description of modules and actuators should certainly be doable. Note that NeuRoMod need not necessarily implement more general control in order for this to be useful; a more general SCCILS definition might be useful just to allow easier use of the SCCILStoolbox data handling and representation functions by a broader range of systems.

The SCASM representation has a number of parameters that can act as “ports” for changing the system. SCCILS makes use of these ports by developing an explicit representation language for their specification and modification. Making a system’s representation as explicit as possible can be a useful exercise in itself, but SCCILS was first developed with the following benefits of a unified/general formal representation of SCASM systems in mind:

a. One set of tools can be developed for operating and configuring SCASM systems, and the representation and interpretation of their data.

b. Unified tools for operating a model might allow:

    i. Use of the same user interface for multiple models (each with specific benefits and weaknesses), or use of multiple interfaces (each with a specific purpose).

    ii. representational and experimental portability. one might run the exact same experiment on multiple systems, each of which might present specific advantages for modeling and development.

SCCILS was developed to fulfill these benefits, and become an integral part of the development of NeuRoMod and its data representation tools. The idea of using one interface to operate different models was present from the very beginning of the work, from the first meeting between Brandon Rutter and Marcus Blümel discussing using the controller from (Ekeberg et al. 2004), previously used in simulation, in a robot. Initial brainstorming for the actual implementation of SCCILS was done by Brandon Rutter, Marcus Blümel and Arndt von Twickel in Cologne in June-July of 2006; they wanted to be able to control multiple models without rewriting the user interface. Furthermore, they wanted to be able to run exactly the same experiment in multiple models with minimal fuss.

1.1 The structure of a SCCILS-capable system

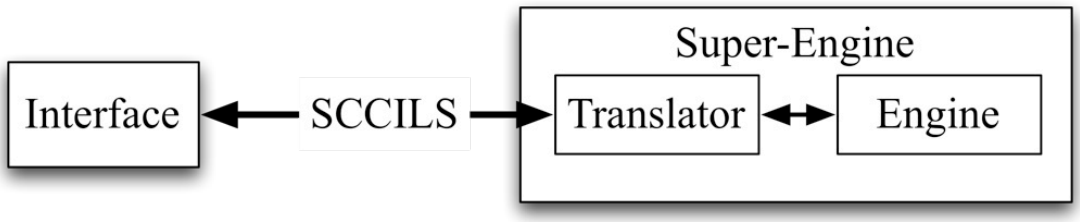


Figure 1-1 The basic layout of a system using SCCILS. "Engine" is "The model"- robot & dynamic control software, or simulation. Once configured the engine can operate by itself. An "interface" is something that sends configuration & command information, and may or may not actually be a user interface.

The structure of a system using SCCILS online has two basic elements that communicate using SCCILS, the "interface" and the "engine", as shown in Figure 1-1. The engine is the entity directly controlling the plant. This could for example be the physical robot and control software, or a computational simulation. SCCILS is used for configuration and control of the engine, and contains methods for data handling as well. An interface-less engine might still use SCCILS for configuration and experiment scripting.

The interface is something that sends these configuration and control commands, and might want data in return. It could be a graphical user interface (GUI) as done in NeuRoMod, or a higher-level control program, or even a program that could run optimizations or higher-level experiments.

The translator is an element that takes in the SCCILS XML commands and translates them into the native representation of the engine used directly in control. It is possible that a program implementing an engine might also contain translator code, or they might be separate programs.

The concepts of SCASM and SCCILS are not tied to implementation in legged systems, but since the NeuRoMod control code was originally developed for such systems *before* the development of SCCILS, there are still some portions of that code that have not been generalized, and assume implementation in legs.

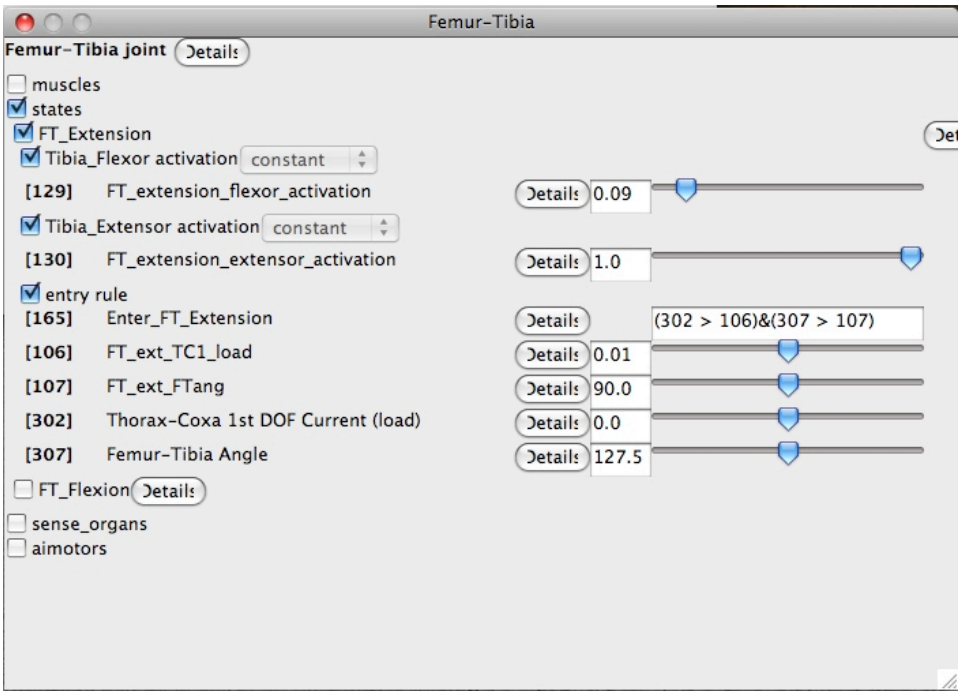


Figure 1-2 Romod_GUI representation of elements associated with the FTi Extension action. HDSO elements 129 and 130 are muscle activations; a part of the dynamics of this action, which may be modified online. 165 is the Boolean entry rule "Enter_FT_Extension"; its arguments are the IDs of the HDSOs shown below it. It says ((ThC1 load > FT_EXT_TC1_load) AND (FTi angle > FT_EXT_FTang)). The paramater HDSOs (106 & 107) below may be modified in order to change the strength/thresholds of those influences. Sensor HDSOs report robot state. Modificaiton of transition parameters or activation parameters can drastically affect system performance, but the entry rule is what determines the connections in the sensory flow and event space diagrams, and defines the nominal behavioral progression in the global state diagram.

The NeuRoMod software contains a graphical user interface for SCCILS written by John Bender. This interface is capable of modifying any SCCILS-aware parameters in the engine it connects to, and it automatically arranges its layout based on the configuration of that engine. Examples of usage of this UI are given in the NurRoMod manual, and

Figure 1-2 shows the representation of parameters associated with a particular action of the cockroach forward walking configuration described in the dissertation.

1.2 The Configuration/Data Type Sublanguage

The jointlist is the root element of a SCCILS model description file, and defines at least one leg as a chain of joints. This contains or references all the data needed to define the interface-engine communication in SCCILS for a particular model. A jointlist describes a SCASM system to the level of an "implementation-specific sensory flow diagram" as described in section 4.5 of Brandon Rutter's dissertation, representing both the flow of sensory signals represented in an event-space diagram and the thresholds and action parameters necessary for a full specification. Numerous parameters may be modified online, but the available selection of modules, actions and sensors is constant during system operation.

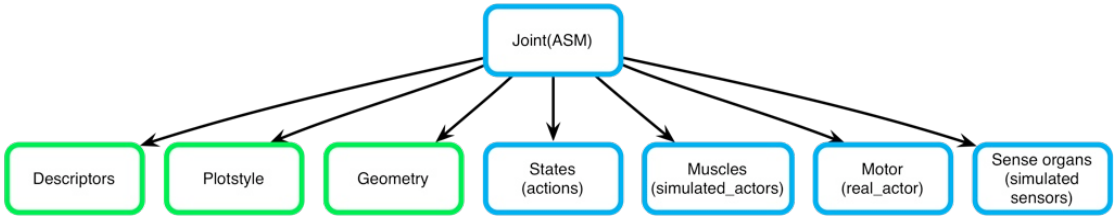


Figure 1-3 The contents of a SCCILS joint. The green elements are useful for user interfaces and data handling, but are not descriptors of the SCASM action switching module represented by the joint.

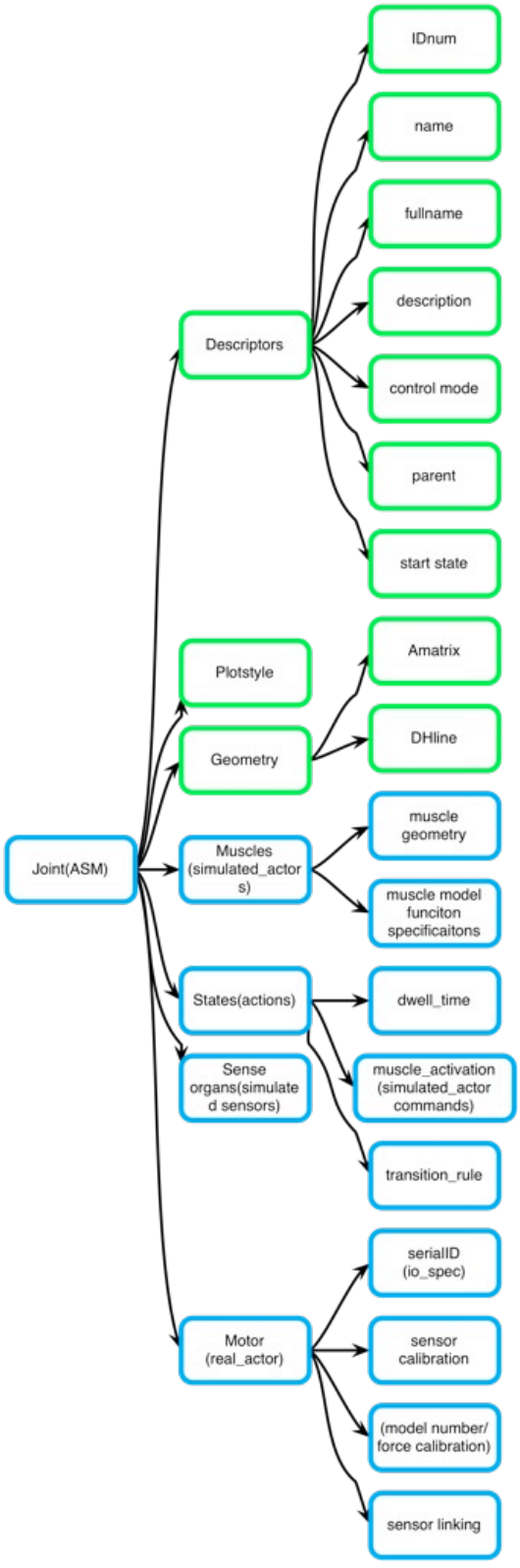


Figure 1-4 Everything in a joint. Yikes!

During the initial development of the SCCILS concept, we discovered that much of the data we wanted to pass between the interface and the engine was in a similar form. Handy Data Structure Objects (HDSOs) are really for data storage, but there are a number of things within them for making modification & UI representation more easily. Anything represented as an HDSO can theoretically be modified online or changed as part of an experimental script, though the engine is not necessarily required to be able to implement all HDSO changes online. These modifications are accomplished using an HDSOscript, which is a timed list of HDSO data modifications. Most of the SCCILS commands are in the form of an HDSOscript.

The full configuration of a SCASM system is currently specified by three XML files: the jointlist, a list of parameter HDSOs, and a list of sensor HDSOs.

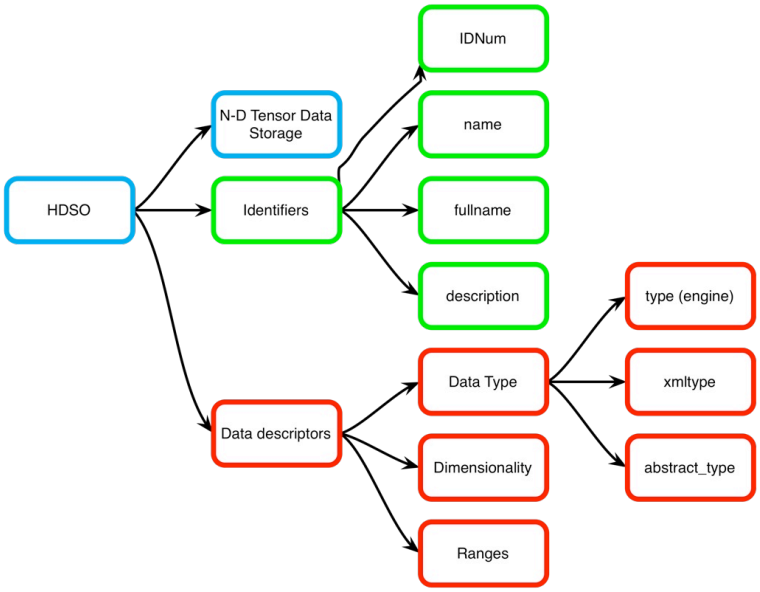


Figure 1-5 The Handy Data Storage Object; an N-dimensional data store with a set of identifiers and descriptors.

1.3 The Command Sublanguage

The SCCILS command sublanguage is a fairly simple set of elements: the sccils_command, sccils_command_response, and sccils_config_command. The sccils_command allows run, stop, data logging configuration, and extended commands. In order to be considered SCCILS-capable, an engine or interface must implement and respond properly to the run and stop commands, at the very least. A sccils_command may contain an HDSOscript for specification of HDSO data having to do with an extended command. The sccils_command_response is used for acknowledging both command and configuration messages, and reporting whether these messages were received and implemented successfully.

The sccils_config_command is used to communicate which SCCILS configuration to use (if several are available to the engine), as well as to specify the configuration of the SCCILS XML communication itself, such as setting TCP port numbers for data streaming.

2 XML DTDs

The Document Type Definition files (DTDs) that define SCCILS are a part of the NeuRoMod Subversion repository. The four defining files are included here. Though writing a DTD is pretty easy to do if you have a clear idea what you want, it is not that easy to see the hierarchical structure of the data. In order to do this, we recommend using a program such as Oxygen to convert the DTD to a schema, and look at it graphically. Oxygen is also useful for syntax highlighting (as shown below), file creation and validation, and editing multiple XML files simultaneously. It or a similarly capable XML editor is highly recommended for configuration and script editing.

There are also a number of restrictions or conventions that are a part of SCCILS but not explicitly enforced by the DTDs. These extra specifications are included in the comments of the DTDs, however, which are generally quite helpful in the description of elements of the specification.

2.1sccils_joint : the jointlist DTD

```
<!--The scills_joint is a structure/data type which should contain
or reference all the data needed to define the interface/engine
communication in SCCILS (we hope!)-->

<!--Also included in this file are engine-specific configuration elements.
These are separately defined, so that they may be more easily moved to a
separate, extending DTD, and the SCCILS definitions can then be self-contained
and general -->

<!-- sccils_joint uses hdsoMod data specifiers; include that DTD -->

<!ENTITY % hdsoscript SYSTEM "hdsoscript.dtd">
%hdsoscript;

<!-- Definitions specific to the romod engine; some of these might be
appropriately incorporated in the SCCILS specification -->

<!-- The joint also has sense organ descriptors and motor configuration data.
    joint plotstyle refers to the style used to plot joint position, muscles are
    those which originate proximally and insert distally to this joint, states are
```

```
    those which specify control of those muscles, which in turn determine the actual
    output of the (ai)motor here.  Geometry is described in it's own place.-->
<!ELEMENT joint (description?, plotstyle?, muscle*, geometry, state*, sense_organ*, aimotor?)>

<!-- Control for this joint starts in start_state, if defined.  Specified
    as the state's idNum. start_state_idNum defaults to the (hopefully) invalid "-1", which
    allows for a more reasonable treatment of "endpoint" joints.-->
<!-- the control mode can be set to plain position or force control, as well
    as muscle model control -->

<!-- TODO make this and other idNum references actual ID (i.e. name) references
-->

<!ATTLIST joint
   start_state_idNum CDATA "-1"
   control_mode (position | force | muscle | none) "muscle"
>

<!-- The sense organ describes a biological sense organ, which may be
associated with any number of real model sensors (or simulated/derived sensations)

The sensors are described in a sensor HDSOlist; here specified as HDSOmods.  They should
probably have empty data components, though one might imagine initialization being useful.
-->

<!ELEMENT sense_organ (description?, hdsoMod*)>

<!ATTLIST sense_organ
   name ID #REQUIRED
   idNum CDATA #REQUIRED
>

<!ELEMENT aimotor (description?, sensorspec*, forceCommand?, positionCommand?)>

<!-- dir: directionality of torque (and angle?) -->
<!-- angle calibration: raw = angleCal1 + angleCal2*angle_in_degrees -->
<!-- modelnum: model number of the motor; carries max torque info -->
<!-- TODO convert "aimotor" to general "motor" description; move calibrations to
the sensorspecs and generalize as a functionspec entry, change serialID to general
IOAddrSpec of some sort -->
<!-- positionCommand and forceCommand should act as 'sensors' in muscle control mode, but
probably should be "parameter" abstract type since they will be commands in force or position
control mode -->

<!ATTLIST aimotor
        name ID #REQUIRED
        idNum CDATA #REQUIRED
        serialID CDATA #REQUIRED
        dir ( 1 | -1) #REQUIRED
        angleCal1 CDATA #REQUIRED
        angleCal2 CDATA #REQUIRED
        modelnum CDATA #IMPLIED
>


<!-- Sensorspec is used to specify sensors with HDSO data storage; particularly those associated
```

```
with data from a particular aimotor. The "type" is used to tell which motor data should be pushed
to the HDSO indicated by the hdsomod (which should probably not have any initial data included).-->

<!ELEMENT sensorspec (hdsoMod)>

<!ATTLIST sensorspec
      type (angle | velocity | load | other) #REQUIRED
>

<!ELEMENT forceCommand (hdsoMod)>

<!ELEMENT positionCommand (hdsoMod)>

<!-- ************************************************************************
************************************************************************
************************************************************************
Here is where the SCCILS general description starts -->

<!ELEMENT jointlist (description?, joint*, (endpoint? | jointlist*))>
<!-- A joint list should define, at least, a leg.  Perhaps an entire animal,
possibly heirarchically.  Thus it might want a name -->
<!-- TODO: does a jointlist want a parent attribute, or is it enough for only
the joints to have this? -->

<!ATTLIST jointlist
   name ID #IMPLIED
   fullname CDATA #IMPLIED
>

<!-- Each joint has a set of muscles, a geometry, and a set of possible
muscle activation states. -->

<!-- commented out here because We don't have modularity working
<!ELEMENT joint (muscle*, geometry, state*)>
-->

<!-- parent contains the idNum of the parent joint (i.e., the connected joint
    which is closer to the base frame).  0 if the parent is the base frame -->
<!ATTLIST joint
   name ID #REQUIRED
   fullname CDATA #IMPLIED
   idNum CDATA #REQUIRED
   parent CDATA #REQUIRED
>

<!-- A "endpoint" is a special stateless, motorless joint; mostly provided for UI and
data processing use. Note that it just contains a joint; however this joint should not
have any muscles, states or motors-->

<!ELEMENT endpoint (joint)>

<!-- Each muscle activation state has a set of muscle activation rules/functions
and a transition rule for entering this state -->

<!ELEMENT state (description?, plotstyle?, dwell_time, muscle_activation*, transition_rule)>
```

```
<!ATTLIST state
   name ID #REQUIRED
   idNum CDATA #REQUIRED
   fullname CDATA #IMPLIED
>

<!-- The muscle activation can currently be nameless; it is presumed that it only
has meaning withing the enclosing state, and each applies to only one muscle.
The hdsoMods are specifiers for the sensors and parameters used in the activation
function; number required depends on the activation function, and the order matters.
-->

<!ELEMENT muscle_activation (description?, hdsoMod*) >

<!-- The activation function is currently just an index into the activation
     functions defined in enginemod- I guess this belongs in the "extension"
     part of this file somehow... -->
<!-- Whether the following are required or not depends on what the
     activation function is.  They are lists of idNums of sensor and
     parameter HDSOs, respectively. -->

<!ATTLIST muscle_activation
   name CDATA #IMPLIED
   muscle CDATA #REQUIRED
   activation_function (constant | linear | bilinear) #REQUIRED
   activation_sensors CDATA #IMPLIED
   activation_parameters CDATA #IMPLIED
>

<!-- transition_rule is represented at runtime as an HDSO of abstract type
   boolstring; however for the configuration file it is allowed to be a
   string, for more user-friendly representation, and can be transformed
   into the lower-level representation by the appropriate parsing function.
   It is expected that in any case the "variables" in the string are idNums
   for either parameters or sensors.

   The final list of hdsoMods are (optional) initializers for any of the
   HDSOs referenced in the boolean string.  If, in the process of parsing
   the document, an HDSO is initialized more than once, the later entries
   will overwrite previous ones, as is standard for hdsoMod entries.
-->

<!ELEMENT transition_rule (description?, hdsoMod+)>

<!-- The representation_type attribute indicates whether this
transition rule is represented as a textual string or an HDSO; the
hdsoMod specifying the initial/default data; this then determines
whether this data comes from the boolstring attribute or is in the initial
hdsoMod, which must at any rate be there in order to specify the appropriate
HDSO for runtime storage and communication.  It defaults to text -->

<!ATTLIST transition_rule
   representation_type (text | HDSO) "text"
   boolstring CDATA #IMPLIED
>
```

```
<!-- The "muscle" describes a simply modeled muscle: current implementation
    more or less assumes a very linear muscle with only one nervous response
    characteristic. Also assumes that joint angle is available, I think; should
    make this explicit.

    The muscle force is calculated as:
    passive_stiffness + activation * force-length * force-velocity * fmax

    The muscle should really have three function_specs, with appropriate
    function_use attributes:
    "passive_stiffness"
    "force-length"
    "force-velocity"
-->

<!ELEMENT muscle (description?, function_spec*)>

<!-- Lnaught: Resting length of active portion of the muscle -->
<!-- r: insertion radius -->
<!-- alpha: joint angle at which muscle is at its resting length,
    in degrees -->
<!-- fmax: maximum force muscle is capable of producing (undefined
    units!) -->

<!ATTLIST muscle
   name ID #REQUIRED
   fullname CDATA #IMPLIED
   idNum CDATA #REQUIRED
   Lnaught CDATA #REQUIRED
   r CDATA #REQUIRED
   alpha CDATA #REQUIRED
   fmax CDATA #REQUIRED
>

<!-- function_spec specifies a "generic function", currently one of the options
    given in the function_type attribute.  The hdsoMod entries both indicate the
    function parameter HDSOs, and values with which to initialize them.
    function types piecewise_linear and exponential are not yet implemented, but
    should be at some point.
          -->
<!ELEMENT function_spec (hdsoMod*)>

<!--function_type: constant: output = params[0];
              linear: output = params[0]*arg + params[1];
       parabolic: params[0] = a, params[1] = xo,
         output = (arg^2 - 2*arg*xo + xo^2 + 4*a*params[2]) / (4*a);
       hyperbolic: output = (params[0]/(arg-params[1])) + params[2];
   -->
<!-- function_use is used to tag the appropriate usage of the described
    function within the enclosing model element -->
<!ATTLIST function_spec
   type (constant | linear | parabolic | hyperbolic | other) #REQUIRED
   use CDATA #IMPLIED
>

<!-- The geometry element describes how this joint is related to the previous one
```

```
    (using Denavit-Hartenberg parameters?) -->

<!ELEMENT geometry (description?, amatrix?, DHline?)>
<!-- Geometry should contain perhaps an amatrix and/or a DHline, with which to calculate the
   position of the endpoint of the link starting at this joint.  A joint with an empty
   geometry indicates the end of the kinematic chain, though that may serve as the parent
   of another joint/jointlist-->

<!ELEMENT amatrix (hdsoMod)>
<!-- amatrix is an augmented 4x4 matrix, (poentially) indicating the translation and rotation of
    the base of said joint with respect to the endpoint of the previous joint in the chain or, if the
    joint's parent is "0", the body, with respect to the body coordinate system. -->

<!ELEMENT DHline (hdsoMod)>
<!-- DHline is a four-element vector consisting of the following:<br>
       (1) the "a_i" Denavit-Hartenberg parameter; length of common normal
          between axis z_(i-1) and z_i <br>
       (2) the "d_i" parameter; distance between O_(i-1) and H_i (along
          z_(i-1).  The variable in prismatic joints. <br>
       (3) the "alpha_i" parameter; angle in RADIANS between axis i
          (along z_(i-1)) and z_i  <br>
       (4) a scaling factor, multiplied by the theta parameter to convert
          it to radians.  <br>
          TODO put this scaling factor somewhere else, fer cryin' out loud!
-->

<!-- a description is just that- a textual description of the enclosing element. Now inherited from hdsoscript-
->
<!-- <!ELEMENT description (#PCDATA)> -->

<!-- <!ELEMENT charboolstring (#PCDATA)> -->

<!ELEMENT dwell_time (hdsoMod)>

<!--plotstyle allows for the specification of MATLAB-style line and color specifications. Later users
should
feel free to make this more flexible; for the moment "linespec" is a character line description; i.e. 'kx-'
for a black solid line with x's at the data points, and "colorspec" is an (r,g,b) vector which is passed as
the 'Color' plot attribute for the line only -->
<!-- TODO HDSO-ify linespec and colorspec; probably requires better handling of "string" HDSO types --
>
<!ELEMENT plotstyle EMPTY>

<!ATTLIST plotstyle
   linespec CDATA #IMPLIED
   colorspec CDATA #IMPLIED
 >

<!-- the linespec hdsoMod should correspond to a character (i.e. string) HDSO with at least four elements --
>
<!--<!ELEMENT  linespec (hdsoMod)> -->

<!-- colorspec is a three-element (r, g, b) vector; should correnspond to an HDSO with float range
from 0-1.  Used for display of state data, initially in path-state plots and leg movies -->
<!--<!ELEMENT colorspec (hdsoMod)> -->
```

2.2hdso

```
<!-- an hdsolist is a list of hdso entries. -->
<!ELEMENT hdsolist (description?, hdso*)>
<!-- An hdsolist can be tagged as a particular data set type, i.e. "parameterList" -->
<!ATTLIST hdsolist
   name CDATA #IMPLIED
   fullname CDATA #IMPLIED
   DataSetType CDATA #IMPLIED
>

<!-- all actual HDSOs should have a non-negative idNum; negative idNums may be
    specified in, say, an HDSOmod, with at least the following special meanings:
       -1 : all HDSOs in the data store; HDSO_ALL_IDNUM
       -2 : no HDSOs ; HDSO_NONE_IDNUM
       -3 : id for an initialized but unused HDSO; HDSO_BLANK_IDNUM
       -4 : explicitly invalid; ideally should throw an error somewhere; HDSO_INVALID_IDNUM
-->
<!-- an hdso has a bunch of attributes and contains numdims dimsize elements -->
<!-- fullname is a longer name for better user friendliness-->
<!-- The abstractType describes what kind of thing is in the HDSO.  A parameter
    is expected to be a constant unless modified by the user.  A sensor is
    expected to change "on its own", and only be "looked up" by the user, though
    one might imagine exceptions to this expectation.  It is intended that
    support be added for making "virtual" sensors, whose values are calculated
    from other sensors upon accessing the HDSO; this is not yet implemented.
    Both parameters and sensors may be tensors of any size, of any underlying
    data type.

    The boolstring abstract type is a special type consisting of a 2xn matrix
    of integers.  The first row of the matrix contains a map of which entries
    in the second row are boolean operators, represented as a 1 for an operator,
    0 for a variable, -1 for "null string" i.e., the boolean string stops
    there.  The second row contains this boolean string, with
    operators being represented by an enumerated type consisting of the
    possible boolean operators, and variables
    containing the idNum of an HDSO; most likely a parameter or sensor!
-->
<!-- overflowCorrection indicates whether range overflow should
    be detected and/or corrected; e.g. for sensors; 0=no, 1=yes,
    other values may be differently interpreted-->

<!ELEMENT hdso (description?, (dimsize, range*)*, initial_data?)>
<!ATTLIST hdso
   idNum CDATA #REQUIRED
   name ID #REQUIRED
   fullname CDATA #IMPLIED
   abstractType CDATA #IMPLIED
   units CDATA #IMPLIED
   datatype (float | double | int | lint | llint | char | uchar) "float"
   xmltype (float | double | int | lint | llint | char | uchar) "float"
   numdims CDATA #REQUIRED
   overflowCorrection CDATA "0"
>
```

```
<!-- The dimsize element has an attribute telling which dimension it
     specifies the size of, and the element should just be an integer.
     Currently the dimension sizes need to be listed in order (?) -->
<!ELEMENT dimsize (#PCDATA)>
<!ATTLIST dimsize
   dimNr CDATA #REQUIRED
>

<!-- A range defines both the range and quantization of an HDSO variable.
   Datatype will limit valid quantization arguments; this must be checked
   by the program using validating the data content if this behavior is
   important.  At present there is no convention on what a tensor range
   or quantization means, but there is also no rule against declaring them.
   lack of range or quantization elements implies that the range is unlimited
   or the variable is continuous, respectively.
-->
<!ELEMENT range (bound*, quantum?)>
<!ATTLIST range
   type (suggested | valid | normal | invalid) #REQUIRED
>

<!ELEMENT bound (#PCDATA)>
<!ATTLIST bound
   type (upper | lower) #REQUIRED
>

<!ELEMENT quantum (#PCDATA)>

<!-- the optional data initializer; a list of elements ordered appropriately
for the dimsizes

TODO Make this an hdsoMod (?)-->
<!ELEMENT initial_data (#PCDATA)>

<!ELEMENT description (#PCDATA)>
```

2.3hdsoscript

```
<!-- The hdsoscript is a list of modifications to the data portion
     of an HDSO.  It can optionally be tagged with a script type attribute. -->
<!ELEMENT hdsoscript (description?, (hdsoMod* | hdsoSpec*)*)>
<!ATTLIST hdsoscript
   name CDATA #IMPLIED
   fullname CDATA #IMPLIED
   ScriptType CDATA #IMPLIED
>
<!-- The hdsoMod would ideally allow the rewrite of specific elements in
     the HDSO data tensor.  Right now, though, it just writes from the
     beginning, specifying the data type and number of elements.  The
     interpretation program then parses the PCDATA element appropriately.
     idNum specifies the appropriate HDSO to modify.  Perhaps do this with
     the actual HDSO name later? - beginning support of this; "idByName" indicates
     whether to do assignment by name instead of idNum.  if "y", idNum could be set
     to HDSO_INVALID_IDNUM or some such-->
<!ELEMENT hdsoMod (#PCDATA)>
```

```
<!-- note that "time" is time in seconds since start of experiment-->
<!ATTLIST hdsoMod
   idNum CDATA #REQUIRED
   name CDATA #IMPLIED
   idByName (y | n) "n"
   NumElts CDATA #REQUIRED
   datatype (float | double | int | lint | llint | char | uchar) "float"
   time CDATA "0"
>

<!-- hsdoSpec is essentially an hdsoMod with no enclosed data; its intended use is
    for requesting, or specifying, an HDSO, or elements thereof.  At the moment
    it only specifies number of elements requested starting at the beginning, as in
    an hdsoMod. Datatype is an inherent property of the HDSO, and is not necessary for
    calculating data size here, so is omitted.  No, it isn't.  Why is it here?-->

<!ELEMENT  hdsoSpec EMPTY>

<!ATTLIST hdsoSpec
   idNum CDATA #REQUIRED
   name CDATA #IMPLIED
   idByName (y | n) "n"
   NumElts CDATA "0"
   time CDATA "0"
   datatype (float | double | int | lint | llint | char | uchar) "float"
>

<!ELEMENT description (#PCDATA)>
```

2.4sccils_commandscript

```
<!-- The sccils_commandscript is a list of SCCILS model commands.  It is possible
    and even likely that the command and command_response elements will be used outside
    of a commandscript.  There is currently no required order to the commands and command
    responses.  It can optionally be tagged with a ScriptType. -->
<!ELEMENT sccils_commandscript (sccils_command*,
                        sccils_command_response*,
                        sccils_command_config*)*>
<!ATTLIST sccils_commandscript
   ScriptType CDATA #IMPLIED
>

<!-- sccils_command uses hdsoscripts; include that DTD -->

<!ENTITY % hdsoscript SYSTEM "hdsoscript.dtd">
%hdsoscript;

<!-- The sccils_command contains a required command type attribute, a number of
    attributes whose meaning may vary with command type, a possible list of HDSOs
    to which to apply the command, and possible further
    character data for use in execution of the command. -->

<!ELEMENT sccils_command (hdsoscript?) (#PCDATA)>
```

```
<!ATTLIST sccils_command
   commandtype (run | stop | log | stoplog | extended) #REQUIRED
   extendedcommand CDATA #IMPLIED
   starttime CDATA "0"
   stoptime CDATA "-1"
   data_destination (local | network) "network"
>

<!-- commandtype: one of the four standard SCCILS commands (described below), or
        an engine-specific extended command.
    extendedcommand: name of the extended command, if appropriate
    starttime: for "run", a delay to wait until starting the engine., for "log", the time
        since engine start at which to start data logging.
    stoptime: for "run" and "log" commands, time at which to stop engine or logging, with
        respect to engine start time.  Not used for "stop" or "stoplog"
            TODO: should this be changed?  do we want to be able to do a delayed stop
                at the user's request?  just do this with "run" and "log" maybe?
    data_destination: where to log the data; locally at the engine, or via the SCCILS
        network HDSO data service to the interface.
-->

<!-- The sccils_command_response is intended as just that; a response to a specific
    sccils_command.  It contains said sccils_command, possibly modified, and optional
    character data.
-->

<!ELEMENT sccils_command_response (sccils_command) (#PCDATA)>

<!ATTLIST sccils_command_response
   responsetype (ack_only | withdata) "ack_only"
   responsestatus (success | fail | conditional | other) #REQUIRED
   responsetime CDATA #IMPLIED
>

<!-- responsetype: ack_only means this is only and acknowledgement of the success or
        failure of the enclosed, unmodified sccils_command.  withdata implies that
        either the encolsed command has somehow been modified, or the response PCDATA
        has something of interest.  Defaults to ack_only
    responsestatus: status of the execution of sccils_command; "conditional" means that
        success or failure depends on whether this response is processed successfully.
        The response status must always be specified.
    responsetime: optional time for when the response was made/sccils_command was acted upon
-->

<!ELEMENT sccils_config_command (sccils_config_parameter)* (#PCDATA)>
<!ATTLIST sccils_config_command
   configtype (available | start | parameter) #REQUIRED
   configname CDATA #REQUIRED
>

<!ELEMENT sccils_config_parameter (#PCDATA)>

<!-- sccils_config_command is used for a server to communicate what
    types of simulations it is capable of running, then for a client
    to request one of those, then for the server to pass additional
    configuration information (PCDATA) related to the request.
```

```
    Also used to pass other parameters in response to requests.
-->

<!-- example commands -->

<!-- Just run the engine:

<sccils_command commandtype="run" />

This will tell the engine to run at time 0 (now, no delay) with no stop time (since the stop time
is negative and therefore invalid).


Run the engine in ten seconds, for ten seconds, and then stop:
<sccils_command commandtype="run" starttime="10" stoptime="10" />

-->


<!-- Stop the engine; just stops the engine immediately.  Similar for stoplog.

<sccils_command commandtype="stop" />

-->

<!-- Start the network HDSO data logging service immediately:

<sccils_command commandtype="log" />

-->

<!-- A more complex experimental run.  Start the engine in ten seconds, and run it for ten.  Start
    logging data locally after 2.5 seconds of engine run time, and stop after having logged for five
    seconds, at 7.5 seconds of engine run time.

<sccils_commandscript ScriptType="Example">
  <sccils_command commandtype="run" starttime="10" stoptime="10" />
  <sccils_command commandtype="log" starttime="2.5" stoptime="7.5" data_destination="local" />
</sccils_commandscript>

-->


<!--
here's an example of a series of communications between server and client to
set up a new connection, configure, and start the engine, then poll for data

[client connects to server, they handshake]

[server gives list of available configurations]
   <sccils_commandscript>
      <sccils_config_command configtype="available" configname="null_roach"/>
      <sccils_config_command configtype="available" configname="lin_roach"/>
   </sccils_commandscript>
[client acknowledges]
   <sccils_commandscript>
```

```
      <sccils_command_response responsestatus="success"/>
   </sccils_commandscript>

[after user input, client chooses a confguration]
   <sccils_commandscript>
      <sccils_config_command configtype="start" configname="lin_roach"/>
   </sccils_commandscript>
[server acknowledges receipt, but is agnostic about success]
   <sccils_commandscript>
      <sccils_command_response responsestatus="conditional"/>
   </sccils_commandscript>

[server begins its configuration, sends config-specific info to client]
   <sccils_commandscript>
      <sccils_config_command configtype="parameter" configname="lin_roach">
         <sccils_config_parameter>joints.xml</sccils_config_parameter>
         <sccils_config_parameter>sensors.xml</sccils_config_parameter>
      </sccils_config_command>
   </sccils_commandscript>
[client acknowledges]
   <sccils_commandscript>
      <sccils_command_response responsestatus="success"/>
   </sccils_commandscript>

[now server sends the complete XML scripts specified ('joints.xml'
 and 'sensors.xml') to the client; client acknowledges successful
 receipt of each]

[server does internal configuration, sends success or failure]
   <sccils_commandscript>
      <sccils_command_response responsestatus="success"/>
   </sccils_commandscript>

[client decides to start the recently configured engine]
   <sccils_commandscript>
      <sccils_command commandtype="run"/>
   </sccils_commandscript>
[server begins running, sends success or failure]
   <sccils_commandscript>
      <sccils_command_response responsestatus="success"/>
   </sccils_commandscript>

[client changes a configuration parameter and sends a script to modify it]
   <hdsoscript>
      <hdsoMod time="0" idNum="13" NumElts="1" datatype="float">
         3.0
      </hdsoMod>
   </hdsoscript>
[server acknowledges script receipt, but is agnostic about success]
   <sccils_commandscript>
      <sccils_command_response responsestatus="conditional"/>
   </sccils_commandscript>

[server runs script and sends success]
   <sccils_commandscript>
      <sccils_command_response responsestatus="success"/>
```

```
   </sccils_commandscript>

[client requests a polling data stream (a new TCP/IP connection)]
   <sccils_commandscript>
      <sccils_command commandtype="log" extendedcommand="pollingmode"/>
   </sccils_commandscript>
[server acknowledges]
   <sccils_commandscript>
      <sccils_command_response responsestatus="success"/>
   </sccils_commandscript>

[server sends configuration information for data stream -- a port number]
   <sccils_commandscript>
      <sccils_config_command configtype="parameter" configname="dataport">
         <sccils_config_parameter>44106</sccils_config_parameter>
      </sccils_config_command>
   </sccils_commandscript>
[client acknowledges]
   <sccils_commandscript>
      <sccils_command_response responsestatus="success"/>
   </sccils_commandscript>

[server and client run new threads for data streaming; they handshake]

[client thread polls server thread for a data update on a single HDSO]
   <sccils_commandscript>
      <sccils_command commandtype="log" extendedcommand="poll">
         <hdsoscript>
            <hdsoMod time="0" idNum="13" NumElts="1" datatype="float">
               3.0 <!-- data is ignored, but must be present for parsing ->
            </hdsoMod>
         </hdsoscript>
      </sccils_command>
   </sccils_commandscript>
[server thread acknowledges]
   <sccils_commandscript>
      <sccils_command_response responsestatus="success"/>
   </sccils_commandscript>

[server thread retrieves values for the identified HDSOs, sends them back]
   <hdsoscript>
      <hdsoMod time="0" idNum="13" NumElts="1" datatype="float">
         2.5
      </hdsoMod>
   </hdsoscript>
[client thread acknowledges]
   <sccils_commandscript>
      <sccils_command_response responsestatus="success"/>
   </sccils_commandscript>
-->
```

3 Example Configurations

3.1lin_stickmiddle_minimal

This is a configuration which implements the minimal SCASM stepper in the stick insect robot model. It reduces that 3-DOF robot to a 2-DOF system by configuring muscles and a single state to hold the FTi joint essentially fixed.

3.1.1 joints

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE jointlist SYSTEM "/usr/local/src/romod_dtds/sccils_joint.dtd">
<!--<!DOCTYPE jointlist SYSTEM "http://roach.biol.cwru.edu/romod-dtds/sccils_joint.dtd"> -->

<jointlist name="lin_minimal_stick" fullname="stick insect middle leg, minimal SCASM example">
 <description>This is a minimal two-joint SCASM controller, implemented on the stick insect
 robot model; FT parameters are all there just because we physically need to hold the FT joint
 out for this to work.</description>

 <joint name="Thorax-Coxa" idNum="1" parent="0" start_state_idNum="201">
  <muscle name="Coxa_Remotor" idNum="401" Lnaught="2" r="-.5" alpha="0" fmax="18">
   <function_spec type="constant" use="passive_stiffness">
    <hdsoMod idNum="190" NumElts="0"/>
   </function_spec>
   <function_spec type="constant" use="force-length">
    <hdsoMod idNum="191" NumElts="0"/>
   </function_spec>
   <function_spec type="constant" use="force-velocity">
    <hdsoMod idNum="191" NumElts="0"/>
   </function_spec>
  </muscle>
  <muscle name="Coxa_Promotor" idNum="402" Lnaught="2" r=".5" alpha="0" fmax="15">
   <function_spec type="constant" use="passive_stiffness">
    <hdsoMod idNum="190" NumElts="0"/>
   </function_spec>
   <function_spec type="constant" use="force-length">
    <hdsoMod idNum="191" NumElts="0"/>
   </function_spec>
   <function_spec type="constant" use="force-velocity">
    <hdsoMod idNum="191" NumElts="0"/>
   </function_spec>
  </muscle>
  <geometry>
   <description>This A-matrix orients the z axis (joint axis) properly for the ThC joint on the right
    middle leg; in a body reference frame where positive x is forward with y pointing to the left.
    First rotation about Z Thetaz = -85 degrees, second rotation about YThetay = 40 degrees.
    Note that this currently has this joint located at the body
    (0,0,0); should figure out what the standard is for body zero-point and offset from there</description>
   <amatrix>
    <hdsoMod idNum="180" NumElts="16" datatype="float">
```

```
      0.066765172417751   0.996194698091746   0.056022631551222       0
      -0.763129412737770   0.087155742747658  -0.640341608768797      0
      -0.642787609686539                  0   0.766044443118978       0
      0                                   0                   0       1
    </hdsoMod>
   </amatrix>
   <DHline>
    <!-- These DHlines are in centimeters -->
    <hdsoMod idNum="181" NumElts="4">
     <!-- 2.26 0 (90/180)*pi pi/180 -->
     2.26   0   1.570796326794897   0.017453292519943
    </hdsoMod>
   </DHline>

  </geometry>
  <state name="TC_protraction" idNum="201">
   <dwell_time>
    <hdsoMod idNum="194" NumElts="0"/>
   </dwell_time>
   <muscle_activation muscle="401" activation_function="constant" activation_parameters="121">
    <hdsoMod idNum="121" NumElts="1" time="0">0</hdsoMod>
   </muscle_activation>
   <muscle_activation muscle="402" activation_function="constant" activation_parameters="122">
    <hdsoMod idNum="122" NumElts="1" time="0">0.6</hdsoMod>
   </muscle_activation>
   <transition_rule representation_type="text" boolstring="304 &lt; 101">
    <hdsoMod idNum="161" NumElts="0"/>
    <hdsoMod idNum="101" NumElts="1">-0.5</hdsoMod>
   </transition_rule>
  </state>
  <state name="TC_retraction" idNum="202">
   <dwell_time>
    <hdsoMod idNum="194" NumElts="0"/>
   </dwell_time>
   <muscle_activation muscle="401" activation_function="constant" activation_parameters="123">
    <hdsoMod idNum="123" NumElts="1">1.6</hdsoMod>
   </muscle_activation>
   <muscle_activation muscle="402" activation_function="constant" activation_parameters="124">
    <hdsoMod idNum="124" NumElts="1" time="0">0.1</hdsoMod>
   </muscle_activation>
   <transition_rule representation_type="text" boolstring="304 &gt; 102">
    <hdsoMod idNum="162" NumElts="0"/>
    <hdsoMod idNum="102" NumElts="1">0.0</hdsoMod>
   </transition_rule>
  </state>
  <sense_organ name="TC_sense_organ" idNum="501">
   <description>The TC sense organ is involved in sensing the angle of the Thorax-Coxa joint.
    Maybe just limits?</description>
   <hdsoMod idNum="301" NumElts="0"/>
  </sense_organ>
  <aimotor name="Thorax-Coxa_motor" idNum="601" serialID="10" dir="-1" angleCal1="120"
   angleCal2="-1.529" modelnum="701"/>
 </joint>
 <joint name="Coxa-Trochanter" idNum="2" parent="1" start_state_idNum="203">
  <muscle name="Femur_Depressor" idNum="403" Lnaught="2" r="-1" alpha="0" fmax="12">
   <function_spec type="constant" use="passive_stiffness">
```

```
    <hdsoMod idNum="190" NumElts="0"/>
   </function_spec>
   <function_spec type="constant" use="force-length">
    <hdsoMod idNum="191" NumElts="0"/>
   </function_spec>
   <function_spec type="constant" use="force-velocity">
    <hdsoMod idNum="191" NumElts="0"/>
   </function_spec>
  </muscle>
  <muscle name="Femur_Levator" idNum="404" Lnaught="2" r="1" alpha="0" fmax="15">
   <function_spec type="constant" use="passive_stiffness">
    <hdsoMod idNum="190" NumElts="0"/>
   </function_spec>
   <function_spec type="constant" use="force-length">
    <hdsoMod idNum="191" NumElts="0"/>
   </function_spec>
   <function_spec type="constant" use="force-velocity">
    <hdsoMod idNum="191" NumElts="0"/>
   </function_spec>
  </muscle>
  <geometry>
   <DHline>
    <hdsoMod idNum="182" NumElts="4">
     <!--   19.28 0 pi pi/180 -->
     19.28   0   3.141592653589793   0.017453292519943
    </hdsoMod>
   </DHline>
  </geometry>
  <state name="CTr_Levation" idNum="203">
   <dwell_time>
    <hdsoMod idNum="194" NumElts="0"/>
   </dwell_time>
   <muscle_activation muscle="403" activation_function="constant" activation_parameters="125">
    <hdsoMod idNum="125" NumElts="1">0</hdsoMod>
   </muscle_activation>
   <muscle_activation muscle="404" activation_function="constant" activation_parameters="126">
    <hdsoMod idNum="126" NumElts="1">0.5</hdsoMod>
   </muscle_activation>
   <transition_rule representation_type="text" boolstring="301 &lt; 104">
    <hdsoMod idNum="163" NumElts="0"/>
    <hdsoMod idNum="104" NumElts="1">-25</hdsoMod>
   </transition_rule>
  </state>
  <state name="CTr_Depression" idNum="204">
   <dwell_time>
    <hdsoMod idNum="194" NumElts="0"/>
   </dwell_time>
   <muscle_activation muscle="403" activation_function="constant" activation_parameters="127">
    <hdsoMod idNum="127" NumElts="1">0.7</hdsoMod>
   </muscle_activation>
   <muscle_activation muscle="404" activation_function="constant" activation_parameters="128">
    <hdsoMod idNum="128" NumElts="1">0</hdsoMod>
   </muscle_activation>
   <transition_rule representation_type="text" boolstring="301 &gt; 105">
    <hdsoMod idNum="164" NumElts="0"/>
    <hdsoMod idNum="105" NumElts="1">20</hdsoMod>
```

```
    </transition_rule>
   </state>
   <sense_organ name="Trochanter_CS" idNum="502">
    <description>The trochanteral canpaniform sensilla are load sensors critical to agile behavior
     and adaptation to varying conditions.</description>
    <hdsoMod idNum="304" NumElts="0"/>
   </sense_organ>
   <aimotor name="Coxa-Trochanter_motor" idNum="602" serialID="11" dir="-1" angleCal1="124"
    angleCal2="-1.529" modelnum="1001"/>
  </joint>
  <joint name="Femur-Tibia" idNum="3" parent="2" start_state_idNum="205">
   <muscle name="Tibia_Flexor" idNum="405" Lnaught="5" r="1" alpha="95" fmax="30">
    <function_spec type="constant" use="passive_stiffness">
     <hdsoMod idNum="190" NumElts="0"/>
    </function_spec>
    <function_spec type="linear" use="force-length">
     <hdsoMod idNum="171" NumElts="2">2 -1.5</hdsoMod>
    </function_spec>
    <function_spec type="constant" use="force-velocity">
     <hdsoMod idNum="191" NumElts="0"/>
    </function_spec>
   </muscle>
   <muscle name="Tibia_Extensor" idNum="406" Lnaught="5" r="-1" alpha="100" fmax="14">
    <function_spec type="constant" use="passive_stiffness">
     <hdsoMod idNum="190" NumElts="0"/>
    </function_spec>
    <function_spec type="linear" use="force-length">
     <hdsoMod idNum="172" NumElts="2">2 -1.5</hdsoMod>
    </function_spec>
    <function_spec type="constant" use="force-velocity">
     <hdsoMod idNum="191" NumElts="0"/>
    </function_spec>
   </muscle>
   <geometry>
    <DHline>
     <hdsoMod idNum="183" NumElts="4">
      <!--     18.90 0 0 pi/180  -->
      18.9    0    0   0.017453292519943
     </hdsoMod>
    </DHline>
   </geometry>
   <state name="FT_Extension" idNum="205">
    <dwell_time>
     <hdsoMod idNum="194" NumElts="0"/>
    </dwell_time>
    <muscle_activation muscle="405" activation_function="constant" activation_parameters="129">
     <hdsoMod idNum="129" NumElts="1">0.09</hdsoMod>
    </muscle_activation>
    <muscle_activation muscle="406" activation_function="constant" activation_parameters="130">
     <hdsoMod idNum="130" NumElts="1">0.9</hdsoMod>
    </muscle_activation>
    <transition_rule representation_type="text" boolstring="191">
     <hdsoMod idNum="165" NumElts="0"/>
     <hdsoMod idNum="191" NumElts="0"/>
    </transition_rule>
   </state>
```

```
    <sense_organ name="Femur_CS" idNum="503">
      <description>The femoral canpaniform sensilla are load sensors; used similarly to the
        trochanteral CS, as far as we know/this model knows.</description>
      <hdsoMod idNum="302" NumElts="0"/>
      <hdsoMod idNum="304" NumElts="0"/>
    </sense_organ>
    <sense_organ name="Femur_CO" idNum="504">
      <description>The Femoral Chordotonal Organ is a continuous angle-sensing organ which operates
        as a specialized apodeme/muscle in the femur, sensing the motion of the Femur-Tibia
        joint.</description>
      <hdsoMod idNum="305" NumElts="0"/>
    </sense_organ>
    <aimotor name="Femur_Tibia_motor" idNum="603" serialID="12" dir="1" angleCal1="11"
      angleCal2="1.529" modelnum="701"/>
  </joint>
  <endpoint>
    <joint name="Tibia-Tarsus" idNum="4" parent="3">
      <description>The Tibia-Tarsus joint is in this model only the terminal endpoint.
        No actual tarsus is currently modeled.</description>
      <plotstyle linespec="r"/>
      <geometry />
    </joint>
  </endpoint>
</jointlist>
```

3.1.2 parameters

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE hdsolist SYSTEM "/usr/local/src/romod_dtds/hdso.dtd">
<!-- <!DOCTYPE hdsolist SYSTEM "http://roach.biol.cwru.edu/romod-dtds/hdso.dtd"> -->

<hdsolist DataSetType="parameterList">

  <!-- global engine parameters -->
  <hdso idNum="0" name="motor_speed" datatype="int" numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="valid">
      <bound type="upper">4</bound>
      <bound type="lower">0</bound>
      <quantum>1</quantum>
    </range>
    <range type="suggested">
      <bound type="upper">4</bound>
      <bound type="lower">0</bound>
      <quantum>1</quantum>
    </range>
    <initial_data>2</initial_data>
  </hdso>
  <hdso idNum="1" name="motor_control_delay_ns" datatype="llint" numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="suggested">
      <bound type="upper">1000000000</bound>
      <bound type="lower">0</bound>
      <quantum>10000000</quantum>
```

```
  </range>
  <initial_data>0</initial_data>
</hdso>
<hdso idNum="2" name="statetrans_period_ns" datatype="llint" numdims="1">
  <dimsize dimNr="1">1</dimsize>
  <range type="suggested">
    <bound type="upper">1000000000</bound>
    <bound type="lower">0</bound>
    <quantum>1000000</quantum>
  </range>
  <initial_data>10000000</initial_data>
</hdso>

<!-- state transition parameters -->
<hdso idNum="101" name="TC_pro_CTr_load" datatype="float" numdims="1">
  <dimsize dimNr="1">1</dimsize>
  <range type="suggested">
    <bound type="upper">2</bound>
    <bound type="lower">-3</bound>
    <quantum>0.1</quantum>
  </range>
  <range type="valid">
    <bound type="upper">20</bound>
    <bound type="lower">-20</bound>
  </range>
</hdso>
<hdso idNum="102" name="TC_ret_CTr_load" datatype="float" numdims="1">
  <dimsize dimNr="1">1</dimsize>
  <range type="suggested">
    <bound type="upper">3</bound>
    <bound type="lower">-2</bound>
    <quantum>0.1</quantum>
  </range>
  <range type="valid">
    <bound type="upper">20</bound>
    <bound type="lower">-20</bound>
  </range>
</hdso>
<hdso idNum="104" name="CTr_lev_TCang" numdims="1">
  <dimsize dimNr="1">1</dimsize>
  <range type="suggested">
    <bound type="upper">-10</bound>
    <bound type="lower">-35</bound>
    <quantum>1</quantum>
  </range>
  <range type="valid">
    <bound type="upper">180</bound>
    <bound type="lower">-180</bound>
  </range>
</hdso>
<hdso idNum="105" name="CTr_dep_TCang" numdims="1">
  <dimsize dimNr="1">1</dimsize>
  <range type="suggested">
    <bound type="upper">35</bound>
    <bound type="lower">10</bound>
    <quantum>1</quantum>
```

```
  </range>
  <range type="valid">
    <bound type="upper">180</bound>
    <bound type="lower">-180</bound>
  </range>
</hdso>


<!-- MUSCLE ACTIVATION PARAMETERS -->
<hdso idNum="121" name="TC_protraction_retractor_activation" numdims="1">
  <dimsize dimNr="1">1</dimsize>
  <range type="suggested">
    <bound type="upper">1</bound>
    <bound type="lower">0</bound>
    <quantum>0.05</quantum>
  </range>
  <range type="valid">
    <bound type="upper">2</bound>
    <bound type="lower">0</bound>
  </range>
</hdso>
<hdso idNum="122" name="TC_protraction_protractor_activation" numdims="1">
  <dimsize dimNr="1">1</dimsize>
  <range type="suggested">
    <bound type="upper">1</bound>
    <bound type="lower">0</bound>
    <quantum>0.05</quantum>
  </range>
  <range type="valid">
    <bound type="upper">2</bound>
    <bound type="lower">0</bound>
  </range>
</hdso>
<hdso idNum="123" name="TC_retraction_retractor_activation" numdims="1">
  <dimsize dimNr="1">1</dimsize>
  <range type="suggested">
    <bound type="upper">1</bound>
    <bound type="lower">0</bound>
    <quantum>0.05</quantum>
  </range>
  <range type="valid">
    <bound type="upper">2</bound>
    <bound type="lower">0</bound>
  </range>
</hdso>
<hdso idNum="124" name="TC_retraction_protractor_activation" numdims="1">
  <dimsize dimNr="1">1</dimsize>
  <range type="suggested">
    <bound type="upper">1</bound>
    <bound type="lower">0</bound>
    <quantum>0.05</quantum>
  </range>
  <range type="valid">
    <bound type="upper">2</bound>
    <bound type="lower">0</bound>
  </range>
```

```
</hdso>

<hdso idNum="125" name="CTr_levation_depressor_activation" numdims="1">
 <dimsize dimNr="1">1</dimsize>
 <range type="suggested">
  <bound type="upper">1</bound>
  <bound type="lower">0</bound>
  <quantum>0.05</quantum>
 </range>
 <range type="valid">
  <bound type="upper">2</bound>
  <bound type="lower">0</bound>
 </range>
</hdso>
<hdso idNum="126" name="CTr_levation_levator_activation" numdims="1">
 <dimsize dimNr="1">1</dimsize>
 <range type="suggested">
  <bound type="upper">1</bound>
  <bound type="lower">0</bound>
 </range>
 <range type="valid">
  <bound type="upper">2</bound>
  <bound type="lower">0</bound>
 </range>
</hdso>
<hdso idNum="127" name="CTr_depression_depressor_activation" numdims="1">
 <dimsize dimNr="1">1</dimsize>
 <range type="suggested">
  <bound type="upper">1</bound>
  <bound type="lower">0</bound>
 </range>
 <range type="valid">
  <bound type="upper">2</bound>
  <bound type="lower">0</bound>
 </range>
</hdso>
<hdso idNum="128" name="CTr_depression_levator_activation" numdims="1">
 <dimsize dimNr="1">1</dimsize>
 <range type="suggested">
  <bound type="upper">1</bound>
  <bound type="lower">0</bound>
  <quantum>0.05</quantum>
 </range>
 <range type="valid">
  <bound type="upper">2</bound>
  <bound type="lower">0</bound>
 </range>
</hdso>

<hdso idNum="129" name="FT_extension_flexor_activation" numdims="1">
 <dimsize dimNr="1">1</dimsize>
 <range type="suggested">
  <bound type="upper">1</bound>
  <bound type="lower">0</bound>
  <quantum>0.05</quantum>
 </range>
```

```
    <range type="valid">
      <bound type="upper">2</bound>
      <bound type="lower">0</bound>
    </range>
  </hdso>
  <hdso idNum="130" name="FT_extension_extensor_activation" numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="suggested">
      <bound type="upper">1</bound>
      <bound type="lower">0</bound>
      <quantum>0.05</quantum>
    </range>
    <range type="valid">
      <bound type="upper">2</bound>
      <bound type="lower">0</bound>
    </range>
  </hdso>


  <!-- STATE TRANSITION RULE PARAMETERS -->
  <hdso idNum="161" name="Enter_TC_Protraction" abstractType="boolstring" datatype="int"
numdims="2">
    <dimsize dimNr="1">2</dimsize>
    <dimsize dimNr="2">30</dimsize>
  </hdso>
  <hdso idNum="162" name="Enter_TC_Retraction" abstractType="boolstring" datatype="int"
numdims="2">
    <dimsize dimNr="1">2</dimsize>
    <dimsize dimNr="2">30</dimsize>
  </hdso>
  <hdso idNum="163" name="Enter_CTr_Levation" abstractType="boolstring" datatype="int"
numdims="2">
    <dimsize dimNr="1">2</dimsize>
    <dimsize dimNr="2">30</dimsize>
  </hdso>
  <hdso idNum="164" name="Enter_CTr_Depression" abstractType="boolstring" datatype="int"
numdims="2">
    <dimsize dimNr="1">2</dimsize>
    <dimsize dimNr="2">30</dimsize>
  </hdso>
  <hdso idNum="165" name="Enter_FT_Extension" abstractType="boolstring" datatype="int"
numdims="2">
    <dimsize dimNr="1">2</dimsize>
    <dimsize dimNr="2">30</dimsize>
  </hdso>


  <!-- MUSCLE MODEL PARAMETERS -->
  <hdso idNum="171" name="Tibia_Flexor_F-L_Params" numdims="1">
    <dimsize dimNr="1">2</dimsize>
  </hdso>
  <hdso idNum="172" name="Tibia_Extensor_F-L_Params" numdims="1">
    <dimsize dimNr="1">2</dimsize>
  </hdso>

  <!-- "constants" -->
```

```
 <hdso idNum="190" name="zero_float_const" abstractType="constant" datatype="float" numdims="1">
  <dimsize dimNr="1">1</dimsize>
  <initial_data>0</initial_data>
 </hdso>
 <hdso idNum="191" name="one_float_const" abstractType="constant" datatype="float" numdims="1">
  <dimsize dimNr="1">1</dimsize>
  <initial_data>1</initial_data>
 </hdso>
 <hdso idNum="192" name="zero_int_const" abstractType="constant" datatype="int" numdims="1">
  <dimsize dimNr="1">1</dimsize>
  <initial_data>0</initial_data>
 </hdso>
 <hdso idNum="193" name="one_int_const" abstractType="constant" datatype="int" numdims="1">
  <dimsize dimNr="1">1</dimsize>
  <initial_data>1</initial_data>
 </hdso>
 <hdso idNum="194" name="zero_llint_const" abstractType="constant" datatype="llint" numdims="1">
  <dimsize dimNr="1">1</dimsize>
  <initial_data>0</initial_data>
 </hdso>
 <hdso idNum="195" name="one_llint_const" abstractType="constant" datatype="llint" numdims="1">
  <dimsize dimNr="1">1</dimsize>
  <initial_data>1</initial_data>
 </hdso>


</hdsolist>
```

3.1.3 sensors

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE hdsolist SYSTEM "/usr/local/src/romod_dtds/hdso.dtd">
<!-- <!DOCTYPE hdsolist SYSTEM "http://roach.biol.cwru.edu/romod-dtds/hdso.dtd"> -->


<hdsolist DataSetType="sensorList">
 <hdso idNum="301" abstractType="sensor" name="TC_angle"
     overflowCorrection="1" fullname="Thorax-Coxa Angle"
     datatype="float" numdims="1">
     <dimsize dimNr="1">1</dimsize>
     <range type="normal">
      <bound type="upper">255</bound>
      <bound type="lower">0</bound>
      <quantum>1</quantum>
     </range>
 </hdso>
 <hdso idNum="302" abstractType="sensor" name="TC_current"
     fullname="Thorax-Coxa Current (load)" datatype="float"
     numdims="1">
     <dimsize dimNr="1">1</dimsize>
     <range type="normal">
      <bound type="upper">10</bound>
      <bound type="lower">-10</bound>
     </range>
```

```
</hdso>
<hdso idNum="303" abstractType="sensor" name="CTr_angle"
    overflowCorrection="1" fullname="Coxa-Trochanter Angle"
    datatype="float" numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="normal">
      <bound type="upper">255</bound>
      <bound type="lower">0</bound>
      <quantum>1</quantum>
    </range>
</hdso>
<hdso idNum="304" abstractType="sensor" name="CTr_current"
    fullname="Coxa-Troachanter Current (load)" datatype="float"
    numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="normal">
      <bound type="upper">10</bound>
      <bound type="lower">-10</bound>
    </range>
</hdso>
<hdso idNum="305" abstractType="sensor" name="FT_angle"
    overflowCorrection="1" fullname="Femur-Tibia Angle"
    datatype="float" numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="normal">
      <bound type="upper">255</bound>
      <bound type="lower">0</bound>
      <quantum>1</quantum>
    </range>
</hdso>
<hdso idNum="306" abstractType="sensor" name="FT_current"
    fullname="Femur-Tibia Current (load)" datatype="float"
    numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="normal">
      <bound type="upper">10</bound>
      <bound type="lower">-10</bound>
    </range>
</hdso>
</hdsolist>
```

3.2lin_roach

This is the forward stepping configuration; used as the starting configuration for the transition experiments described in the dissertation.

3.2.1 joints

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE jointlist SYSTEM "/usr/local/src/romod_dtds/sccils_joint.dtd">
<!-- <!DOCTYPE jointlist SYSTEM "http://roach.biol.cwru.edu/romod-dtds/sccils_joint.dtd"> -->
```

```
<jointlist name="lin_roachmiddle"
        fullname="cockroach middle leg, linear muscle models">
 <description>This describes a Blaberus discoidalis middle leg, forward stepping. Rearranged
 from the associated stickmiddle configuration file; currently with nothing
 going on at the ThC2 DOF.  Activations are currently all constant per-state.
 Linear muscle models at ThC1, CTr and FTi joints.
 </description>

 <joint name="Thorax-Coxa1" idNum="1" parent="0" start_state_idNum="201">
  <plotstyle linespec="ko"/>
  <muscle name="Coxa_RemotorLEV" idNum="401" Lnaught="20" r="-5"
        alpha="25" fmax="18">
   <function_spec type="constant" use="passive_stiffness">
    <hdsoMod idNum="190" NumElts="0"/>
   </function_spec>
   <function_spec type="linear" use="force-length">
    <hdsoMod idNum="171" NumElts="2">2 -1.5</hdsoMod>
   </function_spec>
   <function_spec type="constant" use="force-velocity">
    <hdsoMod idNum="191" NumElts="0"/>
   </function_spec>
  </muscle>
  <muscle name="Coxa_PromotorDEP" idNum="402" Lnaught="20" r="10"
        alpha="20" fmax="15">
   <function_spec type="constant" use="passive_stiffness">
    <hdsoMod idNum="190" NumElts="0"/>
   </function_spec>
   <function_spec type="linear" use="force-length">
    <hdsoMod idNum="172" NumElts="2">2 -1.5</hdsoMod>
   </function_spec>
   <function_spec type="constant" use="force-velocity">
    <hdsoMod idNum="191" NumElts="0"/>
   </function_spec>
  </muscle>
  <geometry>
   <description>This A-matrix orients the z axis (joint axis) properly for the ThC joint on the left
    middle leg; in a body reference frame where positive x is forward with y pointing to the left.
    First rotation about X Thetax = 130 degrees, second rotation about Y Thetay = -20 degrees,
    and a third rotation about Z Thetaz = 180 degrees to set the theta zero point properly.  The
    resulting coordinate system points Z along ThC1 into the body, X points basically cuadally and
    Y essentially ventrally. The
    offset corresponts to a (0,0,0) point on the ground beneath the coordinate frame; this is
    different from the supposed standard of this Amatrix defining an offset from the _body_ coordinate
    system.  Units in centimeters.
    The DHLine in this joint's geometry entry has a length of 1 cm entered- really there is no length
    before the ThC2 DOF, but this one centimeter doesn't change things much, and allows the possibility
    of properly displaying ThC1 state.  This is a kluge; really makelegavi should allow different line/point
    styles and widths per state; this would allow a zero-length line to change visibly. Currently it's
    just colors that change, which are not usually visible for a zero-length line.
   </description>
   <amatrix>
    <!-- New, actual amatrix (z and x point the other way)
     thetax = (130/180)*pi;
     thetay = (-20/180)*pi;
     thetaz = (180/180)*pi;
```

```
        >> Rx = rx(thetax)
        Rx =
        1.0000        0        0
        0   -0.6428   -0.7660
        0    0.7660   -0.6428

        >> Ry = ry(thetay)
        Ry =
        0.9397        0   -0.3420
        0    1.0000        0
        0.3420        0    0.9397

        >> Rz = rz(thetaz)
        Rz =
        -1.0000   -0.0000        0
        0.0000   -1.0000        0
        0        0    1.0000

        >> Rx*Ry*Rz
        ans =
        -0.9397   -0.0000   -0.3420
        0.2620    0.6428   -0.7198
        0.2198   -0.7660   -0.6040
       -->
        <hdsoMod idNum="180" NumElts="16">
         -0.939692620785908                0  -0.342020143325669  0
         0.262002630229385   0.642787609686539  -0.719846310392954  0
         0.219846310392954  -0.766044443118978  -0.604022773555054  10.5
         0                 0                 0                 1
        </hdsoMod>
       </amatrix>
       <DHline>
        <hdsoMod idNum="181" NumElts="4">
         <!-- 0 0 (-90/180)*pi (pi/180) -->
         <!-- In reality, the alpha here changes due to the loose connection between the ThC1 and ThC2
         motors; it's a little more like 80 degrees when the leg is being held out in the air-->
         1      0     -1.570796326794897      0.017453292519943
        </hdsoMod>
       </DHline>
      </geometry>
      <state name="TC1_protractionDEP" idNum="201">
<!--      <plotstyle colorspec="0    0.5725    0.2549"/>  -->
<!--      <plotstyle colorspec="0.0039 0.7137 0.3020"/>-->
       <plotstyle colorspec="0    0.5804    0.3882"/>
       <dwell_time>
       <hdsoMod idNum="194" NumElts="0"/>
       </dwell_time>
       <muscle_activation muscle="401" activation_function="constant"
            activation_parameters="121">
            <hdsoMod idNum="121" NumElts="1" time="0">0.05</hdsoMod>
       </muscle_activation>
<!--bilinear activation for 402 here, if analogous to stick insect -->
       <muscle_activation muscle="402" activation_function="constant"
            activation_parameters="122">
            <hdsoMod idNum="122" NumElts="1" time="0">0.15</hdsoMod>
       </muscle_activation>
```

```
      <transition_rule representation_type="text" boolstring="307 &gt; 105">
            <hdsoMod idNum="161" NumElts="0" />
            <hdsoMod idNum="105" NumElts="1">120</hdsoMod>
      </transition_rule>
    </state>
    <state name="TC1_retractionLEV" idNum="202">
<!--      <plotstyle colorspec="0.3098    0.8667    0.5765"/>-->
      <plotstyle colorspec="0.6627 1.000 0"/>
 <dwell_time>
<hdsoMod idNum="194" NumElts="0"/>
</dwell_time>
<!--linear activation for 401 here, if analogous to stick insect -->
      <muscle_activation muscle="401" activation_function="constant"
            activation_parameters="123">
            <hdsoMod idNum="123" NumElts="1">0.25</hdsoMod>
      </muscle_activation>
      <muscle_activation muscle="402" activation_function="constant"
            activation_parameters="124">
            <hdsoMod idNum="124" NumElts="1" time="0">0.05</hdsoMod>
      </muscle_activation>
      <transition_rule representation_type="text"
       boolstring="(307 &lt; 103)|(305 &lt; 104)">
            <hdsoMod idNum="162" NumElts="0" />
            <hdsoMod idNum="103" NumElts="1">60</hdsoMod>
            <hdsoMod idNum="104" NumElts="1">70</hdsoMod>
      </transition_rule>
    </state>
    <sense_organ name="TC1_sense_organ" idNum="501">
      <description>The TC sense organ is involved in sensing the angle of the
      Thorax-Coxa joint.</description>
      <hdsoMod idNum="301" NumElts="0"/>
    </sense_organ>
    <aimotor name="Thorax-Coxa1_motor" idNum="601" serialID="0" dir="1"
      angleCal1="144" angleCal2="1.529" modelnum="1001">
      <description>The range for theta=0 for this motor is approximately 130-158 in raw
      A/D units, as of 06/04/09- see pp. 31-32 in Brandon L. Rutter Biorobotics lab notebook
#3</description>
      <forceCommand>
        <hdsoMod idNum="201" NumElts="0"/>
      </forceCommand>
      <positionCommand>
        <hdsoMod idNum="202" NumElts="0"/>
      </positionCommand>
    </aimotor>
  </joint>
  <joint name="Thorax-Coxa2" idNum="2" parent="1" start_state_idNum="201"
control_mode="position">
    <plotstyle linespec="bx"/>
    <geometry>
      <description>
        The d value of 6.5 in this DHline can cause things to look a little wierd sometimes; it may
        be useful to set d=0 for debugging.  This is the jump from the upper to the lower plane of the leg;
        it might be sensible to make this the "length" of of ThC1, with the zero-theta X for that joint pointing
        down instead of back.
      </description>
      <DHline>
```

```
      <hdsoMod idNum="182" NumElts="4">
       <!-- 8.9 6.5  0  (pi/180) -->
       8.9  6.5   0   0.017453292519943
      </hdsoMod>
     </DHline>
    </geometry>
    <aimotor name="Thorax-Coxa2_motor" idNum="602" serialID="1" dir="-1"
     angleCal1="131.5" angleCal2="-1.529" modelnum="701">
     <description>The range for theta=0 for this motor is approximately 122-141 in raw
      A/D units, as of 06/04/09- see pp. 31-32 in Brandon L. Rutter Biorobotics lab notebook
#3</description>
     <forceCommand>
      <hdsoMod idNum="203" NumElts="0"/>
     </forceCommand>
     <positionCommand>
      <hdsoMod idNum="204" NumElts="1">128</hdsoMod>
     </positionCommand>
    </aimotor>
   </joint>
   <joint name="Coxa-Trochanter" idNum="3" parent="2" start_state_idNum="203">
    <plotstyle linespec="b"/>
    <muscle name="Femur_DepressorRET" idNum="403" Lnaught="85" r="-10"
          alpha="80" fmax="15">
     <function_spec type="constant" use="passive_stiffness">
      <hdsoMod idNum="190" NumElts="0"/>
     </function_spec>
     <function_spec type="linear" use="force-length">
      <hdsoMod idNum="173" NumElts="2">2 -1.5</hdsoMod>
     </function_spec>
     <function_spec type="constant" use="force-velocity">
      <hdsoMod idNum="191" NumElts="0"/>
     </function_spec>
    </muscle>
    <muscle name="Femur_LevatorPRO" idNum="404" Lnaught="75" r="2.5"
          alpha="60" fmax="20">
     <function_spec type="constant" use="passive_stiffness">
      <hdsoMod idNum="190" NumElts="0"/>
     </function_spec>
     <function_spec type="linear" use="force-length">
      <hdsoMod idNum="174" NumElts="2">2 -1.5</hdsoMod>
     </function_spec>
     <function_spec type="constant" use="force-velocity">
      <hdsoMod idNum="191" NumElts="0"/>
     </function_spec>
    </muscle>
    <geometry>
     <description>
      This description entry keeps the femur at the lower plane of the leg, even though the
      physical beam in the robot is at the upper plane.
     </description>
     <DHline>
      <hdsoMod idNum="183" NumElts="4">
       <!-- 9.7 0 (180/180)*pi (pi/180) -->
       9.7  0  3.141592653589793  0.017453292519943
      </hdsoMod>
     </DHline>
```

```
    </geometry>
    <state name="CTr_LevationPRO" idNum="203">
<!--      <plotstyle colorspec="0.6235    0.6667    0.9255"/>-->
      <plotstyle colorspec="0 0.8627 0.8627"/>
      <dwell_time>
        <hdsoMod idNum="194" NumElts="0"/>
      </dwell_time>
      <muscle_activation muscle="403" activation_function="constant"
            activation_parameters="125">
            <hdsoMod idNum="125" NumElts="1">0.01</hdsoMod>
      </muscle_activation>
      <muscle_activation muscle="404" activation_function="constant"
            activation_parameters="126">
            <hdsoMod idNum="126" NumElts="1">0.85</hdsoMod>
      </muscle_activation>
      <transition_rule representation_type="text"
            boolstring="(302 &lt; 101)">
            <hdsoMod idNum="163" NumElts="0" />
            <hdsoMod idNum="101" NumElts="1">0</hdsoMod>
      </transition_rule>
    </state>
    <state name="CTr_DepressionRET" idNum="204">
<!--      <plotstyle colorspec="0    0.5608    0.8784"/>-->
      <plotstyle colorspec="0 0.2980 0.6392"/>
      <dwell_time>
<hdsoMod idNum="194" NumElts="0"/>
</dwell_time>
      <muscle_activation muscle="403" activation_function="constant"
            activation_parameters="127">
            <hdsoMod idNum="127" NumElts="1">0.55</hdsoMod>
      </muscle_activation>
      <muscle_activation muscle="404" activation_function="constant"
            activation_parameters="128">
            <hdsoMod idNum="128" NumElts="1">0.1</hdsoMod>
      </muscle_activation>
      <transition_rule representation_type="text" boolstring="302 &gt; 102">
            <hdsoMod idNum="164" NumElts="0" />
            <hdsoMod idNum="102" NumElts="1">0.01</hdsoMod>
      </transition_rule>
    </state>
    <sense_organ name="Trochanter_CS" idNum="502">
      <description>The trochanteral canpaniform sensilla are load sensors critical
      to agile behavior and adaptation to varying conditions.</description>
      <hdsoMod idNum="306" NumElts="0"/>
    </sense_organ>
    <aimotor name="Coxa-Trochanter_motor" idNum="603" serialID="2" dir="1"
      angleCal1="-12.11" angleCal2="1.529" modelnum="701">
      <description>The range for theta=0 for this motor is approximately 119-132 in raw
        A/D units, as of 06/04/09- see pp. 31-32 in Brandon L. Rutter Biorobotics lab notebook
#3</description>
    </aimotor>
  </joint>
  <joint name="Femur-Tibia" idNum="4" parent="3" start_state_idNum="205">
    <plotstyle linespec="g"/>
    <muscle name="Tibia_Flexor" idNum="405" Lnaught="95" r="15"
          alpha="100" fmax="14">
```

```
      <function_spec type="constant" use="passive_stiffness">
        <hdsoMod idNum="190" NumElts="0"/>
      </function_spec>
      <function_spec type="linear" use="force-length">
        <hdsoMod idNum="175" NumElts="2">2 -1.5</hdsoMod>
      </function_spec>
      <function_spec type="constant" use="force-velocity">
        <hdsoMod idNum="191" NumElts="0"/>
      </function_spec>
    </muscle>
    <muscle name="Tibia_Extensor" idNum="406" Lnaught="90" r="-10"
          alpha="80" fmax="20">
      <function_spec type="constant" use="passive_stiffness">
        <hdsoMod idNum="190" NumElts="0"/>
      </function_spec>
      <function_spec type="linear" use="force-length">
        <hdsoMod idNum="176" NumElts="2">2 -1.5</hdsoMod>
      </function_spec>
      <function_spec type="constant" use="force-velocity">
        <hdsoMod idNum="191" NumElts="0"/>
      </function_spec>
    </muscle>
    <geometry>
      <DHline>
        <hdsoMod idNum="184" NumElts="4">
          <!-- 8.5 0 0 (pi/180) -->
          8.5 0 0 0.017453292519943
        </hdsoMod>
      </DHline>
    </geometry>
    <state name="FT_Extension" idNum="205">
<!--      <plotstyle colorspec="0.9647    0.3686    0.3922"/>-->
      <plotstyle colorspec="1 0.4 0"/>
<dwell_time>
<hdsoMod idNum="194" NumElts="0"/>
</dwell_time>
      <muscle_activation muscle="405" activation_function="constant"
            activation_parameters="129">
            <hdsoMod idNum="129" NumElts="1">0.09</hdsoMod>
      </muscle_activation>
      <muscle_activation muscle="406" activation_function="constant"
            activation_parameters="130">
            <hdsoMod idNum="130" NumElts="1">1.0</hdsoMod>
      </muscle_activation>
      <transition_rule representation_type="text" boolstring="(302 &amp;gt; 106)&amp;amp;(307 &amp;gt; 107)">
            <hdsoMod idNum="165" NumElts="0" />
            <hdsoMod idNum="106" NumElts="1">0.01</hdsoMod>
            <hdsoMod idNum="107" NumElts="1">90</hdsoMod>
      </transition_rule>
    </state>
    <state name="FT_Flexion" idNum="206">
<!--      <plotstyle colorspec="0.8627    0.1686    0.0980"/>-->
      <plotstyle colorspec="0.7098 0 0.2196"/>
<dwell_time>
<hdsoMod idNum="194" NumElts="0"/>
</dwell_time>
```

```
    <muscle_activation muscle="405" activation_function="constant"
          activation_parameters="131">
          <hdsoMod idNum="131" NumElts="1">0.15</hdsoMod>
    </muscle_activation>
    <muscle_activation muscle="406" activation_function="constant"
          activation_parameters="132">
          <hdsoMod idNum="132" NumElts="1" time="0">0.001</hdsoMod>
    </muscle_activation>
    <transition_rule representation_type="text" boolstring="(302&amp;lt;108)|(307&amp;lt;=109)">
          <hdsoMod idNum="166" NumElts="0" />
          <hdsoMod idNum="108" NumElts="1">0.0</hdsoMod>
          <hdsoMod idNum="109" NumElts="1">60</hdsoMod>
    </transition_rule>
   </state>
   <sense_organ name="Femur_CS" idNum="503">
    <description>The femoral canpaniform sensilla are load sensors; used similarly
    to the trochanteral CS, as far as we know/this model knows.</description>
    <hdsoMod idNum="302" NumElts="0"/>
    <hdsoMod idNum="306" NumElts="0"/>
   </sense_organ>
   <sense_organ name="Femur_CO" idNum="504">
    <description>The Femoral Chordotonal Organ is a continuous angle-sensing
    organ which operates as a specialized apodeme/muscle in the femur, sensing
    the motion of the Femur-Tibia joint.</description>
    <hdsoMod idNum="307" NumElts="0"/>
   </sense_organ>
   <aimotor name="Femur_Tibia_motor" idNum="604" serialID="3" dir="1"
    angleCal1="-12.11" angleCal2="1.529" modelnum="701">
    <description>The range for theta=0 for this motor is approximately 119-132 in raw
     A/D units, as of 06/04/09- see pp. 31-32 in Brandon L. Rutter Biorobotics lab notebook
#3</description>
   </aimotor>
  </joint>
  <endpoint>
   <joint name="Tibia-Tarsus" idNum="5" parent="4">
    <description>The Tibia-Tarsus joint is in this model only the terminal endpoint.
     No actual tarsus is currently modeled.</description>
    <plotstyle linespec="r"/>
    <geometry />
   </joint>
  </endpoint>

</jointlist>
```

3.2.2 parameters

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE hdsolist SYSTEM "/usr/local/src/romod_dtds/hdso.dtd">
<!-- <!DOCTYPE hdsolist SYSTEM "http://roach.biol.cwru.edu/romod-dtds/hdso.dtd"> -->

<hdsolist name="lin_roachmiddle_params" DataSetType="parameterList">

<!-- global engine parameters -->
  <hdso idNum="0" name="motor_speed" datatype="int" numdims="1">
```

```
    <dimsize dimNr="1">1</dimsize>
    <range type="valid">
      <bound type="upper">4</bound>
      <bound type="lower">0</bound>
      <quantum>1</quantum>
    </range>
    <range type="suggested">
      <bound type="upper">4</bound>
      <bound type="lower">0</bound>
      <quantum>1</quantum>
    </range>
    <initial_data>3</initial_data>
  </hdso>
  <hdso idNum="1" name="motor_control_delay_ns" datatype="llint" numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="suggested">
      <bound type="upper">1000000000</bound>
      <bound type="lower">0</bound>
      <quantum>10000000</quantum>
    </range>
    <initial_data>0</initial_data>
  </hdso>
  <hdso idNum="2" name="statetrans_period_ns" datatype="llint" numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="suggested">
      <bound type="upper">1000000000</bound>
      <bound type="lower">0</bound>
      <quantum>1000000</quantum>
    </range>
<!-- corresponding to a hypothesized sensory loop delay of about 7.9 ms in the animal -->
    <initial_data>25000000</initial_data>
  </hdso>

<!-- state transition parameters -->
  <hdso idNum="101" name="CTr_lev_TC1_load" datatype="float" numdims="1">
      <dimsize dimNr="1">1</dimsize>
      <range type="suggested">
        <bound type="upper">2</bound>
        <bound type="lower">-3</bound>
        <quantum>0.1</quantum>
      </range>
      <range type="valid">
        <bound type="upper">20</bound>
        <bound type="lower">-20</bound>
      </range>
  </hdso>
  <hdso idNum="102" name="CTr_dep_TC1_load" datatype="float" numdims="1">
      <dimsize dimNr="1">1</dimsize>
      <range type="suggested">
        <bound type="upper">3</bound>
        <bound type="lower">-2</bound>
        <quantum>0.1</quantum>
      </range>
      <range type="valid">
        <bound type="upper">20</bound>
        <bound type="lower">-20</bound>
```

```
    </range>
</hdso>
<hdso idNum="103" name="TC1_retLEV_FTang" numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="suggested">
      <bound type="upper">90</bound>
      <bound type="lower">50</bound>
      <quantum>1</quantum>
    </range>
    <range type="valid">
      <bound type="upper">170</bound>
      <bound type="lower">-170</bound>
    </range>
</hdso>
<hdso idNum="104" name="TC1_retLEV_CTrang" numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="suggested">
      <bound type="upper">100</bound>
      <bound type="lower">60</bound>
      <quantum>1</quantum>
    </range>
    <range type="valid">
      <bound type="upper">180</bound>
      <bound type="lower">-180</bound>
    </range>
</hdso>
<hdso idNum="105" name="TC1_proDEP_FTang" numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="suggested">
      <bound type="upper">130</bound>
      <bound type="lower">90</bound>
      <quantum>1</quantum>
    </range>
    <range type="valid">
      <bound type="upper">170</bound>
      <bound type="lower">-170</bound>
    </range>
</hdso>
<hdso idNum="106" name="FT_ext_TC1_load" numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="suggested">
      <bound type="upper">3</bound>
      <bound type="lower">-3</bound>
      <quantum>0.1</quantum>
    </range>
    <range type="valid">
      <bound type="upper">20</bound>
      <bound type="lower">-20</bound>
    </range>
</hdso>
<hdso idNum="107" numdims="1" name="FT_ext_FTang">
    <dimsize dimNr="1">1</dimsize>
    <range type="suggested">
      <bound type="upper">110</bound>
      <bound type="lower">70</bound>
      <quantum>1</quantum>
```

```
      </range>
      <range type="valid">
        <bound type="upper">170</bound>
        <bound type="lower">-170</bound>
      </range>
    </hdso>
    <hdso idNum="108" name="FT_flx_TC1_load" numdims="1">
      <dimsize dimNr="1">1</dimsize>
      <range type="suggested">
        <bound type="upper">3</bound>
        <bound type="lower">-3</bound>
        <quantum>0.1</quantum>
      </range>
      <range type="valid">
        <bound type="upper">20</bound>
        <bound type="lower">-20</bound>
      </range>
    </hdso>
    <hdso idNum="109" name="FT_flx_FTang" numdims="1">
      <dimsize dimNr="1">1</dimsize>
      <range type="suggested">
        <bound type="upper">80</bound>
        <bound type="lower">40</bound>
        <quantum>1</quantum>
      </range>
      <range type="valid">
        <bound type="upper">170</bound>
        <bound type="lower">-170</bound>
      </range>
    </hdso>

  <!-- MUSCLE ACTIVATION PARAMETERS -->
    <hdso idNum="121" name="TC1_protractionDEP_retractor_activation" numdims="1">
      <dimsize dimNr="1">1</dimsize>
      <range type="suggested">
        <bound type="upper">1</bound>
        <bound type="lower">0</bound>
        <quantum>0.05</quantum>
      </range>
      <range type="valid">
        <bound type="upper">2</bound>
        <bound type="lower">0</bound>
      </range>
    </hdso>
    <hdso idNum="122" name="TC1_protractionDEP_protractor_activation" numdims="1">
      <dimsize dimNr="1">1</dimsize>
      <range type="suggested">
        <bound type="upper">1</bound>
        <bound type="lower">0</bound>
        <quantum>0.05</quantum>
      </range>
      <range type="valid">
        <bound type="upper">2</bound>
        <bound type="lower">0</bound>
      </range>
    </hdso>
```

```
<hdso idNum="123" name="TC1_retractionLEV_retractor_activation" numdims="1">
 <dimsize dimNr="1">1</dimsize>
 <range type="suggested">
  <bound type="upper">1</bound>
    <bound type="lower">0</bound>
    <quantum>0.05</quantum>
   </range>
   <range type="valid">
    <bound type="upper">2</bound>
    <bound type="lower">0</bound>
   </range>
</hdso>
<hdso idNum="124" name="TC1_retractionLEV_protractor_activation" numdims="1">
 <dimsize dimNr="1">1</dimsize>
 <range type="suggested">
  <bound type="upper">1</bound>
    <bound type="lower">0</bound>
    <quantum>0.05</quantum>
   </range>
   <range type="valid">
    <bound type="upper">2</bound>
    <bound type="lower">0</bound>
   </range>
</hdso>

<hdso idNum="125" name="CTr_levationPRO_depressor_activation" numdims="1">
 <dimsize dimNr="1">1</dimsize>
 <range type="suggested">
  <bound type="upper">1</bound>
    <bound type="lower">0</bound>
    <quantum>0.05</quantum>
   </range>
   <range type="valid">
    <bound type="upper">2</bound>
    <bound type="lower">0</bound>
   </range>
</hdso>
<hdso idNum="126" name="CTr_levationPRO_levator_activation" numdims="1">
 <dimsize dimNr="1">1</dimsize>
 <range type="suggested">
  <bound type="upper">1</bound>
    <bound type="lower">0</bound>
    <quantum>0.05</quantum>
   </range>
   <range type="valid">
    <bound type="upper">2</bound>
    <bound type="lower">0</bound>
   </range>
</hdso>
<hdso idNum="127" name="CTr_depressionRET_depressor_activation" numdims="1">
 <dimsize dimNr="1">1</dimsize>
 <range type="suggested">
  <bound type="upper">1</bound>
    <bound type="lower">0</bound>
    <quantum>0.05</quantum>
   </range>
```

```
    <range type="valid">
      <bound type="upper">2</bound>
      <bound type="lower">0</bound>
    </range>
</hdso>
<hdso idNum="128" name="CTr_depressionRET_levator_activation" numdims="1">
 <dimsize dimNr="1">1</dimsize>
 <range type="suggested">
   <bound type="upper">1</bound>
     <bound type="lower">0</bound>
     <quantum>0.05</quantum>
   </range>
   <range type="valid">
     <bound type="upper">2</bound>
     <bound type="lower">0</bound>
   </range>
</hdso>

<hdso idNum="129" name="FT_extension_flexor_activation" numdims="1">
 <dimsize dimNr="1">1</dimsize>
 <range type="suggested">
   <bound type="upper">1</bound>
     <bound type="lower">0</bound>
     <quantum>0.05</quantum>
   </range>
   <range type="valid">
     <bound type="upper">2</bound>
     <bound type="lower">0</bound>
   </range>
</hdso>
<hdso idNum="130" name="FT_extension_extensor_activation" numdims="1">
 <dimsize dimNr="1">1</dimsize>
 <range type="suggested">
   <bound type="upper">1</bound>
     <bound type="lower">0</bound>
     <quantum>0.05</quantum>
   </range>
   <range type="valid">
     <bound type="upper">2</bound>
     <bound type="lower">0</bound>
   </range>
</hdso>
<hdso idNum="131" name="FT_flexion_flexor_activation" numdims="1">
 <dimsize dimNr="1">1</dimsize>
 <range type="suggested">
   <bound type="upper">1</bound>
     <bound type="lower">0</bound>
     <quantum>0.05</quantum>
   </range>
   <range type="valid">
     <bound type="upper">2</bound>
     <bound type="lower">0</bound>
   </range>
</hdso>
<hdso idNum="132" name="FT_flexion_extensor_activation" numdims="1">
 <dimsize dimNr="1">1</dimsize>
```

```
    <range type="suggested">
     <bound type="upper">1</bound>
       <bound type="lower">0</bound>
       <quantum>0.05</quantum>
     </range>
     <range type="valid">
       <bound type="upper">2</bound>
       <bound type="lower">0</bound>
     </range>
  </hdso>

<!-- STATE TRANSITION RULE PARAMETERS -->
  <hdso idNum="161" name="Enter_TC1_ProtractionDEP" abstractType="boolstring" datatype="int"
xmltype="char" numdims="2">
    <dimsize dimNr="1">2</dimsize>
    <dimsize dimNr="2">30</dimsize>
  </hdso>
  <hdso idNum="162" name="Enter_TC1_RetractionLEV" abstractType="boolstring" datatype="int"
xmltype="char" numdims="2">
    <dimsize dimNr="1">2</dimsize>
    <dimsize dimNr="2">30</dimsize>
  </hdso>
  <hdso idNum="163" name="Enter_CTr_LevationPRO" abstractType="boolstring" datatype="int"
xmltype="char" numdims="2">
    <dimsize dimNr="1">2</dimsize>
    <dimsize dimNr="2">30</dimsize>
  </hdso>
  <hdso idNum="164" name="Enter_CTr_DepressionRET" abstractType="boolstring" datatype="int"
xmltype="char" numdims="2">
    <dimsize dimNr="1">2</dimsize>
    <dimsize dimNr="2">30</dimsize>
  </hdso>
  <hdso idNum="165" name="Enter_FT_Extension" abstractType="boolstring" datatype="int"
xmltype="char" numdims="2">
    <dimsize dimNr="1">2</dimsize>
    <dimsize dimNr="2">30</dimsize>
  </hdso>
  <hdso idNum="166" name="Enter_FT_Flexion" abstractType="boolstring" datatype="int"
xmltype="char" numdims="2">
    <dimsize dimNr="1">2</dimsize>
    <dimsize dimNr="2">30</dimsize>
  </hdso>

<!-- MUSCLE MODEL PARAMETERS -->
  <hdso idNum="171" name="ThC1_RemotorLEV_F-L_Params" numdims="1">
    <dimsize dimNr="1">2</dimsize>
  </hdso>
  <hdso idNum="172" name="ThC1_PromotorDEP_F-L_Params" numdims="1">
    <dimsize dimNr="1">2</dimsize>
  </hdso>
  <hdso idNum="173" name="CTr_DepressorRET_F-L_Params" numdims="1">
    <dimsize dimNr="1">2</dimsize>
  </hdso>
  <hdso idNum="174" name="CTr_LevatorPRO_F-L_Params" numdims="1">
    <dimsize dimNr="1">2</dimsize>
  </hdso>
```

```
<hdso idNum="175" name="Tibia_Flexor_F-L_Params" numdims="1">
 <dimsize dimNr="1">2</dimsize>
</hdso>
<hdso idNum="176" name="Tibia_Extensor_F-L_Params" numdims="1">
 <dimsize dimNr="1">2</dimsize>
</hdso>


<!-- JOINT GEOMETRY PARAMETERS -->
<hdso idNum="180" name="ThC1_Amatrix" numdims="2">
 <dimsize dimNr="1">4</dimsize>
 <dimsize dimNr="2">4</dimsize>
 <!-- Though the rotational portion of the Amatrix has a range -1 to 1, this
  is not true for the translational portion, and we do not have per-element range
  definitions.  So, no ranges-->
</hdso>
<hdso idNum="181" name="ThC1_DHline" numdims="1" units="cm">
 <dimsize dimNr="1">4</dimsize>
</hdso>
<hdso idNum="182" name="ThC2_DHline" numdims="1" units="cm">
 <dimsize dimNr="1">4</dimsize>
</hdso>
<hdso idNum="183" name="CTr_DHline" numdims="1" units="cm">
 <dimsize dimNr="1">4</dimsize>
</hdso>
<hdso idNum="184" name="FTi_DHline" numdims="1" units="cm">
 <dimsize dimNr="1">4</dimsize>
</hdso>

<!-- Motor/joint command parameters.  TODO set/derive better suggested position ranges -->
<hdso idNum="201" name="ThC1_forceCommand" numdims="1" units="Raw_Implicit">
 <dimsize dimNr="1">1</dimsize>
 <range type="valid">
  <bound type="lower">0</bound>
  <bound type="upper">254</bound>
  <quantum>1</quantum>
 </range>
 <range type="suggested">
  <bound type="lower">0</bound>
  <bound type="upper">15</bound>
  <quantum>1</quantum>
 </range>
</hdso><hdso idNum="202" name="ThC1_positionCommand" numdims="1" units="Raw_Implicit">
 <dimsize dimNr="1">1</dimsize>
 <range type="valid">
  <bound type="lower">0</bound>
  <bound type="upper">254</bound>
  <quantum>1</quantum>
 </range>
 <range type="suggested">
  <bound type="lower">0</bound>
  <bound type="upper">254</bound>
  <quantum>1</quantum>
 </range>
</hdso><hdso idNum="203" name="ThC2_forceCommand" numdims="1" units="Raw_Implicit">
 <dimsize dimNr="1">1</dimsize>
```

```
    <range type="valid">
      <bound type="lower">0</bound>
      <bound type="upper">254</bound>
      <quantum>1</quantum>
    </range>
    <range type="suggested">
      <bound type="lower">0</bound>
      <bound type="upper">15</bound>
      <quantum>1</quantum>
    </range>
  </hdso><hdso idNum="204" name="ThC2_positionCommand" numdims="1" units="Raw_Implicit">
    <dimsize dimNr="1">1</dimsize>
    <range type="valid">
      <bound type="lower">0</bound>
      <bound type="upper">254</bound>
      <quantum>1</quantum>
    </range>
    <range type="suggested">
      <bound type="lower">0</bound>
      <bound type="upper">254</bound>
      <quantum>1</quantum>
    </range> </hdso><hdso idNum="205" name="CTr_forceCommand" numdims="1"
units="Raw_Implicit">
    <dimsize dimNr="1">1</dimsize>
    <range type="valid">
      <bound type="lower">0</bound>
      <bound type="upper">254</bound>
      <quantum>1</quantum>
    </range>
    <range type="suggested">
      <bound type="lower">0</bound>
      <bound type="upper">15</bound>
      <quantum>1</quantum>
    </range>
  </hdso><hdso idNum="206" name="CTr_positionCommand" numdims="1" units="Raw_Implicit">
    <dimsize dimNr="1">1</dimsize>
    <range type="valid">
      <bound type="lower">0</bound>
      <bound type="upper">254</bound>
      <quantum>1</quantum>
    </range>
    <range type="suggested">
      <bound type="lower">0</bound>
      <bound type="upper">254</bound>
      <quantum>1</quantum>
    </range>  </hdso><hdso idNum="207" name="FTi_forceCommand" numdims="1"
units="Raw_Implicit">
    <dimsize dimNr="1">1</dimsize>
    <range type="valid">
      <bound type="lower">0</bound>
      <bound type="upper">254</bound>
      <quantum>1</quantum>
    </range>
    <range type="suggested">
      <bound type="lower">0</bound>
      <bound type="upper">15</bound>
```

```
    <quantum>1</quantum>
   </range>
  </hdso><hdso idNum="208" name="FTi_positionCommand" numdims="1" units="Raw_Implicit">
   <dimsize dimNr="1">1</dimsize>
   <range type="valid">
    <bound type="lower">0</bound>
    <bound type="upper">254</bound>
    <quantum>1</quantum>
   </range>
   <range type="suggested">
    <bound type="lower">0</bound>
    <bound type="upper">254</bound>
    <quantum>1</quantum>
   </range> </hdso>

<!-- "constants" -->
  <hdso idNum="190" name="zero_float_const" abstractType="constant" datatype="float" numdims="1">
   <dimsize dimNr="1">1</dimsize>
   <initial_data>0</initial_data>
  </hdso>
  <hdso idNum="191" name="one_float_const" abstractType="constant" datatype="float" numdims="1">
   <dimsize dimNr="1">1</dimsize>
   <initial_data>1</initial_data>
  </hdso>
  <hdso idNum="192" name="zero_int_const" abstractType="constant" datatype="int" numdims="1">
   <dimsize dimNr="1">1</dimsize>
   <initial_data>0</initial_data>
  </hdso>
  <hdso idNum="193" name="one_int_const" abstractType="constant" datatype="int" numdims="1">
   <dimsize dimNr="1">1</dimsize>
   <initial_data>1</initial_data>
  </hdso>
  <hdso idNum="194" name="zero_llint_const" abstractType="constant" datatype="llint" numdims="1">
   <dimsize dimNr="1">1</dimsize>
   <initial_data>0</initial_data>
  </hdso>
  <hdso idNum="195" name="one_llint_const" abstractType="constant" datatype="llint" numdims="1">
   <dimsize dimNr="1">1</dimsize>
   <initial_data>1</initial_data>
  </hdso>

</hdsolist>
```

3.2.3 sensors

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE hdsolist SYSTEM "/usr/local/src/romod_dtds/hdso.dtd">
<!-- <!DOCTYPE hdsolist SYSTEM "http://roach.biol.cwru.edu/romod-dtds/hdso.dtd"> -->

<hdsolist name="lin_roachmiddle_sensors" DataSetType="sensorList">
  <description>Sensor list for lin_roachmiddle (and variant behaviors). It should be noted that if the
joints.xml file
  does not have sensorspecs for the motor sensors, if there are N motors then the first N+2
  entries in this file must be associated with the angle and current (load) of those motors.
```

```
</description>
<hdso idNum="301" abstractType="sensor" name="TC1_angle"
    overflowCorrection="1" fullname="Thorax-Coxa 1st DOF Angle"
    datatype="float" numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="normal">
      <bound type="upper">255</bound>
      <bound type="lower">0</bound>
      <quantum>1</quantum>
    </range>
</hdso>
<hdso idNum="302" abstractType="sensor" name="TC1_current"
    fullname="Thorax-Coxa 1st DOF Current (load)" datatype="float"
    numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="normal">
      <bound type="upper">10</bound>
      <bound type="lower">-10</bound>
    </range>
</hdso>
<hdso idNum="303" abstractType="sensor" name="TC2_angle"
    overflowCorrection="1" fullname="Thorax-Coxa 2nd DOF Angle"
    datatype="float" numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="normal">
      <bound type="upper">255</bound>
      <bound type="lower">0</bound>
      <quantum>1</quantum>
    </range>
</hdso>
<hdso idNum="304" abstractType="sensor" name="TC2_current"
    fullname="Thorax-Coxa 2nd DOF Current (load)" datatype="float"
    numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="normal">
      <bound type="upper">10</bound>
      <bound type="lower">-10</bound>
    </range>
</hdso>
<hdso idNum="305" abstractType="sensor" name="CTr_angle"
    overflowCorrection="1" fullname="Coxa-Trochanter Angle"
    datatype="float" numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="normal">
      <bound type="upper">255</bound>
      <bound type="lower">0</bound>
      <quantum>1</quantum>
    </range>
</hdso>
<hdso idNum="306" abstractType="sensor" name="CTr_current"
    fullname="Coxa-Troachanter Current (load)" datatype="float"
    numdims="1">
    <dimsize dimNr="1">1</dimsize>
    <range type="normal">
      <bound type="upper">10</bound>
      <bound type="lower">-10</bound>
```

```
      </range>
  </hdso>
  <hdso idNum="307" abstractType="sensor" name="FT_angle"
      overflowCorrection="1" fullname="Femur-Tibia Angle"
      datatype="float" numdims="1">
      <dimsize dimNr="1">1</dimsize>
      <range type="normal">
        <bound type="upper">255</bound>
        <bound type="lower">0</bound>
        <quantum>1</quantum>
      </range>
  </hdso>
  <hdso idNum="308" abstractType="sensor" name="FT_current"
      fullname="Femur-Tibia Current (load)" datatype="float"
      numdims="1">
      <dimsize dimNr="1">1</dimsize>
      <range type="normal">
        <bound type="upper">10</bound>
        <bound type="lower">-10</bound>
      </range>
  </hdso>
  <hdso idNum="20" name="engine_start_time" numdims="1" datatype="llint"
    abstractType="sensor">
    <description>This is the global sensor HDSO for the engine start time; should be set  by the
      engine via script to facilitate logging every time the engine is started.  Script times are measured with
respect
      to this time. Initial value of -1 indicates that the engine has not yet started.</description>
    <dimsize dimNr="1">1</dimsize>
    <range type="normal">
      <bound type="lower">0</bound>
    </range>
    <initial_data>-1</initial_data>
  </hdso>
  <hdso idNum="21" name="engine_stop_time" numdims="1" datatype="llint"
    abstractType="sensor">
    <description>This is the global sensor HDSO for the engine stop time; should be set
      by the engine via script to facilitate logging every time the engine is stopped.  Not actually as necessary
as
      engine start time, but included for completeness.</description>
    <dimsize dimNr="1">1</dimsize>
    <range type="normal">
      <bound type="lower">0</bound>
    </range>
    <initial_data>-1</initial_data>
  </hdso>
</hdsolist>
```

4 Example script

The following hdsoscript is the experimental script for “Experiment A”, which starts in forward stepping, goes to inside-turn-forward behavior, and then to inside-turn-backward. This is the script from which all of the other scripts for the transition experiments were derived.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hdsoscript SYSTEM "/usr/local/src/romod_dtds/hdsoscript.dtd">
<!-- <!DOCTYPE hdsoscript SYSTEM "http://roach.biol.cwru.edu/romod-dtds/hdsoscript.dtd"> -->
<hdsoscript ScriptType="LegScript" name="linR_fwd-itf-itb">
 <description>Roch linear muscle model script for going from forward stepping to forward-flavored inside turning to backward-flavored inside turning.
  Based on 09-06-13_16-32-03 (forward), 09-06-13_18-58-03 (ITF), 09-06-13_20-08-36 (ITB); modified from those in an attempt
  to find the configurations which led to minimal changes between behaviors.
 </description>
 <hdsoMod idNum="0" name="motor_speed" NumElts="1" datatype="int" time="0">3 </hdsoMod>
 <hdsoMod idNum="1" name="motor_control_delay_ns" NumElts="1" datatype="llint" time="0">0
</hdsoMod>
 <hdsoMod idNum="2" name="statetrans_period_ns" NumElts="1" datatype="llint" time="0">25000000
</hdsoMod>
 <hdsoMod idNum="101" name="CTr_lev_TC1_load" NumElts="1" datatype="float" time="0">0
</hdsoMod> <!-- Changed from -0.1 -->
 <hdsoMod idNum="102" name="CTr_dep_TC1_load" NumElts="1" datatype="float" time="0">0.01
</hdsoMod>
 <hdsoMod idNum="103" name="TC1_retLEV_FTang" NumElts="1" datatype="float" time="0">60
</hdsoMod>
 <hdsoMod idNum="104" name="TC1_retLEV_CTrang" NumElts="1" datatype="float" time="0">70
</hdsoMod>
 <hdsoMod idNum="105" name="TC1_proDEP_FTang" NumElts="1" datatype="float" time="0">120
</hdsoMod>
 <hdsoMod idNum="106" name="FT_ext_TC1_load" NumElts="1" datatype="float" time="0">0.01
</hdsoMod>
 <hdsoMod idNum="107" name="FT_ext_FTang" NumElts="1" datatype="float" time="0">90
</hdsoMod>
 <hdsoMod idNum="108" name="FT_flx_TC1_load" NumElts="1" datatype="float" time="0">0
</hdsoMod>
 <hdsoMod idNum="109" name="FT_flx_FTang" NumElts="1" datatype="float" time="0">60
</hdsoMod>
 <hdsoMod idNum="121" name="TC1_protractionDEP_retractor_activation" NumElts="1"
datatype="float" time="0">0.05 </hdsoMod>
 <hdsoMod idNum="122" name="TC1_protractionDEP_protractor_activation" NumElts="1"
datatype="float" time="0">0.15 </hdsoMod>
 <hdsoMod idNum="123" name="TC1_retractionLEV_retractor_activation" NumElts="1"
datatype="float" time="0">0.25 </hdsoMod>
 <hdsoMod idNum="124" name="TC1_retractionLEV_protractor_activation" NumElts="1"
datatype="float" time="0">0.05 </hdsoMod>
```

```
  <hdsoMod idNum="125" name="CTr_levationPRO_depressor_activation" NumElts="1" datatype="float"
time="0">0.01 </hdsoMod>
  <hdsoMod idNum="126" name="CTr_levationPRO_levator_activation" NumElts="1" datatype="float"
time="0">0.85 </hdsoMod>
  <hdsoMod idNum="127" name="CTr_depressionRET_depressor_activation" NumElts="1"
datatype="float" time="0">0.55 </hdsoMod>
  <hdsoMod idNum="128" name="CTr_depressionRET_levator_activation" NumElts="1" datatype="float"
time="0">0.1 </hdsoMod>
  <hdsoMod idNum="129" name="FT_extension_flexor_activation" NumElts="1" datatype="float"
time="0">0.09 </hdsoMod>
  <hdsoMod idNum="130" name="FT_extension_extensor_activation" NumElts="1" datatype="float"
time="0">1 </hdsoMod>
  <hdsoMod idNum="131" name="FT_flexion_flexor_activation" NumElts="1" datatype="float"
time="0">0.15 </hdsoMod>
  <hdsoMod idNum="132" name="FT_flexion_extensor_activation" NumElts="1" datatype="float"
time="0">0.001 </hdsoMod>
  <hdsoMod idNum="161" name="Enter_TC1_ProtractionDEP" NumElts="60" datatype="int" time="0">0
1 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 307 11 105 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </hdsoMod>
  <hdsoMod idNum="162" name="Enter_TC1_RetractionLEV" NumElts="60" datatype="int" time="0">1
0 1 0 1 1 1 0 1 0 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 19 307 12 103 20 17 19 305 12 104
20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </hdsoMod>
  <hdsoMod idNum="163" name="Enter_CTr_LevationPRO" NumElts="60" datatype="int" time="0">1 0
1 0 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 19 302 12 101 20 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </hdsoMod>
  <hdsoMod idNum="164" name="Enter_CTr_DepressionRET" NumElts="60" datatype="int" time="0">0
1 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 302 11 102 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </hdsoMod>
  <hdsoMod idNum="165" name="Enter_FT_Extension" NumElts="60" datatype="int" time="0">1 0 1 0 1
1 1 0 1 0 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 19 302 11 106 20 16 19 307 11 107 20 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </hdsoMod>
  <hdsoMod idNum="166" name="Enter_FT_Flexion" NumElts="60" datatype="int" time="0">1 0 1 0 1 1
1 0 1 0 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 19 302 12 108 20 17 19 307 14 109 20 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </hdsoMod>
  <hdsoMod idNum="171" name="ThC1_RemotorLEV_F-L_Params" NumElts="2" datatype="float"
time="0">2 -1.5 </hdsoMod>
  <hdsoMod idNum="172" name="ThC1_PromotorDEP_F-L_Params" NumElts="2" datatype="float"
time="0">2 -1.5 </hdsoMod>
  <hdsoMod idNum="173" name="CTr_DepressorRET_F-L_Params" NumElts="2" datatype="float"
time="0">2 -1.5 </hdsoMod>
  <hdsoMod idNum="174" name="CTr_LevatorPRO_F-L_Params" NumElts="2" datatype="float"
time="0">2 -1.5 </hdsoMod>
  <hdsoMod idNum="175" name="Tibia_Flexor_F-L_Params" NumElts="2" datatype="float" time="0">2
-1.5 </hdsoMod>
  <hdsoMod idNum="176" name="Tibia_Extensor_F-L_Params" NumElts="2" datatype="float"
time="0">2 -1.5 </hdsoMod>


  <!-- Changes for Inside Turning, Forward -->
  <hdsoMod idNum="103" name="TC1_retLEV_FTang" NumElts="1" datatype="float" time="20">120
</hdsoMod>
  <hdsoMod idNum="104" name="TC1_retLEV_CTrang" NumElts="1" datatype="float" time="20">40
</hdsoMod> <!-- Not needed if it doesn appear in 162, Enter_TC_RetractionLEV -->
  <hdsoMod idNum="105" name="TC1_proDEP_FTang" NumElts="1" datatype="float" time="20">50
</hdsoMod>
  <hdsoMod idNum="106" name="FT_ext_TC1_load" NumElts="1" datatype="float" time="20">-0.05
```

```
</hdsoMod>  <!-- Needed?  - yes -->
  <hdsoMod idNum="107" name="FT_ext_FTang" NumElts="1" datatype="float" time="20">70
</hdsoMod>
<!--  <hdsoMod idNum="108" name="FT_flx_TC1_load" NumElts="1" datatype="float" time="20">0.01
</hdsoMod> -->  <!-- Needed? - probably not-->
  <hdsoMod idNum="109" name="FT_flx_FTang" NumElts="1" datatype="float" time="20">40
</hdsoMod>
  <hdsoMod idNum="161" name="Enter_TC1_ProtractionDEP" NumElts="60" datatype="int"
time="20">0 1 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 307 12 105 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </hdsoMod>
  <hdsoMod idNum="162" name="Enter_TC1_RetractionLEV" NumElts="60" datatype="int"
time="20">1 0 1 0 1 1 1 0 1 0 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 19 307 11 103 20 17 19
305 12 104 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </hdsoMod>
  <hdsoMod idNum="165" name="Enter_FT_Extension" NumElts="60" datatype="int" time="20">1 0 1 0
1 1 1 0 1 0 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 19 302 12 106 20 16 19 307 11 107 20 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </hdsoMod>
  <hdsoMod idNum="166" name="Enter_FT_Flexion" NumElts="60" datatype="int" time="20">1 0 1 0 1
1 1 0 1 0 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 19 302 11 108 20 17 19 307 14 109 20 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </hdsoMod>

  <!-- Further changes for Inside Turning, Backward -->
<!--  <hdsoMod idNum="101" name="CTr_lev_TC1_load" NumElts="1" datatype="float" time="40">0
</hdsoMod> -->  <!-- Needed? -yes, but could have it as 0 in prior behaviors-->
<!--  <hdsoMod idNum="102" name="CTr_dep_TC1_load" NumElts="1" datatype="float" time="40">0
</hdsoMod> -->  <!-- Needed? - probably not-->
  <hdsoMod idNum="104" name="TC1_retLEV_CTrang" NumElts="1" datatype="float" time="40">150
</hdsoMod>
<!--  <hdsoMod idNum="123" name="TC1_retractionLEV_retractor_activation" NumElts="1"
datatype="float" time="40">0.35 </hdsoMod> -->  <!-- Needed? probably not; maybe for backward
stepping- but might help for ITF too-->
  <hdsoMod idNum="126" name="CTr_levationPRO_levator_activation" NumElts="1" datatype="float"
time="40">0.4 </hdsoMod>
  <hdsoMod idNum="127" name="CTr_depressionRET_depressor_activation" NumElts="1"
datatype="float" time="40">0.1 </hdsoMod>
  <hdsoMod idNum="128" name="CTr_depressionRET_levator_activation" NumElts="1" datatype="float"
time="40">0.3 </hdsoMod>
<!--  <hdsoMod idNum="130" name="FT_extension_extensor_activation" NumElts="1" datatype="float"
time="40">2 </hdsoMod> -->   <!-- Needed? -probably not; maybe make a difference at higher motor
speeds-->
  <hdsoMod idNum="162" name="Enter_TC1_RetractionLEV" NumElts="60" datatype="int"
time="40">1 0 1 0 1 1 1 0 1 0 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 19 307 11 103 20 17 19
305 11 104 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </hdsoMod>
  <hdsoMod idNum="163" name="Enter_CTr_LevationPRO" NumElts="60" datatype="int" time="40">1
0 1 0 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 19 302 11 101 20 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </hdsoMod>
  <hdsoMod idNum="164" name="Enter_CTr_DepressionRET" NumElts="60" datatype="int"
time="40">0 1 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 302 12 102 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 </hdsoMod>

<!-- For the end, just time -->
  <hdsoMod idNum="190" name="zero_float_const" NumElts="1" datatype="float" time="60">0
</hdsoMod>
</hdsoscript>
```

10.3 The SCCILStoolbox MATLAB Data Analysis Package

THE SCCILSTOOLBOX MATLAB DATA ANALYSIS PACKAGE

by

BRANDON LEWIS RUTTER

Department of Mechanical and Aerospace Engineering

CASE WESTERN RESERVE UNIVERSITY

January 2010

Table of Contents

1 Introduction

If you're reading this, there's a good chance that you are working (or intending to work) on a system that you can either control or model using SCASM, and represent using SCCILS. Furthermore, you have gotten so far in it that you actually have, or can imagine having, actual data to interpret. Congratulations! This stuff is cool.

It is my hope that this package will be useful to others, and with a little modification more broadly so. SCASM arose from biological knowledge of legged systems, as did SCCILS. Because of this, there is a considerable amount of this specification that is specific to legged systems. Most of the tools described here could be extended to the general SCCILS system without too much work, and some of them should be generally applicable right off the bat.

Many of the "jl*" functions have precursor or supporting functions that are also part of the SCCILStoolbox repository. Most of these are not described here, but will be more easily applicable to data that does not have a SCCILS configuration associated with it. These were used for the early work done before SCCILS, described in the first part of the experimental chapter in my dissertation.

If you are working with a system that these tools don't fit exactly, please try to contact me (or possibly a new package maintainer) before writing your own software from scratch. There's a good chance that you could put the same or lesser effort into generalizing this software package, with benefits both to yourself and others. Please keep in mind, though, that I will be busy until I'm dead– so RTFM first.

Happy Interpreting,

Brandon Rutter

2 Installation

Get a copy of SCCILStoolbox. It is included in the romod repository; but if you want to download it separately you can use a Subversion client to check out the current version from:

svn://consolation.cwru.edu/releg/romod/trunk/SCCILStoolbox

Then, put everything in this directory on your MATLAB path.

3 SCCILS Configuration Representation and Parsing

The SCCILStoolbox representation of data is largely based around an object-oriented representation of the SCCILS jointlist configuration. This consists of five classes, each with a number of properties and methods. Properties are listed here; methods are listed when needed. Unfortunately, generating documentation with Doxygen and mtoc does not work well for MATLAB classdef files, so you'll have to look at the in-source comments for a complete description of the methods.

3.1 Representation

The SCCILS property fields below are copies of the SCCILS XML values of these fields; descriptions are given in the SCCILS specification. The current names of these classes are used, with the more general element description given in parentheses. For instance, a SCCILS "joint" is an implementation of the more general "action switching module" concept.

1. jointlist: analogous to the SCCILS jointlist (module list)
    a. SCCILS property fields:
        i. name
        ii. fullname
        iii. description
        iv. joints
        v. jointlists
    b. Additional fields:
        i. experiment_name : an identifier for the data contained in this jointlist, used in automatic figure labeling.
2. joint : analogous to the SCCILS joint (Action Switching Module)

a. SCCILS property fields:
    i. name
    ii. fullname
    iii. idnum
    iv. parent
    v. control_mode
    vi. description
    vii. amatrix
    viii. DHline
    ix. colorspec
    x. linespec
    xi. muscles
    xii. states
    xiii. motors
b. Additional fields:
    i. isendpoint : Boolean indicator of whether this is an “endpoint” joint or not. Endpoint joints are not expected to have muscles, states, or motors; but are used in plotting and holding coordinate data.
    ii. statedata : a vector containing the record of state changes for this joint
    iii. coords : a matrix of x-y-z coordinate values for the path of this joint through space
    iv. time_states, time_coords : time vectors for the above data
    v. statedata_unified, coords_unified, time_unified : data which has been unified to a common timebase.

3. muscle : analogous to the SCCILS muscle (simulated_actor)
    a. SCCILS property fields
        i. name
        ii. fullname
        iii. idnum
        iv. description
    b. Additional fields
        i. activation : vector containing the record of activation for this muscle
        ii. force : record of force
        iii. length : record of length
        iv. stime: time vector which applies to activation
        v. mtime: time which applies to force and length
        vi. activation_unified, time_unified : data which has been unified to the common timebase.
4. motor: analogous to the SCCILS aimotor (real_actor)
    a. SCCILS property fields
        i. name
        ii. idnum
    b. Additional fields. The motor contains data for angle and torque in three different ways, each with its own timebase.
        i. raw : raw A/D data from the motor, unprocessed
        ii. fast : corresponding data from the “sensors” – calibrated and perhaps filtered

iii. unified : data from the “fast” representation which has been unified to the common timebase

5. state: analogous to the SCCILS state (action). Note that the state does not contain any sensor- or action-related data of its own.
    a. SCCILS property fields
        i. name
        ii. fullname
        iii. idnum
        iv. linespec
        v. colorpsec

3.2 Parsing

The parsing of the SCCILS configuration should be engine-independent, therefore the parsing of the XML SCCILS configuration is done pretty much entirely in the object constructors themselves.

```
jlobj = ReadJointConfig(datadir)
```

ReadJointConfig reads in the jointlist.xml file in datadir, passes the root element of the document to the jointlist constructor, and returns the resulting jointlist. The jointlist constructor calls the constructors for its constituent classes when necessary. More advanced/complete configuration parsing would include the parameters and sensors configuration files, in order to have an accessible copy of the HDSO data store, but this is not yet implemented. It would be nice, though; one possibility in this case would be automatically plotting a variable sensory event detection threshold and the associated sensor value on the same graph.

It is also possible to enter configuration data in these objects manually via the command line, or in programs. This might be useful if you want to add or edit config information which is missing, incorrect, or not what you want, such as geometry or plot style elements.

The parsing of the data is, or can be, engine-specific- the data parser for the NeuRoMod engine is described in the next section.

4 Getting and Loading Data

4.1 Getting Data

If you don't already have some data, you'll need to get some; if you're using NeuRoMod, look in the NeuRoMod manual at the "Seeing and Gathering Data" section of "Using the GUI", or use enginecli in script mode, covered in "Command-line operation". Then, get it onto a computer where you've installed SCCILStoolbox, perhaps using the methods covered in chapter 3 of the NeuRoMod manual.

The data used in the plots below is from an experimental run demonstrating transitions between behaviors in a cockroach robot model.

4.2 Loading Data

Data from the NeuRoMod engine is loaded using LoadRomodEngineData:

```
jlobj = LoadRomodEngineData(datadir)
```

This function first calls ReadJointConfig, then uses loadlegdata to get the text file data from datadir into the workspace, then distributes that data into the appropriate fields in the jlobj jointlist object and its joint, motor, state and muscle children. This functionality is engine-specific and much of the information about how the data files map to these fields is hard-coded in this function.

5 Data Processing and Display

5.1 Basic Data Display and Manipulation

5.1.1 Summary Figures: jlrelegplots

Jlrelegplots is a good function to use to quickly get the gist of data from an experimental run. It does not require any processing; just load the data first.

```
jlrelegplots(jlist, [startfigno], [timeshift], [range]);
```

Two figures are plotted, one with raw and "sensor" based sensor data, the other with state and activation data for each joint. An example for a cockroach model experimental run is given below.

Parameters:

jlist	The joint list where all the data comes from. Need not have coordinate or unified data.
startfigno	If given, the two figures will start here and go in order, overwriting whatever might have been there before. If left out, new figures will be created in the first available slots.
timeshift	Optional boolean indicating if time should be shifted to start at zero. Defaults to 0 (no timeshift)
range	Optional two-element range argument passed to subset_data for each item to be plotted. Range is applied after any timeshift.

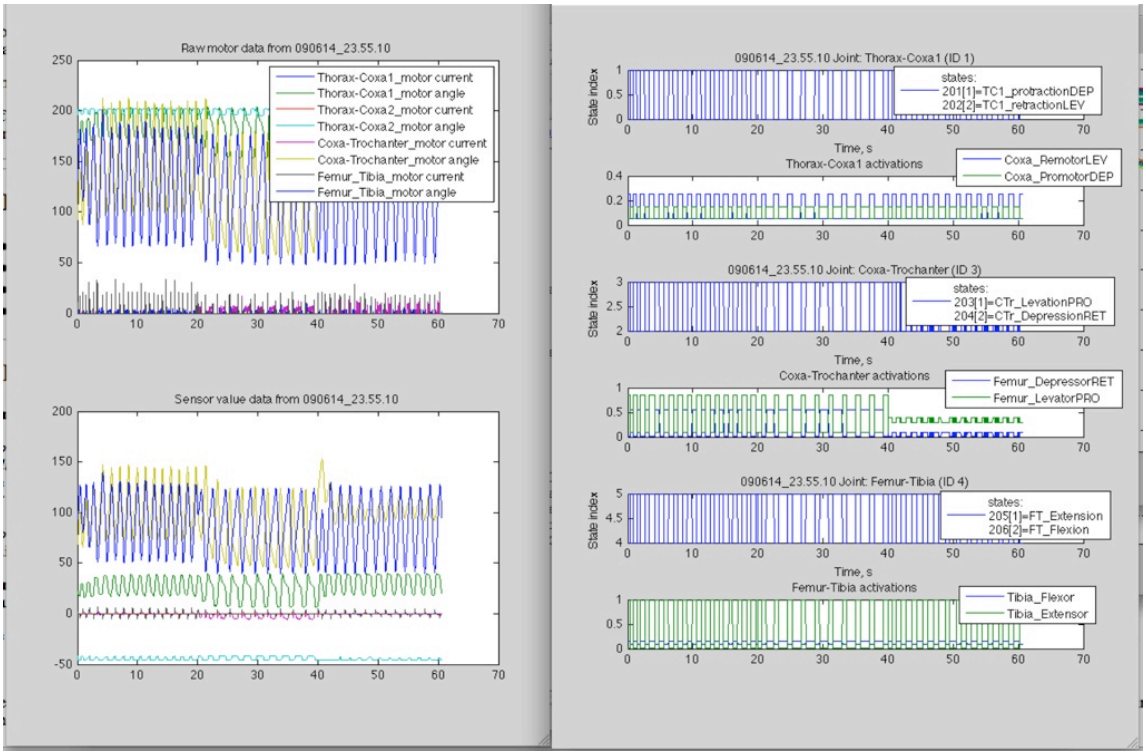


Figure 5-1 The output of jlrelegplots, in two figure windows. Note that the state labels are a little cryptic; in all current configurations the state labeled [1] will be displayed as the lower index on the graph. There is nothing in place to enforce this explicitly, however.

5.1.2 Object plot methods

Jointlists and most of the other objects have plot methods, with optional title prefixes; mostly used for automatically placing the experiment name in the figure titles. For a jointlist object jlobj, calling jlobj.plot() will set this experiment name title, and call the plot method on all its child joints and jointlists. The joint plot method plots that joint's statedata, then calls the plot methods on all of its child muscles and motors.

Calling jlobl.plot() generates a large number of figures; it is usually a good idea to use jlrelegplots first, and then generate the more detailed plots offered by joints, motors, and muscles when you know what you're looking for. The data plotted in the muscle plot function (example shown below), in particular, is not all shown by jlrelegplots.

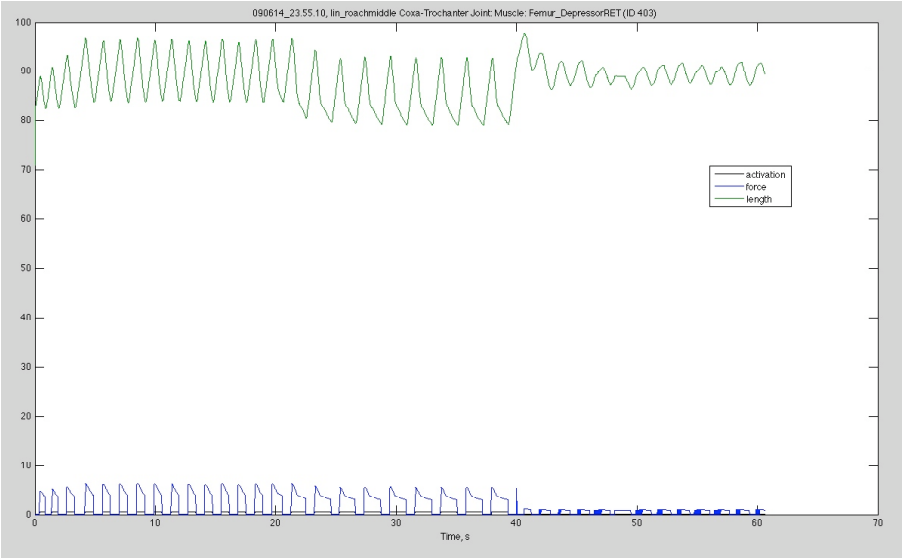


Figure 5-2 Object plot output for the femur depressor, including length and force output.

5.1.3 Object print methods

A text summary of configuration data and the size of contained data vectors is implemented in the disp and fullprint functions of objects. Jointlists also have a fullprint_recursive option, which calls the fullprint method of all contained joints. For our example data:

```
>> A.fullprint_recursive()
Jointlist: lin_roachmiddle
Fullname: cockroach middle leg, linear muscle models
Description: This describes a Blaberus discoidalis middle leg, forward stepping. Rearranged
  from the associated stickmiddle configuration file; currently with nothing
  going on at the ThC2 DOF.  Activations are currently all constant per-state.
  Linear muscle models at ThC1, CTr and FTi joints.

JOINTS

Joint: Thorax-Coxa1  idnum: [1]  parent [0]
Fullname:
Control mode: muscle
Description:
2 muscles, 2 states, 1 motors
```

Geometry:

ans =

```
  -0.9397         0   -0.3420         0
   0.2620    0.6428   -0.7198         0
   0.2198   -0.7660   -0.6040   10.5000
        0         0         0    1.0000
```

ans =

```
    1.0000         0   -1.5708    0.0175
```

isendpoint: 0
linespec: [ko]
colorspec:
Coordinates elements: 0 0

Coordinates_unified elements: 0 0

Thorax-Coxa1 MUSCLES
Muscle: Coxa_RemotorLEV idnum: 401
Fullname:
Data elements: activation [2493] activation_unified [0] force [2493] length [2493]
Muscle: Coxa_PromotorDEP idnum: 402
Fullname:
Data elements: activation [2493] activation_unified [0] force [2493] length [2493]

Thorax-Coxa1 STATES
State: TC1_protractionDEP idnum: 201
Fullname: []
linespec: []
colorspec: 0 0.5804 0.3882

State: TC1_retractionLEV idnum: 202
Fullname: []
linespec: []
colorspec: 0.6627 1.0000 0

Thorax-Coxa1 MOTORS
Motor: Thorax-Coxa1_motor idnum: 601
Angle elements; unified [0], fast [11952], raw [11952]
Torque elements; unified [0], fast [11952], raw [11952]

Joint: Thorax-Coxa2 idnum: [2] parent [1]
Fullname:
Control mode: position
Description:
0 muscles, 0 states, 1 motors
Geometry:

ans =

8.9000 6.5000 0 0.0175

isendpoint: 0
linespec: [bx]
colorspec:
Coordinates elements: 0 0

Coordinates_unified elements: 0 0

Thorax-Coxa2 MUSCLES

Thorax-Coxa2 STATES

Thorax-Coxa2 MOTORS
Motor: Thorax-Coxa2_motor idnum: 602
Angle elements; unified [0], fast [11952], raw [11952]
Torque elements; unified [0], fast [11952], raw [11952]

Joint: Coxa-Trochanter idnum: [3] parent [2]
Fullname:
Control mode: muscle
Description:
2 muscles, 2 states, 1 motors
Geometry:

ans =

9.7000 0 3.1416 0.0175

isendpoint: 0
linespec: [b]
colorspec:
Coordinates elements: 0 0

Coordinates_unified elements: 0 0

Coxa-Trochanter MUSCLES
Muscle: Femur_DepressorRET idnum: 403
Fullname:
Data elements: activation [2493] activation_unified [0] force [2493] length [2493]
Muscle: Femur_LevatorPRO idnum: 404
Fullname:
Data elements: activation [2493] activation_unified [0] force [2493] length [2493]

Coxa-Trochanter STATES
State: CTr_LevationPRO idnum: 203
Fullname: []
linespec: []
colorspec: 0 0.8627 0.8627

State: CTr_DepressionRET idnum: 204
Fullname: []
linespec: []
colorspec: 0 0.2980 0.6392

Coxa-Trochanter MOTORS
Motor: Coxa-Trochanter_motor idnum: 603
Angle elements; unified [0], fast [11952], raw [11952]
Torque elements; unified [0], fast [11952], raw [11952]

Joint: Femur-Tibia idnum: [4] parent [3]
Fullname:
Control mode: muscle
Description:
2 muscles, 2 states, 1 motors
Geometry:

ans =

   8.5000   0   0   0.0175

isendpoint: 0
linespec: [g]
colorspec:
Coordinates elements: 0   0

Coordinates_unified elements: 0   0

Femur-Tibia MUSCLES
Muscle: Tibia_Flexor idnum: 405
Fullname:
Data elements: activation [2493] activation_unified [0] force [2493] length [2493]
Muscle: Tibia_Extensor idnum: 406
Fullname:
Data elements: activation [2493] activation_unified [0] force [2493] length [2493]

Femur-Tibia STATES
State: FT_Extension idnum: 205
Fullname: []
linespec: []
colorspec: 1.0000   0.4000   0

State: FT_Flexion idnum: 206
Fullname: []
linespec: []
colorspec: 0.7098   0   0.2196

Femur-Tibia MOTORS
Motor: Femur_Tibia_motor idnum: 604
Angle elements; unified [0], fast [11952], raw [11952]
Torque elements; unified [0], fast [11952], raw [11952]

Joint: Tibia-Tarsus idnum: [5] parent [4]
Fullname:
Control mode: muscle

Description: The Tibia-Tarsus joint is in this model only the terminal endpoint.
No actual tarsus is currently modeled.
0 muscles, 0 states, 0 motors
Geometry:
isendpoint: 1
linespec: [r]
colorspec:
Coordinates elements: 0 0

Coordinates_unified elements: 0 0

Tibia-Tarsus MUSCLES

Tibia-Tarsus STATES

Tibia-Tarsus MOTORS

5.1.4 Defining data subsets & time

Since data from the engine can be voluminous, and the timebase can start at very large numbers, functions are built into jointlist and sub-objects to facilitate the definition of data subsets. jlist.tshift(tbase) subtracts tbase from all time data. jlist.tzero finds the smallest time present in any child object, and uses that as tbase. The plots shown have had the time zeroed, by running A = A.tzero()

Subsets of data can be created using the subset method, which uses a two-element time-based range to define a new object. e.g. A_transition1 = A.subset([18 22]) creates a new jointlist containing only the data from t=18 to t=22. This does not modify the time data; if you want the subset time to start at zero it will be necessary to call tzero on that new jointlist.

5.1.5 ListDataConfigs

This just prints a list of the subdirectories of the given directory name, along with the name of the jointlist configuration contained. Useful if you have a lot of experimental runs, and want to quickly see which configuration was used for each.

5.2 Kinematics: jlforwardkin

This computes the revolute forward kinematics for the kinematic linkage defined by a jointlist, using the Denavit-Hartenberg geometry information provided in the SCCILS configuration. The resulting 3D positions are put in the appropriate joint coords matrices

forwardkin calculates the pose of the kinematic chain described by jointlist, using the angle data therein, and storing the coordinates back in the output jointlist

```
jlist = forwardkin(jlist)
```

It is assumed that the unified data necessary to generate coords_unified exists, and also that there is sufficient geometry data to actually calculate something.

Currently, the code for unifying data is only in jlromodpathstates; it has not been put in a separate function. So you will have to run jlromodpathstates before running forwardkin– but jlromodpathstates runs forwardkin anyway.

5.3 Plots and Video with Kinematic Data

5.3.1 jlromodpathstates

Jlromodpathstates does the forward kinematics calculations for a jointlist object, also calculating unified times. 3D position output is placed in the appropriate "coords" element of each joint.

```
jlist = jlromodpathstates(jlist, [projection]);
```

projection is a 3-element vector with 1's specifying the coordinate plane of projection. This currently only in one of the coordinate planes; e.g. [1 1 0] is the x-y plane.

A path-state plot of the final joint in the jointlist (usually a special "endpoint" joint) is generated, as well as a 3D plot with traces for each joint colored as specified by the joint colorspec.

5.3.2 jlpathstateplot

```
handle = jlpathstateplot(jlist, projection, [offset],…
[jointname], [range])
```

This is a wrapper for the more general pathstateplot function, in which pathdata, statetada, colors, statenames and linestyles are all harvested from jlist.

jlist: the jointlist object containing desired paths, states, etc.

projection: [x y z] numerical component specifiers for the projection to be plotted. e.g. [1 1 0] is x-y, [1 0 1] x-z

offset: number of units neighboring lines should be set off from each other. Defaults to 0.1.

jointname: string naming the joint whose pathdata to use in generating the plot. Defaults to the last (i.e. <endpoint>) joint in jointlist

range: start and stop indices for plotting; defaults to plotting everything

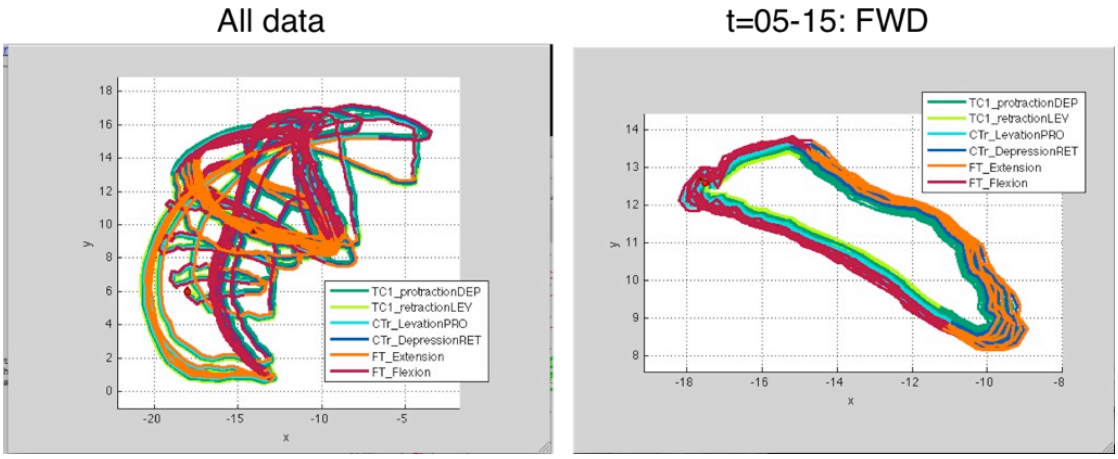


Fig. 5-3. Path-state plots of the example data; at right is the subset corresponding to forward stepping. The path-state plot is a plot showing simultaneous foot path and joint states. Each line represents the state of one joint, styled as shown in the legend. The projection of the 3D path of the designated joint is represented by the path that starts at the circle and terminates at the triangle; in this case the default "endpoint" joint corresponding to the end of the tibia was used.

5.3.3 jlmakelegavi

In many cases, it can be highly informative to see the current joint states along with the 3D pose of the leg. This is accomplished using jlmakelegavi, which plots a line for each kinematic segment, with the color of that line determined by the colorspec of the state active at that joint at that time. Since the robot or other model may have flexible elements, or the kinematic data may be in some other way incomplete, it is even more useful to synchronize these movies with video data taken during the experiment. In this way it is possible to see both the environmental interaction of the model and it's internal control state at the same time.

There is an interactive step in jlmakelegavi where the user manually sets the projection for the plot (or each subplot), and places the text displaying the time in the desired spot.

This is the jointlist-based version of makelegavi; p, colors, t and states are gleaned from jlist.

```
M = jlmakelegavi(jlist, filename, [fps], [timeshift], [range],… [subplots])
```

jlist: A joint list which has had jlforwardkin run on it; i.e. which contains useful coords_unified data.

filename string containing the name of the avi file to be saved.

fps frame rate for output (does not effect total number of frames). Defaults to 1/mean(diff(time_unified)) if time_unified can be found in jlist, 30 otherwise.

timeshift Optional boolean indicating if time should be shifted to start at zero. Defaults to 0 (no timeshift)

range Optional two-element range argument passed to subset_data after all the data has been extracted from jlist, and timeshifted if desired, before passing everything to makelegavi

subplots if present, a two-element vector containing [nrows ncols] for use in generating a multi-view movie

M An avi movie object. Limited usefulness.

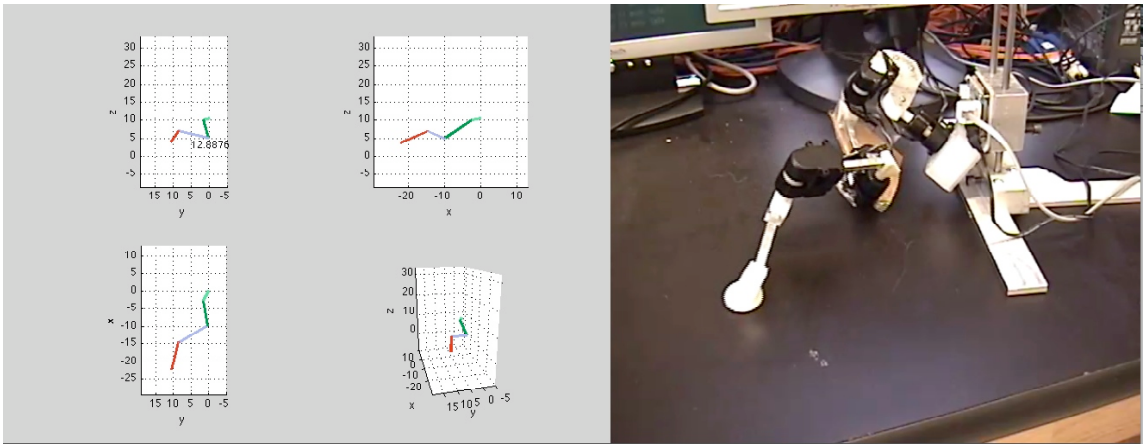


Figure 5-4 Experiment video data side-by-side with synchronized output of jlmakelegavi, with four separate projections.

5.4 Other Stuff

Not described in detail here, but:

stepanalysis is a program that runs the various statistical analyses on step variance and a whole lot more, used in analyzing the data from the muscle modeling experiments.

There are a number of other unfinished or older, pre-SCCILS files also in the SCCILStoolbox repository, which developers may find some use in perusing.

10.4 NeuRoMod Operation Manual

NeuRoMod Operation Manual

The CWRU SCASM Neuromechanical Robotic Model Control Software

Brandon Rutter

in collaboration with

John Bender (GUI)

Case Western Reserve University

January 2010

Table of Contents

Introduction

NeuRoMod, or the Neuromechanical Robot Model control software, is a suite of programs that grew out of the development of Sensory Coupled Action Switching Modules (SCASM) and the SCASM Command and Control Interface Language Specification (SCCILS). It was designed with the goals of allowing robot operation, model configuration, and experimentation. This manual is intended to provide the information necessary for the reader to operate a robot using this software.

A background on RTLinux program operation is provided as well. Through most of the manual a basic familiarity with Linux/Unix systems and basic commands will be assumed, though some introductory material is available in chapter 7.

Chapter 1 describes the basic tasks necessary to install the NeuRoMod system, presuming that you already have a computer running RTLinux. Chapter 2 describes model usage. Once you have installed the programs, you can use the "Quick Start" to get the robot moving, we hope. Chapter 3 describes the data output available.

Chapters 4, 5 and 6 are intended primarily for readers who wish to become more familiar with the underlying operation of the current robot control software, and learn how to write additional software to add to or modify present functionality.

As of the present (January 2010) this document is still a work in progress. If you feel there is something you need to know which is missing, or any other aspect of the manual which could be improved, please consider letting the collaborators know or, better, provide edits. Many of the programs here were based on the RUI software written for the control of pneumatically actuated robots; the manual for those programs is available in a separate document.

1 Installing NeuRoMod elements

NeuRoMod consists primarily of two collections of programs, the *user interface* and the *engine*, which are usually run on different computers that can reach each other over the network, but may also be run on the same computer. Each has a different set of requirements; but in both cases it is currently assumed that it is a UNIX-flavored machine (Linux or Mac), and you have access to a command-line terminal. Full installation will consist of installing both supporting programs and the NeuRoMod source code. If you are a new user on an already configured UI or engine machine, only the source code installation will be necessary.

1.1 User Interface

1) Supporting Programs

In order for the user interface to run, you need an appropriate Python installation. The following directions are for Mac 10.5, and some steps require a little ability for superuser text file creation. If you've got a Linux system you can probably figure out the corresponding commands with your system's package manager.

First install MacPorts: http://www.macports.org/ note: you will have to install Developer Tools if you haven't already, and you may need to update to the most recent version. Also note: if you want a fairly nice graphical interface to MacPorts, Porticus is a relatively useful and free option.

Then install the following ports:

py25-setuptools
python_select
py25-wxpython
py25-numpy
py25-scipy
py25-matplotlib
libpng
py25-pil

(e.g. "sudo port install py25-setuptools")

Actually not all of these are explicitly necessary, due to dependencies. But they all do need to be installed. Now set up the paths:

(as superuser) Add the following text file: /etc/paths.d/MacPorts and put the following two lines in:
/opt/local/bin
/opt/local/sbin

(as superuser) Add the following text file: /etc/manpaths.d/MacPorts and put the following line in:
/opt/local/share/man

These work for allowing access to MacPorts-installed stuff, but it generates paths that search the apple defaults first. We want MacPorts to take precedence, so put the following line in the .profile text file in the home directory for each user who wants to do this:

export PATH=/opt/local/bin:/opt/local/sbin:$PATH

Now, Since motmot is now on PyPI, just do
sudo /opt/local/bin/easy_install-2.5 motmot.wxvalidatedtext

And for the last bit, select the appropriate python:
sudo python_select python25

To test the installation so far, try the following few commands. This starts python, and verifies that you have installed wx. You should be able to get something very similar to the below, with no errors.

```
factotum:~ rutter$ python
Python 2.5.4 (r254:67916, Jun 21 2009, 19:37:44)
[GCC 4.0.1 (Apple Inc. build 5490)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import wx
>>> exit()
factotum:~ rutter$
```

2) Personal Installation
Once you have Python properly installed and configured, you will need to download the NeuRoMod user interface programs using Subversion.

On your user-interface machine:

Open a terminal window and navigate to the directory where you want to keep these files. Type the following commands to check out the programs and the documentation:

mkdir neuromod

svn co svn://consolation.cwru.edu/releg/romod/trunk neuromod

mkdir doc-neuromod

svn co svn://consolation.cwru.edu/releg/doc-romod/trunk doc-neuromod

At this point you should be able to get the UI up and running as in Figure 1-1 by doing the following:

cd neuromod/ui/pygui/romod-gui/
python romod_gui.py

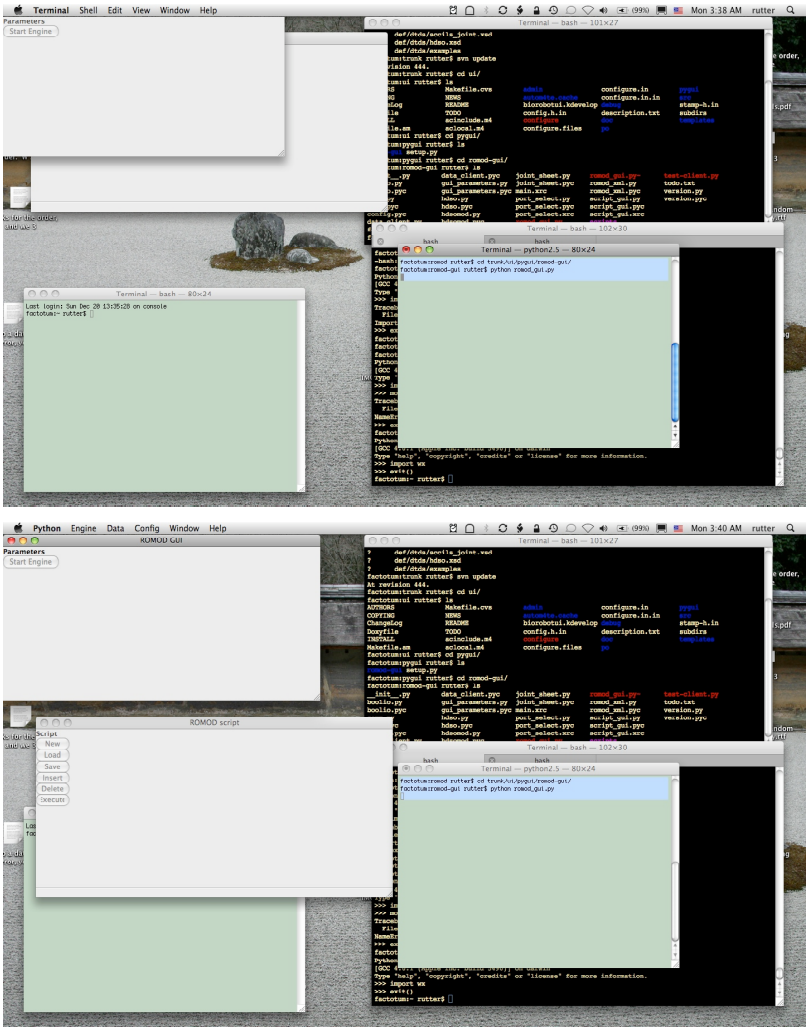


Figure 1-1 Top: Romod GUI windows as they may appear right after starting the program. Bottom: full views of the windows after some wrangling, which may include choosing window>zoom if you want to see the top bar of the main window.

1.2 Engine

1) Supporting Programs

For the engine to run you will need a computer which has RTLinux installed, as well as the proper version of Python (though the Python requirements are less than for the UI). You will also need to have the ability to run programs as the super user on the engine. An installation log, including details beyond a plain Debian Etch install for the RT-Linux kernel configuration and installation, and third-party RT-Linux modules necessary, is given in chapter 8.

2) Personal Installation

You will need to have a terminal open on your engine machine in order both to run the robot and to do the following install. On your engine machine, in the directory where you want to keep the engine programs, run the following commands:

```
mkdir neuromod
svn co svn://consolation.cwru.edu/releg/romod/trunk neuromod
cd neuromod/engine
make
```

If everything is working properly, you should now be able to run (in this same directory)

sudo ./enginecli help

and it will show you some help on running enginecli, the minimal command-line interface for running the robot.

NOTE: If you are in the habit of updating your working copy of the Subversion repository it will sometimes be necessary for you to rebuild the engine after an update. To be safe, do this as a matter of course whenever you update.

1.3 XML DTDs

This step should not be necessary if the computer you are using has already had supporting programs installed and is working.

In order to reduce errors, NeuRoMod programs use validating XML parsing, which means that they need to have access to the DTD files that define SCCILS. Since the robot controllers are used for demonstrations without network access with some regularity, this means that a local copy of the DTDs is now the default location references in the XML scripts and configuration files. The current XML DTDs should be put in /usr/local/src/romod_dtds/ . The most straightforward way to do this and keep it up to date is to do an SVN checkout there:

cd /usr/local/src

svn co svn://consolation.cwru.edu/releg/romod/trunk/def/dtds romod_dtds

Due to some previous attempts to keep an http-accessible copy of the SCCILS DTDs as the canonical version, some of the configuration and script files will still define their doctype using that URL; for instance, a script might have:

```
<!DOCTYPE hdsoscript PUBLIC "hdsoscript" "http://roach.biol.cwru.edu/romod-dtds/hdsoscript.dtd">
```

In particular, the copies of scripts from data directories may have this definition. If the file is not available there, this will create a validation error- this is usually reported in the case of configuration files, but may cause a more cryptic failure in some script handling. To reference the DTDs that were checked out to your local machine above, change this to the following. Note that the old URL has been left in place as a comment, in case somebody wants to switch back to that system.

Note also that some older configurations just have the filename only in the doctype, assuming that there is a copy or symlink of the DTD in whatever the current directory is. This was deprecated some time ago. If you intend to use any of these configurations, it is strongly recommended (and may be necessary) to update the doctype definitions to point to the DTDs in /usr/local/src/romod_dtds.

```
<!DOCTYPE hdsoscript SYSTEM "/usr/local/src/romod_dtds/hdsoscript.dtd">
<!-- <!DOCTYPE hdsoscript PUBLIC "hdsoscript" "http://roach.biol.cwru.edu/romod-dtds/hdsoscript.dtd"> -->
```

2 Using NeuRoMod

2.1 Introduction

NeuRoMod is a suite of programs for Neuromechanical Robot Modeling. It has grown out of efforts to control prototype model robots using Sensory Coupled Action Switching Modules (SCASM), and further efforts to configure and command these models using the SCASM Command and Configuration Interface Language Specification (SCCILS). The combined goal of all this activity is to allow operation, configuration and experimentation with robotic models of insect neuromechanical systems.

The NeuRoMod programs currently operate in two basic configurations: Graphical User Interface + Server, or command-line only. In this chapter, everything presumes that you have a working, compiled installation of the required programs. If you do not, you must first follow the installation instructions as given in chapter 1. If you are working on a computer that already has all the required software but you do not yet have a working copy of the repository, you may skip to the second half of the installation instructions for each machine.

"Quick Start" should get you from login to a running robot, "UI operation" covers some details of configuration and experimentation using the UI, and "Command-line operation" provides details on using the engineprog and enginecli command-line tools.

Unlike many robotic systems, these are for the most part neither dangerous nor fragile, so this manual is happily devoid of the usual warnings regarding imminent death. It does, nevertheless, help to have a modicum of common sense and caution engaged. Have fun!

2.2 Quick Start

The Quick Start works as long as everything is already properly installed, and there are no "stale" control modules in the kernel. It also assumes that you have checked out the repositories exactly as described in the installation instructions. If you checked out the repositories while in your home directory, you should be able to copy these commands verbatim.

I. GUI quick start: from a separate UNIX-type computer

a. In one terminal window, log in to the robot control computer remotely. e.g.

```
ssh username@computer.case.edu
```

b. Navigate to the translator directory of the repository, then start server.py:

```
cd neuromod/engine/translator/
sudo ./server.py -h computer.case.edu
```

c. In another terminal window, execute the Python UI program

```
cd neuromod/ui/pygui/romod-gui/
python romod_gui.py
```

d. In the "ROMOD_GUI" window that appears, choose Engine -> Connect to Engine… , then enter the robot control computer's host name, e.g. "computer.case.edu", as the hostname and click OK. Leave the port number at its default value.

e. Choose an engine configuration. Note that it must match the robot! For a simple working configuration use the following:

Cockroach model: Engine -> Type -> lin_roach

Stick insect model: Engine -> Type -> lin_stickmiddle

f. Make sure the robot is plugged in! The computer should have power, probably a network connection, and definitely a serial cable going to the robot converter board. The robot converter board should have it's own power connection, serial from the computer, and a four-wire bus connection(s) to the robot(s) being used.

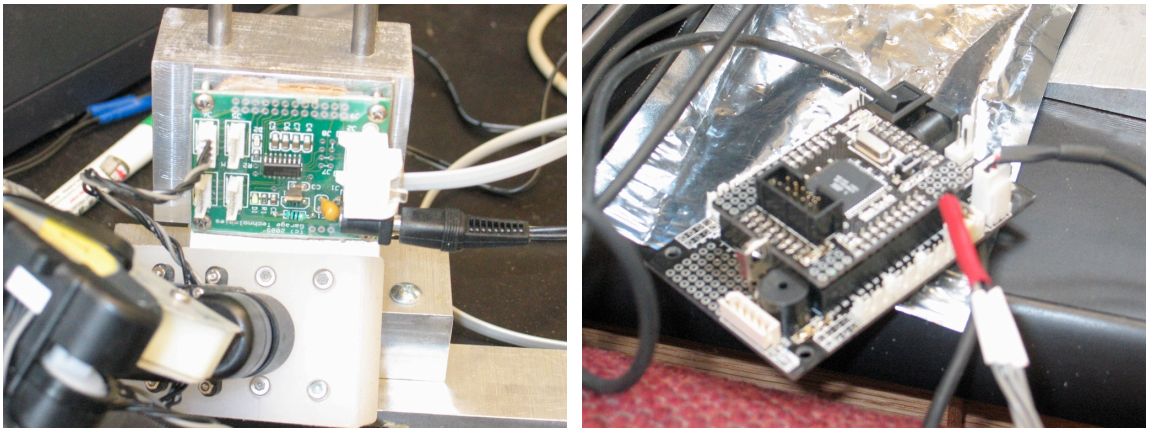


Figure 2-1 Fully connected robot converter boards. Left: the power board/level converter used on the cockroach robot. Right: the AT-Mega board that we just use for power and level conversion on the stick insect robot.

g. Click the "Start Engine" button in the UI main window. Ta-daaaa! If that didn't work, try the detailed instructions in the next section below, or try the command-line Quick Start.

h. To stop the robot motion, click "Stop Engine"

i. When you're done starting and stopping, choose the Engine -> Quit menu item. This will quit both the UI and the translator/engine.

II. Command-line Quick Start, from the control computer
   a. Log in directly to the robot control computer
   b. At the console or in a terminal, navigate to the engine directory of the repository:
      ```
      cd neuromod/engine/
      ```
   c. Choose a configuration, by copying the XML configuration files into this directory. For the cockroach, do (the last period *is* important):
      ```
      cp configurations/lin_roach/*.xml .
      ```
   d. Run enginecli in "ignorant" mode; it will just run for 30 seconds and stop:
      ```
      sudo ./enginecli i poo
      ```
      Note that if you choose a configuration for the wrong robot, nothing will happen.

2.3 Basic Rules

Though it is in theory possible to run the robots using either the GUI or command-line methods using only the Real-Time Linux control computer connected directly to the robot, this is not the way we usually use these systems, and for the most part we use a remote login for all of our work. There are a couple of reasons for doing this. You can use the X Window system, and have multiple terminals open, and run MATLAB, without having the RTL computer (which may be a poor, old, decrepit thing) doing all of the graphics work. You will actually have enough room on your screen to work with UI windows. This way you can also see the direct output from the RTL processes on the robot control computer as it comes up, rather than having to look back at system logs – useful for debugging.

Before running most of the programs which must execute on the RTL machine, including server.py and enginecli, you must become the super-user, otherwise there are a number of system commands which will fail. It won't work, and it may not tell you why. Mostly superuser status is necessary for loading and unloading kernel modules- you may find that you need to do this manually, when for one reason or another a module does not unload when it is supposed to.

You may note that most of the repository paths here refer to the trunk rather than a stable tag; we haven't been tagging much. Mostly the head of the repository is functional, but in some cases you will be better off checking out an older version;

possibly unstable commits are often noted in the logs, so you can use the "svn log" command to find a working version.

2.4 GUI operation

2.4.1 Basic GUI concepts

The Python GUI communicates with the low-level robot control programs via SCCILS. It is the "interface" referred to in Figure 2-2. The SCCILS communication layer takes place over TCP, whereas the Translator-Engine communication in the NeuRoMod engine is via RT-FIFO. So the Engine and Translator must be executed on the robot control computer, whereas the Interface may be in theory executed anywhere on the Internet (though very long delays do still present an issue).

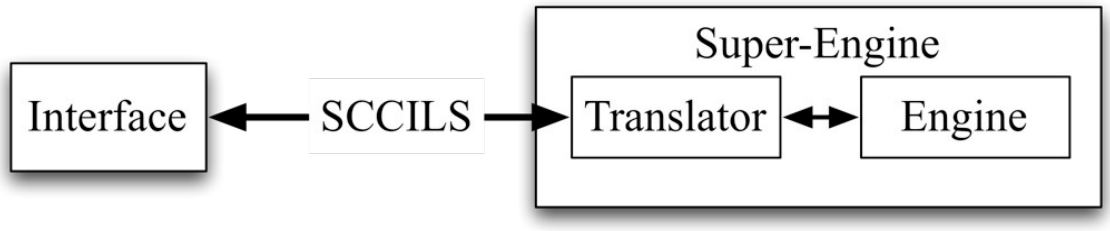


Figure 2-2 A system that has SCCILS. In NeuRoMod, romod_gui.py is the "interface", server.py is the "translator", and engineprog and associated programs are the "engine". Since it starts the engine programs, server.py may be considered the single super-engine program.

Since they are only connected by command, you can start either the Super-Engine (server.py) or the Interface (romod_gui.py) first; they just both need to be running when you try to "Connect to Engine" from the GUI.

2.4.2 Starting the Super-Engine, server.py

Server.py expects that the engine and RTLinux kernel modules are not loaded when it executes; under some conditions there may be engine modules left running, which can cause the super-engine considerable confusion and failure. It is therefore advisable to run `sudo ./cleanup.sh` (also in the translator directory) before running server.py, unless you know that the kernel is clean. You can determine whether any RTLinux modules are left in the kernel by running `lsmod`; if mbuff, enginemod or scriptmod are listed there is "stale" information in the kernel. Unless, that is, these programs are

actually being run actively from a different terminal. In any case, if these things are present it is likely that server.py will get confused.

There are a few other things which server.py expects, essentially boiling down to being in a full and built engine directory– these things are described at the end of the installation section. **NOTE: If you are in the habit of updating your working copy of the Subversion repository it will sometimes be necessary for you to rebuild the engine after an update. To be safe, do this as a matter of course whenever you update.** Files that are expected include the set of configuration files in ../configurations and the directory for saving data in ../data, where ".." refers to the enclosing "engine" directory.

Now, on the robot control computer, start server.py

```
cd neuromod/engine/translator/
sudo ./server.py -h computer.case.edu
```

"sudo" is necessary because server.py must be run as the super-user, since it does a lot of kernel-level module loading and unloading. The -p and -h flags are used to set the port and hostname at which the program listens for connections from an interface; defaults may be seen by running `./server.py --help:`

```
./server.py [-p PORTNUMBER=44106] [-h HOSTNAME=localhost]
Program must be executed from its own directory.
Program must be executed as superuser.
```

"localhost" is the appropriate hostname if you are running both the engine and GUI on the same machine; otherwise use -h to specify the DNS name of the computer where you are running the engine, e.g. minishadowfax.case.edu or digistation.case.edu. Once you have run the server.py command line, you should see something like the following line in that terminal, hereinafter referred to as the "engine terminal":

```
__waiting for connections on minishadowfax.case.edu:44106
Tue May 26 08:32:36 2009
```

2.4.3 Starting the Interface, romod_gui.py, and connecting

As mentioned above, the interface may be run on the same or a different machine from the engine, as long as both computers can communicate with one another via TCP and the appropriate port. The GUI is much closer to a single program than the engine, and as long as it is installed properly it is unlikely to require the kind of tinkering and fixing which the engine sometimes will. In a terminal on whichever machine you choose to run the GUI, execute the Python UI program

```
cd neuromod/ui/pygui/romod-gui/
python romod_gui.py
```

Note that it is usually possible to run the GUI on the robot control computer while logged in remotely using ssh, as long as you have X11 forwarding working. This will bog down the machine, however, and the GUI will be *much* less responsive than if you are running it locally.

In the UI window that appears, choose Engine -> Connect to Engine… , then enter the robot control computer's host name as specified when you ran server.py, e.g. "computer.case.edu" or just "computer" if your DNS is set up nicely, and click OK. If you are running the GUI on the same machine as the engine, the hostname should be "localhost". Leave the port number at its default value unless you set a different port number explicitly when running server.py. If everything goes well there will be no message from the GUI or the GUI terminal, but the engine terminal will probably show something like:

__server handshaking with ('129.22.143.92', 55018) Tue May 26 08:34:44 2009
__server sending simulation list to client
__server connected OK, listening

Now choose a configuration using the Engine menu in the GUI. Note that it must match the robot! The list of configurations here is generated by searching the engine/configurations directory on the engine computer; there is no guarantee that these configurations will actually work or even load the engine properly. For a simple working configuration use the following:

Cockroach model: Engine -> Type -> lin_roach

Stick insect model: Engine -> Type -> lin_stickmiddle

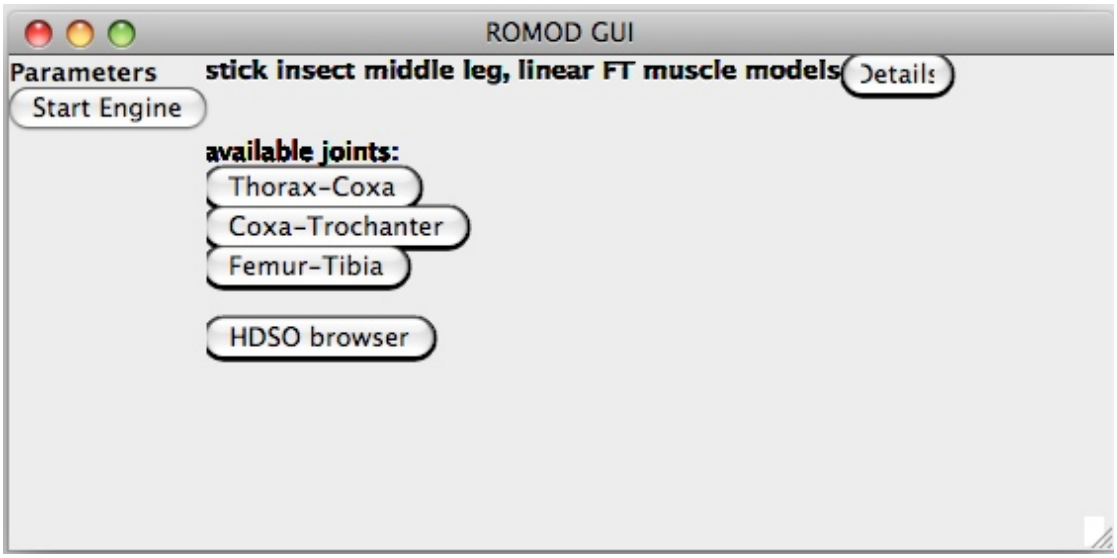


Figure 2-3 Main GUI window, with a successfully loaded configuration

If all goes well, the main GUI window will now look something like Figure 2-3, with a short description of the configuration you have loaded, a list of available joints, and a couple of other useful buttons. At this step the engine terminal and the RT console will produce a bunch of output related to loading the engine programs. If it is successful, the end of the output on the engine terminal should look like this:

```
  Making system calls to start enginemod module
  Warning: loading enginemod.o will taint the kernel: no license
    See http://www.tux.org/lkml/#export-tainted for information
about tainted modules
  Module enginemod loaded, with warnings
  Module                  Size  Used by    Tainted: P
  enginemod              21540   0  (unused)
  rtl_debug              22464   0  (unused)
  rtl_sched              26816   0  [enginemod rtl_debug]
  rtl_fifo                9344   0  [enginemod rtl_debug]
  rtl_posixio             9236   0  [rtl_fifo]
  rtl_time                4336   0  [enginemod rtl_sched
rtl_posixio]
  rt_math                20448   0  (unused)
  rt_com                 19752   1  [enginemod]
```

```
  rtl                    17424   0  [enginemod rtl_debug
rtl_sched rtl_fifo rtl_posixio rtl_time rt_math rt_com]
  mbuff                   5228  70  [enginemod]


  Engine initialized, I think.
  Warning: loading ../scriptmod.o will taint the kernel: no
license
    See http://www.tux.org/lkml/#export-tainted for information
about tainted modules
  Module scriptmod loaded, with warnings
```

If you see something like:

```
Making system calls to start enginemod module
insmod: a module named enginemod already exists
```

For either enginemod or scriptmod this means you had previously running or stale modules loaded. You should probably try to quit now and run cleanup.sh before trying this all again. In a few cases if things go horribly wrong, there will be "zombie" mbuff shared memory references, which may leave inappropriate data in a persistent shared memory store. If mbuff still shows up when you run lsmod after having run cleanup.sh, this has happened. The only way to fix this is to reboot the robot control computer: close all the programs you have open on the computer, make sure nobody else is logged in (by running "who"), and do `'sudo reboot'`.

If things have *not* gone horribly wrong, and you actually want the robot to move, now would be a good time to make sure that it is actually plugged in as in Figure 2-1: serial line connected to the controller's first serial port, and power plugged in and turned on.

2.4.4 Using the GUI: Interactively

The first thing you'll want to do is start the robot running; this is true because there are a set of configuration and data commands that don't work unless the engine is running. (Perhaps it would be a good idea to change this? Not sure). Do this by clicking "Start Engine". If the robot does not actually move, make sure that you have chosen an

appropriate configuration (you might be helped by clicking the “details” button in the main window) and that the robot is fully plugged in.

After you have clicked “Start Engine”, that button should change to “Stop Engine”, and the functionality will change appropriately.

During interaction, you may still have communication errors, as shown in Figure 2-4. After an error like this it is a good idea to quit both the GUI and the engine before trying again. In some cases the communication will have become so broken that the engine just continues running; in this case you must type “control-c” in the engine terminal. Eventually it should realize that you have typed this interrupt character, and unload.

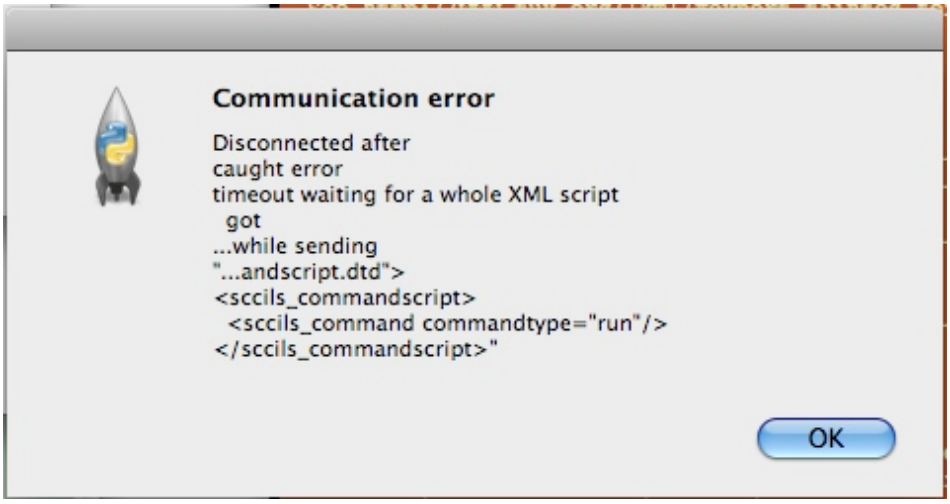


Figure 2-4 Our Friend the SCCILS communication error.

2.4.4.1 Parameter Modification

Once you’ve got the robot running, you can use the GUI to modify any of the SCCILS-aware controller parameters. These are available either by joint or through the all-parameters HDSO browser. The joint browsers reflect the most logically presented parameter and sensor values, but there are some global parameters which can only be viewed and modified in the HDSO browser. There are a couple of things here which have a pretty drastic effect on overall model function; for instance changing motor_speed should have instant and drastic effect (be careful about setting it to 0!). If you want to really mess things up, try changing some of the one and zero constants at the bottom of the list, or the control delays.

In most cases you will find it most helpful to modify parameters through the joint browsers. The easiest example of a benign but critical modification is in the stick insect robot. Get it running and loaded, then choose the Femur-Tibia joint. Click the states, FTi_Extension and Tibia_Extensor activation disclosure boxes, so that you get a window that looks like Figure 2-5. Since the Femur-Tibia angle is critical to the swing-stance transition, you can get the leg stuck in mid-air by turning down HDSO 130, FT_extension_extensor_activation– the activation of the tibia extensor during FTi extension. Turn the activation back up to get it stepping again.

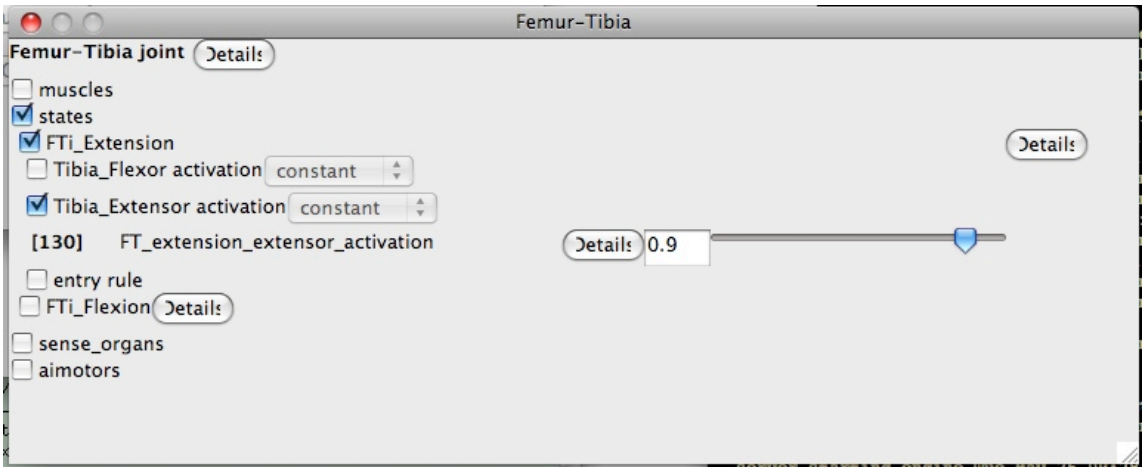


Figure 2-5 Having fun with the stick insect Femur-Tibia joint– slide it down to stop the stepping

There are quite a few more things going on at the joint. Currently, the muscles only display what configuration has been loaded, but do not allow online modification. The same is true of aimotors. Sense_organs do show which robot sensors are associated with a biological sense organ, and can display the current value of those sensors (see section on data); however this is not as useful as it could be at the moment. States is really where the action is.

Each state is an "action" in the action switching module (ASM) associated with this joint. A state has activations for muscles at the joint, and a rule for entering the state. The rules for state entry define the functional control network topology for coordination (sometimes referred to as "timing"), and can be translated to and from event-space diagrams. The activations determine what the dynamic output of this action is, and essentially define the magnitude control/influences in the modeled control system.

In Figure 2-6 we can see the full information for the FTi_Extension state. First there are muscle activations for both of the muscles at this joint; since this is the extension state it makes sense that the extensor is more activated than the flexor. Note the "constant" after each activation- the configuration files allow for more complex activation types. It is possible to modify the parameters for any activation type, but it is not currently possible to modify the kind of activation function online.

The entry rule is a Boolean expression that indicates when this state/action should be activated. Unfortunately, for now it is expressed via HDSO number rather than name, so you need to look at the entries below to figure out what it actually means. In the example shown, the rule is to transfer to the FTi_Extension action if (Coxa-Trochanter Current < FT_ext_CTr_load) OR (Femur-Tibia Angle > FT_ext_FTang). Coxa-Trochanter Current and Femur-Tibia Angle are sensor HDSOs, which change based on sensory input. FT_ext_CTr_load and FT_ext_FTang are parameters which determine the sensory space in which this transition takes place: this sensory space is represented in Figure 2-7.

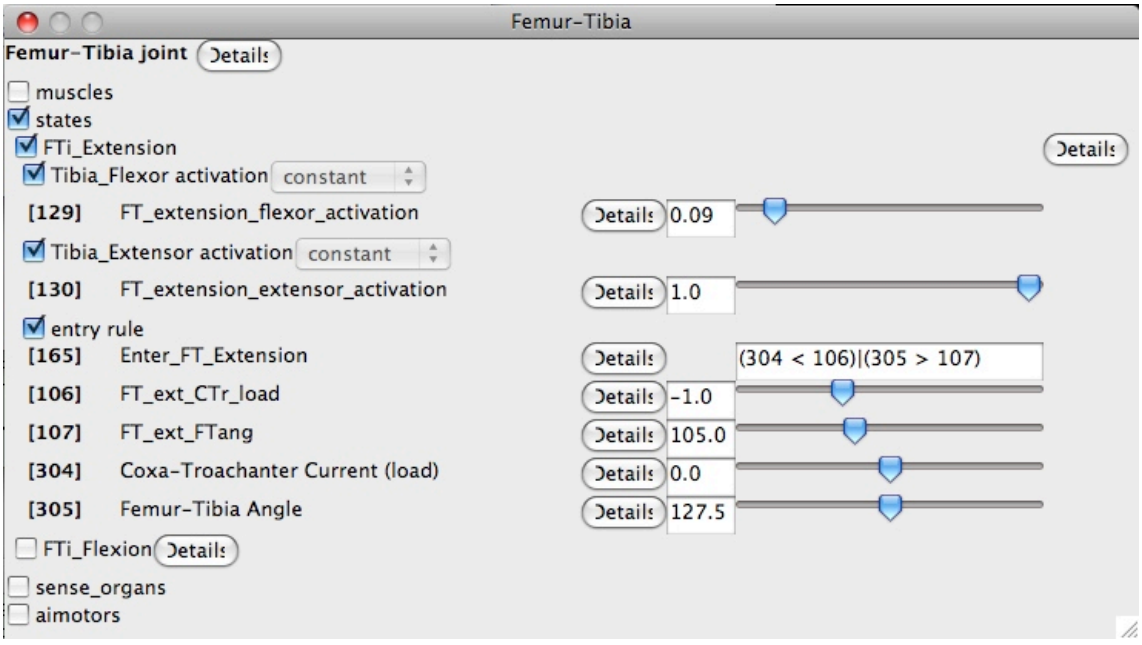


Figure 2-6 More fun with the FTi Joint– transition rule (action switching event) nitty gritty.

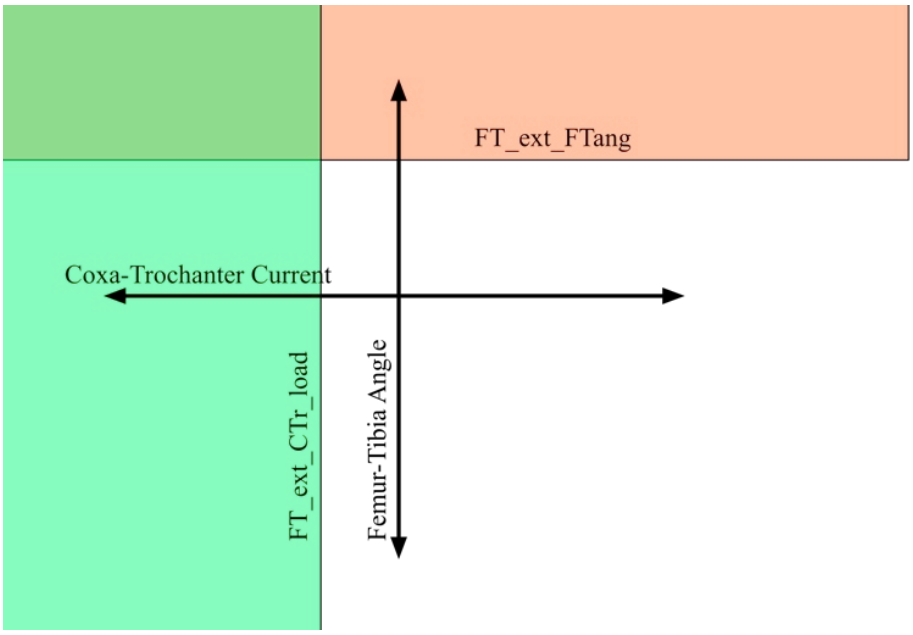


Figure 2-7 Sensor-space representation of Boolean transition rule in Figure 2-6. All shaded areas result in a transition to FTi_Extension.

2.4.4.2 Seeing and Gathering Data

While running the UI, there are two basic ways of routing data: 1) through SCCILS to the UI and 2) locally at the engine. The data available to the UI are all HDSO data elements, including parameters and sensors. To enable data streaming from the engine to the GUI, choose the Data -> Stream Data menu item in the GUI. (without any parameter windows open). You may now chose Data -> Synchronize Controls in order to load all HDSO data values from the engine into the UI. This is useful both for verifying parameter settings, and for getting current sensor data loaded into the UI. In streaming mode, there is now also a checkbox to the right of every HDSO data field, which indicates whether to "stream" this HDSO, updating it every second and displaying in a graph (though the graphing doesen't work too well just yet).

To log a large amount of data locally at the engine, choose Data -> Log Data; choose this item again to stop local logging and save the run in a timestamped data directory in engine/data. For completeness of the data recording, it is best to combine this with the Data -> Stream Data function, since that will activate HDSO logging locally, which should provide you with a guaranteed known configuration at any time in the data record.

For advanced local logging even when using the UI, use the datalogging command line program directly, as described below in section 2.5.3.

If you are running the robot using the command-line "enginecli" program, all data is always logged locally; where it is logged depends on the data directory argument to enginecli.

Caveats:

Currently, it looks like you have to explicitly stop local logging before quitting the engine.

Turning data streaming on causes scriptmod to get stuck in the kernel until server.py is unloaded; you will need to run cleanup.sh before the next run.

2.4.4.3 Saving a Configuration

Currently the easiest way to save a configuration or multiple configurations is thus:

1) turn on local logging
2) stream data
3) Whenever you want to save your current configuration, choose "Data -> Synchronize Controls". These HDSOs are then all logged locally (see the data formats section) and can be extracted for other use with minimal effort.

Alternatively, it is intended that Config -> Save should work in the UI. This appears to nearly work, however there is currently some disagreement between the engine and UI in the initialization of HDSO values, and some values that haven't been modified interactively (like, say, the constants) are likely to have incorrect values saved. The intention is to fix this by changing the HDSO init_value element to be an HDSOmod; this would involve some redundancy of information because the mod would have to list its containing HDSO ID, but would allow easier UI initialization just by scanning the config files for all HDSOmod elements.

Additionally, the current SCCILS spec still passes long-long-int (llint) times for some values, but these are not handled properly in the GUI. The current plan is to change the xmltype of these values to float, and have the translator handle the translation to the underlying llint engine data type.

2.4.5 Using the GUI: Scripting

Script generation, loading, and saving using the GUI are all problematic at this point- there are some issues with GUI rendering (at least on Mac python) as well as the problems mentioned above with llint data handling. I do not currently use the GUI for scripting.

2.5 Command-Line operation

While the GUI is crucial for interactive exploration of the system, and makes for some excellent demonstrations, there are a number of things that are handled a bit better from the command line at this point. All of these tools must be run with super-user privileges in order to succeed; they won't even show you the help as a standard user.

There are, of course, plenty of things that can go wrong here. If you are having trouble and the reasons why are not obvious, it may help to take a look at explanation of program output in section 2.6 . If you are running any of these commands on the console of the engine computer, output from the real-time processes will also be printed to the screen and may be voluminous. These messages are not processed through the UNIX shell, so the command prompt may not automatically appear after they've been printed and the program has finished, even though you can still type a command and hit return to see the output and the next command prompt.

2.5.1 Enginecli

Enginecli is the main command-line tool for running the robot; it will handle calls to the lower-level engineprog and datalogging commands for a particular experimental run. This is the program used for generating all the experimental data presented in Brandon Rutter's dissertation. The help output of enginecli is as follows:

```
Usage: ./enginecli mode_spec outputdir [scriptfile] || help
mode_spec is I for "interactive" or S for Script.
outputdir of "auto" creates a time-derived output directory name
outputdir of "poo" overwrites the old "poo" without confirming
```

“interactive” mode is also occasionally referred to as “ignorant” mode– there is in fact no interactivity; it just loads the locally available configuration files and tries to run the model using that configuration for 30 seconds. Script mode does indeed use an XML script file; if the file name argument [scriptfile] is given, that file will be attempted. Otherwise the user will be prompted to enter a script file name. The script path is expressed with respect to the current working directory.

Outputdir specifies the output data directory name; the contents of this directory are described in section 3.1 . The actual directory path used will be prepended by “data”; in normal usage where enginecli is executed in the neuromod/engine directory this will result in the path to the output directory being neuromod/engine/data/”outputdir”. Specifying an outputdir of “auto” will automatically generate a year-month-day-hour-minute-second based data directory there in the form yymmdd_hh.mm.ss . Specifying “poo” as the output directory does indeed put the data for the run in data/poo, but it will overwrite the previous data there without asking. This is useful when you don’t want to keep the data for long (e.g., debugging), or you don’t want to look at it at all (e.g., demonstrations).

In either mode, enginecli requires the three XML configuration files for the desired configuration to be in the current working directory, neuromod/engine. These are the jointlist file joints.xml and the HDSOlist files parameters.xml and sensors.xml. There is a configuration clean-up “make” script to clean up whatever configuration files might have been in this directory beforehand. So, to make your current configuration lin_roach, one would do:

make cleanconfig

cp configurations/lin_roach/*.xml .

In script mode, the script file must be a valid “hdsoscript” XML file. Since both initial configuration and the script file are saved to the data directory, it will be possible to re-run an experiment nearly exactly by doing the following steps. It should be noted, however, that modification of the source code might in some cases cause different results; the hard-coded configuration-like elements that have an effect on control are now very few, but for a guaranteed reproduction of the run it will be necessary to make sure you

are also using the same version of enginecli and supporting programs and libraries. This can most easily be done by making note of the SVN repository version used for an experiment, then reverting to that SVN version and re-making the engine when you want to exactly replicate the experiment. Note that of course this will not exactly replicate the physical conditions of the original experiment, and given that the force control is uncalibrated and open-loop, you may get substantially different results anyway if the robot is significantly worn, has a higher or lower body posture, is stepping on a different surface, or has a significantly different motor supply voltage (which you will likely see in Europe vs. U.S., with our little wall-wart unregulated voltage supplies which must run at 110, but have different output at 50Hz than at the U.S. 60 Hz).

That all having been said, there are many situations where re-running an experiment from the files in the experiment's data will work just fine without headaches involving different versions of the source code or differing physical conditions. If you are in the neuromod/engine directory and want to re-run an experiment called "publishable_data", the following steps should do it:

make cleanconfig

cp data/publishable_data/*s.xml .

sudo ./enginecli s publishable_data_rerun data/publishable_data/scriptfile.xml

Note that you can name the output directory whatever you like, though you probably don't want to name it "publishable_data" and overwrite that if it really was publishable. For any outputdir other than "poo", though, enginecli should ask you twice before overwriting a previous data directory. If outputdir is "auto" and the clocks are set reasonably well on your robot control machines, this should give you a unique data directory name.

The above only works if that data directory was generated using enginecli; if it was generated using datalogging while using the graphical UI, it may be necessary to use the HDSOlog and Scriptlog files to actually reproduce the configuration changes; there is currently no automated way to do this. The biggest problem here is that datalogging copies default configuration files from the current directory regardless of whether those

configuration files were actually loaded in the engine. It should be possible to detect if the config files are incorrect by comparing them with data in the HDSOlog, but this is a pain in the neck.

Note also that this presumes that the configuration and script XML files are still fully valid and contain references to existing DTDs. If you get “document not valid” errors, most likely these scripts either specify incompatible or nonexistent DTD files; see the notes on DTD installation in section 1.3 above.

2.5.2 Engineprog

Engineprog is the user-level program used for loading configuration files and initializing the RT portion of the engine, and unloading the RT engine when called with that argument. Since it is often called from other programs such as enginecli, it has a return value: zero for no errors, number of errors for non-fatal errors, or a negative number if a fatal error occurred. The output of “sudo ./engineprog help” is as follows:

```
./engineprog [V*|Q*]L [config files]
or ./engineprog [V*|Q*]U
V is for verbose, in which case config/debug info
will be printed to screen. Q(uiet) suppresses most output.
Both V and Q can be repeated to be more verbose or more quiet.
L is for loading the config files and initializing the
RT part. Currently, if one is specified all three must
be specified, in the following order:

joints sensors parameters

Otherwise engineprog will attempt to load the default
config file names.
U is for unloading the RT part and shutting down shared
memory.
```

As a robot operator, the most likely uses you will have for engineprog are configuration debugging and unloading the engine in the case of user interface failure. “engineprog l” is a good initial check of file validity and RT-Linux module loading; vl (verbose load) also prints basic configuration data for the joints, muscles, states and sensors, which can be useful in finding problems if a new configuration is acting strangely. vvl also prints out all the parameter and sensor HDSOs with their data; this is quite a lot of output but can be useful if you are looking for specific problems with HDSO data initialization.

Using the unload command “engineprog u” can be useful or necessary if enginecli or the SCCILS translator crash badly and leave enginemod running with shared memory in place.

Developers of new ui software or higher-level control may find engineprog useful; both enginecli and the Python translator use engineprog to load and unload the low-level robot controller.

2.5.3 Datalogging

Datalogging is a program used to start and stop monitoring and recording of low-level engine data.

```
Usage: ./datalogging mode_spec [outputdir] | help
mode_spec is L for load/start or U for unload/stop.
outputdir may only be specified when loading.
outputdir of "auto" (or blank) creates a time-derived output directory name
outputdir of "poo" overwrites the old "poo" without confirming
```

Largely this is a command-line interface to load and unload the “monitor” data recording and forwarding background process. This is done by enginecli itself (which does not call datalogging) and there is considerable code duplication between these programs. Datalogging is most useful when you want to explicitly control data directory names and logging start/stop times during a GUI interactive session; keep in mind however that datalogging records the configuration files in the current working directory and does not check to see if those represent the actually-loaded configuration. It can also

be useful to attempt unloading monitor when GUI data logging or enginecli has crashed and left it running.

2.6 Command-Line Output

Engineprog and enginecli can produce considerable output; an annotated example of a successful script run is included here to help determine whether what you're seeing is normal.

```
rutter@minishadowfax:~/releg/romod/trunk/engine$ sudo ./enginecli s
ExpA scripts/linR_fwd-itf-itb.xml
I think I have the following 4 arguments:
./enginecli S ExpA scripts/linR_fwd-itf-itb.xml
```

The following lines are the result of engincli calling "engineprog l"

```
I think I have the following 2 arguments:
./engineprog L
Warning: loading /lib/modules/2.4.29-rtl3.2-rc1_8-5-05/misc/rt_com.o
will taint the kernel: no license
  See http://www.tux.org/lkml/#export-tainted for information about
tainted modules
Module rt_com loaded, with warnings
Warning: loading /lib/modules/2.4.29-rtl3.2-rc1_8-5-05/misc/rt_math.o
will taint the kernel: no license
Module rt_math loaded, with warnings
Module rtl_time loaded, with warnings
Module rtl_posixio loaded, with warnings
Module rtl_fifo loaded, with warnings
Module rtl_sched loaded, with warnings
Module rtl_debug loaded, with warnings

Scheme: (-) not loaded, (+) loaded
  (+) mbuff
  (+) rt_com
  (+) rt_math
  (+) rtl
  (+) rtl_debug
  (+) rtl_fifo
  (+) rtl_posixio
  (+) rtl_sched
  (+) rtl_time
```

At this point the RT-Linux system has been loaded successfully; we don't care too much about all the warnings regarding tainted kernel. The following lines report successful validation and parsing of the XML configuration files.

```
xml_open_validate: Validity: 1
Try Validation: 1
xml_open_validate: Validity: 1
Try Validation: 1
ParseJointFile: Starting
xml_open_validate: Validity: 1
```

```
Try Validation: 1
Found 4 joints
engineprog: main mbuff_alloc succeeded
xml_open_validate: Validity: 1
Try Validation: 1
xml_open_validate: Validity: 1
Try Validation: 1
xml_open_validate: Validity: 1
Try Validation: 1
hdso_init_data: initnum = 9
ParseJointFile: Starting
xml_open_validate: Validity: 1
Try Validation: 1
ParseMuscles: Starting
ParseStates: Starting
ParseJoints: Starting
```

The following warnings are usually okay; there is an implicit default mapping of sensors to aimotors that required no configuration in earlier versions of the code. These lines indicate that the joints configuration file has not had the sensor mapping updated and included, and it is setting the old default mapping.

```
ParseJoints: Warning: unreasonable number of sensorspecs [0] found for
aimotor 601; setting defaults
ParseJoints: Warning: unreasonable number of sensorspecs [0] found for
aimotor 602; setting defaults
ParseJoints: Warning: unreasonable number of sensorspecs [0] found for
aimotor 603; setting defaults
ParseJoints: Warning: unreasonable number of sensorspecs [0] found for
aimotor 604; setting defaults


Making system calls to start enginemod module
Warning: loading enginemod.o will taint the kernel: no license
  See http://www.tux.org/lkml/#export-tainted for information about
tainted modules
Module enginemod loaded, with warnings
Module                  Size  Used by    Tainted: P
enginemod              21540   0  (unused)
rtl_debug              22464   0  (unused)
rtl_sched              26816   0  [enginemod rtl_debug]
rtl_fifo                9344   0  [enginemod rtl_debug]
rtl_posixio             9236   0  [rtl_fifo]
rtl_time                4336   0  [enginemod rtl_sched rtl_posixio]
rt_math                20448   0  (unused)
rt_com                 19752   1  [enginemod]
rtl                    17424   0  [enginemod rtl_debug rtl_sched
rtl_fifo rtl_posixio rtl_time rt_math rt_com]
mbuff                   5228  70  [enginemod]

Engine initialized, I think.
```

At this point engineprog is done, and enginecli continues with its job. Note how paranoid enginecli is about overwriting data.

```
enginecli: main mbuff_alloc succeeded
enginecli: I think outdirname is: 'ExpA'
```

```
I think the script file is: 'scripts/linR_fwd-itf-itb.xml'
Data directory data/ExpA already exists. Overwrite, erasing all
previous data in this directory? yes
yes
Are you really sure you want to annihilate data/ExpA and all its
contents?? yes
yes
Making directory because I think I removed it
enginecli: making system calls to start HDSO script engine
Warning: loading scriptmod.o will taint the kernel: no license
  See http://www.tux.org/lkml/#export-tainted for information about
tainted modules
Module scriptmod loaded, with warnings
enginecli: about to open command fifo (1)
enginecli: successfully opened command fifo
enginecli: opening puppet file FIFOs
Parsing puppet file scripts/linR_fwd-itf-itb.xml as xml...
enginecli: about to start monitor
monitor: Beginning monitor loop
enginecli: about to enable all joints
enginecli: about to start RT task
enginecli: started RT task
```

At this point enginecli's job is done for a while, since it has loaded the script onto the script FIFO and started all the RT tasks. The following lines are from the monitor data handling program; all the "wrote inhdso" lines mean that it recorded some HDSO script data, which happens when script entries are enacted by scriptmod and verification is sent on the scriptlog FIFO.

```
Engine Module info, time 8456.904452 sec
Everything Enabled
Engine Module info, time 8456.906836 sec
Starting
monitor: wrote inhdso
Script Module info, time 8456.938274 sec
Starting
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
```

```
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
monitor: wrote inhdso
Script Processor thread info, time 8516.909708 sec
Subtask Done
```

The above reports that the script handler stopped because the script FIFO had been emptied; this indicates that the scripted experiment has finished. This is conveyed to enginecli via shared memory, and enginecli then shuts things down.

```
monitor: wrote inhdso
enginecli: about to stop RT task
enginecli: sent stop commands
enginecli: Waiting for monitor to quit...
monitor: Exiting.
Done.

enginecli: Wrote data files to data/ExpA

I think I have the following 2 arguments:
./engineprog U
rmmod: rtl is in use

Scheme: (-) not loaded, (+) loaded
  (-) enginemod
  (-) mbuff
  (-) rt_com
  (-) rt_math
  (-) rtl
```

```
  (-) rtl_debug
  (-) rtl_fifo
  (-) rtl_posixio
  (-) rtl_sched
  (-) rtl_time


rutter@minishadowfax:~/releg/romod/trunk/engine$
```

All the minuses indicate that the RT-Linux system was successfully and completely unloaded. If there are still modules loaded, try “engineprog u” and “datalogging u” to see if perhaps something crashed but can still be sensibly unloaded. If this doesn’t work you probably have zombie mbuffs or RT-processes and need to reboot.

2.7 Configuration

Configuration and scripting in NeuRoMod are done using the SCASM Command and Configuration Interface Language Specification (SCCILS). The reader is hereby referred to the SCCILS 0.1 documentation, available in the doc-romod repository or as part of Brandon Rutter’s dissertation.

3 Data Output

3.1 The Data Directory

The data directory created by datalogging or enginecli contains pretty much everything you need to recreate a particular experimental run, in a lucky set of 12 or 13 files, in 5 convenient categories.

1. Configuration files. The SCCILS config files used for this run are copied here; joints.xml, parameters.xml and sensors.xml. Content of these files is described in the SCCILS documentation. Note, however, that if this directory was generated by datalogging then it is not guaranteed that these are the correct configuration files.
2. Script input file. If enginecli was run in script mode, the script file used is copied to scriptfile.xml.
3. Message logs.
    a. Infolog.txt : text interpretation of messages sent on the INFO2 real-time FIFO; messages regarding when various RT system events happened; including engine start and stop, and module loading.
    b. Debuglog.txt : text interpretation of messages sent on the DEBUG2 real-time FIFO; specifically for debug data. Not really used for anything at the moment.
    c. dmesglog.txt : output from the "dmesg" command when data logging is stopped; this is where direct output from real-time processes go; primarily useful for debugging.
4. Engine data logs. These four text files contain output from the various engine realtime threads, received by monitor via FDATA2 and CDATA2 FIFOS. Their names and formats are as follows. Elements are listed in the order they occur in the shared-memory arrays. The mapping of these to SCCILS configuration is currently implicit, and can be determined by examining the data loading functions in the SCCILStoolbox MATLAB programs.
    a. MotorData.txt (from motor_action thread in enginemod)

time(ns) motor0current motor0posn. motor1current motor1posn. ...

b. SensData.txt (from motor_conversion thread in enginemod)

time(ns) sensor0value sensor1value ...

c. StateData.txt (from state_transition thread in enginemod)

time(ns) joint0state joint0activation[0] joint0activation[1] joint1state ...

d. MuscleData.txt (from muscle_conversion thread in enginemod)

time(ns) muscle1force muscle1length muscle2force muscle2length ...

5. SCCILS data logs. These are HDSOscript files; raw FIFO data is logged during the run and translated to XML once data logging is stopped. Whereas time reported in the engine data logs above are absolute, the script file times reported are relative to the engine start time; thus you will need to use the logged engine start time in order to unify the time representation between the two sets of files.

   a. Scriptlog.xml contains a log of all the HDSOmods sent to SCRIPT_FIFO which were actually implemented by the script task, along with time of implementation. If the data is from an enginecli run in script mode, this should return something very close to scriptfile.xml as long as the run worked properly. Engine start time is also logged here by enginemod, though due to the FIFO nature of logging it may be reported late (the time values recorded will still be correct) if there are log entries already on SCRIPT_FIFO (as is the case when using enginecli).

   b. HDSOlog.xml contains a log of all the HDSO data requests sent on HDSO_REQUEST_FIFO which are actually handled by the HDSO server task; this contains an exact equivalent of the data sent during runtime to the HDSO_DATA_FIFO. When logging is started, the entire HDSO database is dumped here, to insure that the configuration can be

accurately known at any time. This is to deal with the possibility that HDSO data might be modified after engine start time and before logging is started. Note that both local logging (via monitor) and SCCILS logging must be enabled for HDSOlog to actually contain data.

3.2 Getting data onto your computer

Though in theory it is possible and desirable to channel a lot of data from the robot control "engine" computer to the "interface" computer through SCCILS, current reality is that there is a lot more and much higher time-resolution data available at the engine, and you are likely to want to get this to your own computer for analysis. For the reader with enough UNIX-fu, `scp` and `scp -r` will occasionally be the easiest and fastest way to get data from the engine/data directory. If you've done multiple experiments, though, and for other reasons as well, you may wish to use an sftp client, or rsync. This is what I usually do. The following example gets data from my engine/data directory on digistation, putting it in the data/ directory in the working directory (i.e., you would execute this in the neuromod/engine directory on your local machine (replacing "rutter" with your username, if you had followed the installation instructions in section 1 above while in your home directory on each machine).

rsync -avu -e ssh --exclude=*svn* rutter@digistation:~/neuromod/engine/data/ data

The rsync options used are:

-a: archive mode; preserve all sorts of permissions and other file metadata

-v: verbose; tell me what's happening

-u: update; do not overwrite newer items in my local data directory (in case I've been messing with them)

-e ssh: use ssh as the remote shell

--exclude=*svn*: exclude all of the hidden files SVN uses to keep track of revisions, etc.

The -n option is also often useful; this just prints out what the command *would* do, instead of actually doing it. In other words, you can run “rsync -n put_anything_here” without changing or messing up anything.

4 RTLinux Basics

4.1 Introduction

RTLinux is one of a couple of different methods for giving a computer running Linux real-time capabilities. In the context of operating systems, “hard real-time” does not mean fast, it just means that you have the capability of ensuring that a task will be completed by a specified time. Thus, RTLinux will not make a computer do anything faster but it will give it the ability to do things on a fairly accurate schedule, as long as those things are within the processing capability of the computer.

This chapter will provide a brief overview of how RTLinux adds real-time capabilities to the Linux operating system, and how a program can be given a real-time execution schedule. Much of the information here is available in more detail in *Getting Started with RTLinux*, which is included in HTML form in the documentation of the RTLinux distribution. If you follow the same directions as done for digistation in chapter 8, that will be at /usr/src/rtlinux/rtldoc-3.2-pre1/doc/html/GettingStarted.

4.2 Basic RTL system structure

RTLinux gives a Linux system real-time capability by inserting a scheduler which runs all the real-time tasks in order of their priority. In this paradigm standard Linux is run as the idle task. If there is no real-time task scheduled for execution, the standard Linux operating system runs.

RTLinux real-time programs are written as Linux kernel modules with specific RTL functions giving them real-time functionality. Depending on how the kernel module is written, the real-time task or tasks included might start automatically when the module is inserted into the kernel, or they might be started by commands from another source. Likewise the program might be stopped by removing the module from the kernel, or by external commands.

The fact that real-time programs must be written as kernel modules introduces some limitations and dangers you should be aware of.

Firstly, only the superuser is allowed to insert and remove kernel modules, since by doing so you are modifying the operating system and could easily do nasty things to the system and data in memory on the computer.

Secondly, there are some limitations to what you can do from the kernel; some standard C input and output functions are not supported in this context. It is often possible to allow standard functions by linking the libraries statically, but you run a risk of linking non-RT-safe code (i.e., code which introduces indeterminate blocking, which could result in an inability of the RT scheduler to ensure priority adherence). Actually, some of the current code has not been strictly evaluated for RT-safety; e.g. HDSO searches may not be. Boolean parsing is recursive and may take arbitrary time, but is always interruptible.

Thirdly, since the code you have inserted actually becomes part of the operating system, bugs in the code (like a thread loop with no sleep or make_periodic statements) can quite easily bring the system to a crashing halt. One thing you should do to prevent many cases of this is to make sure you are running the rtl_debug module whenever you are running a real-time module which hasn't been very thoroughly tested. On current installations, rtl_debug is loaded by default whenever the RTLinux system is started.

4.3 Anatomy of an RTL module

A Linux kernel module has several parts, each of which serves different functions. For now, the hello world example in the Getting Started document referred to in the introduction of this chapter will have to suffice.

4.4 Output from RTL processes

As alluded to above, many standard C input and output functions do not work in Linux kernel context. Even worse than that, kernel IO functions may or may not be real-time safe, and should for the most part be avoided. This leaves you with a couple of options for seeing output from real-time processes:

`rtl_printf` can be used for debugging purposes- it prints output to the console and, if configured to do so at compile time, the kernel print buffer, whose output can be

viewed by using the command `dmesg`. `rtl_printf` should not be used too extensively, as it will slow down your code considerably.

Data can be sent via an RT FIFO to a user-space program for display or recording, or it may be possible to use shared memory for a similar functionality.

NeuRoMod uses `rtl_printf` for debugging and initialization-type console output– output that will not be generated frequently, and therefore will not overtax the computer during standard operation. For all other data output, including more frequent status updates and data logging, RT FIFOs are used. These are preferred wherever it is important to preserve time order of messages, and that no messages get lost. It does require that the receiving program have extra code for parsing the data structures used in the FIFO. For situations where a particular piece of *current* data is needed *now,* looking at a shared memory structure is probably best.

5 The NeuRoMod Engine Architecture

This section describes the *engine* portion of the NeuRoMod control suite, in the SCCILS sense of "engine" described in section 2.4.1. The local control system is designed to allow modular addition of muscle models, dynamic muscle activation levels, and the continuous modification of activations and state transition conditions based on higher-level control inputs. This local control might be thought of as occupying the mid-level sequencing and low-level reactive portions of a three-layer architecture.

For those reading this as an appendix, this chapter is largely a reproduction of section 6.2 of Brandon Rutter's dissertation. It may be presented with a little more attention to the programmatic structure and code function, however.

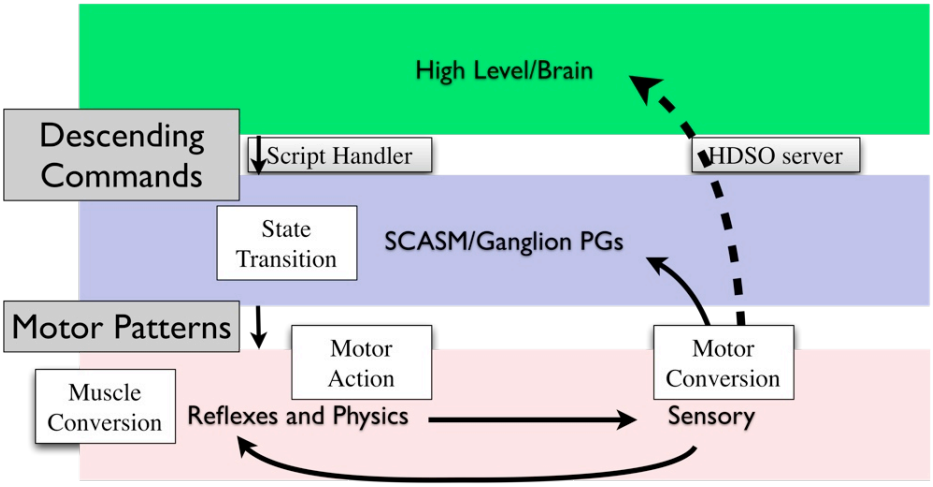


Figure 5-1 Location of real-time threads on the three-layer architecture backdrop. High-level control influences SCASM via descending commands; SCASM specifies motor patterns for lowest-level control. All arrows represent the flow of information between system elements. White rectangles are threads in the engine module; the state transition thread initiates state transitions and sets muscle activations and any low-level reflex rules, making it the thread that actually enacts SCASM. The light grey boxes are threads in the script module and handle SCCILS HDSO I/O: the script handler takes parameter modifications from the high-level (interface) and enacts them in lower levels. The HDSO server provides low-bandwidth access to most of the sensory data *and* parameter values in the lower levels of control.

The control code was implemented as a number of concurrently running real-time threads contained within two kernel modules. This allows for greater control modularity than single-process designs, and is a better structural match to the organization of the associated computational processes in the animal. The conceptual location of these

threads in a three-layer architectural description is shown in Figure 5-1. The same threads are shown in the context of a neuromechanical joint in Figure 5-2.

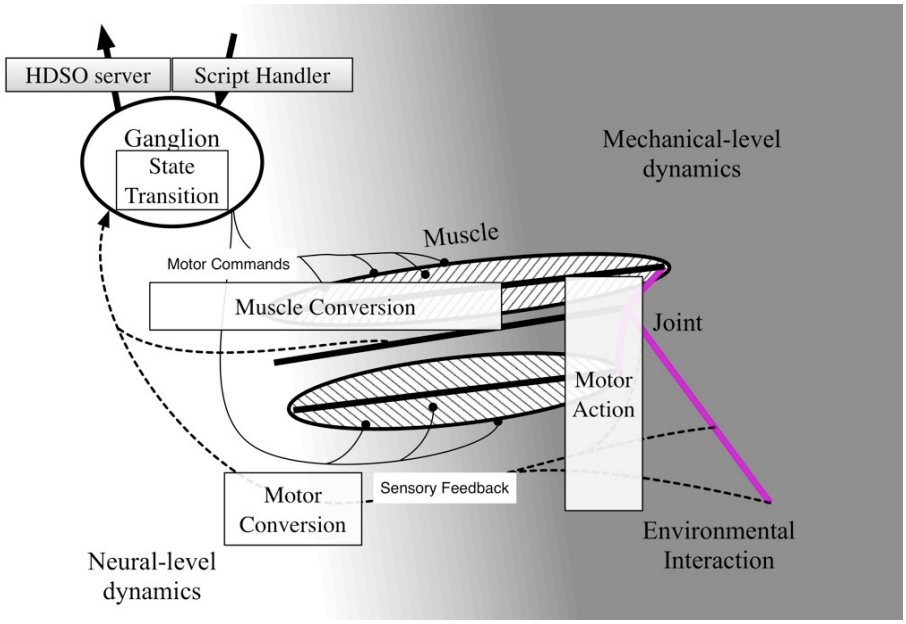


Figure 5-2 Approximate location of real-time threads in the context of the neuromechanical joint system. The script handler and HDSO server threads process higher-level interaction. The state transition thread implements SCASM, modeling the joint pattern generators. The muscle conversion thread models muscle and joint dynamics. The motor action thread handles low-level physics modeling and physical I/O. The "motor conversion" thread implements low-level sensory filtering and processing.

5.1 Enginemod: The Control Real-Time Module

As described above, enginemod is loaded and unloaded by the engineprog user-space command-line program. Engineprog sets up and initializes shared memory properly before launching enginemod. Enginemod runs four concurrent threads, but control takes place primarily in the Motor Action and State Transition threads. The Motor Action thread handles the feed-forward force control and runs up to a maximum rate given by computation and serial communication overhead, currently 197 Hz. The second SCASM control thread runs at a lower priority and handles the state transition and activation calculations. The update rates of these threads are both configurable; currently recommended settings of 100 Hz and 40 Hz respectively. All control and sensory data are stored in shared memory accessible both from real-time and user space, allowing online modification of the control. Each thread logs data every execution cycle via FIFO to the

"monitor" user space program that writes all data, including sensor readings, states and muscle activations, to files.

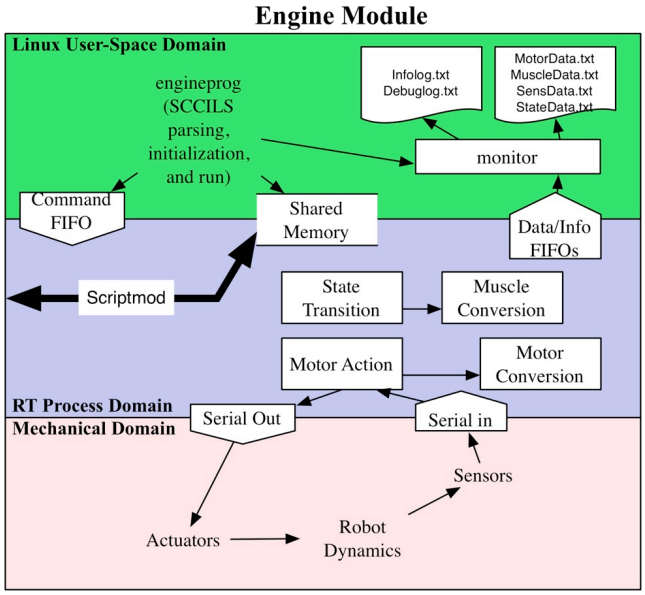


Figure 5-3 Enginemod thread event relationships. State Transition and Motor Action threads run at their own periodic timebases, and kick the conversion threads when necessary. All four threads access shared memory and produce output that goes through the data/info FIFOs. Many shared memory variables may be considered the "output" of one thread and the "input" for another. Scriptmod interacts with the controller through its connection to this shared memory.

5.1.1 The Motor Action Thread

The motor control thread handles low-level hardware I/O, and the feed-forward torque control calculations. The system is fast enough that angle feedback can be used in conjunction with the servomotors' proportional control to implement a crude, but stable, feed-forward torque control. For a given desired torque output, the servo is commanded to go to some delta of position from the current position; the proportional control used by the servo then generates a torque proportional to this delta. This system generates smooth compliant motion from an electrical motor system with relatively little computational overhead.

This is implemented as follows: in the servomotor, the current/torque command sent to the motor coils is at least primarily a proportional control, calculated as in Equations 5-1, where $\theta_{command}$ is the desired angular position and $\theta_{measured}$ is that reported by the servo's sensing device. In servo delta control, the command angle is calculated at a higher level

and at a high rate by using Equation 5-2. In this case θ_{err} , and thereby the motor current/torque, can be controlled as shown in Equations 5-3.

$$current\_command = P * \theta_{err}$$
$$\theta_{err} = \theta_{command} - \theta_{measured} \qquad \textbf{Equations 5-1}$$

for servo delta control, set

$$\theta_{command} = \theta_{measured} + \theta_{servo} \qquad \textbf{Equation 5-2}$$

then

$$\theta_{err} = \theta_{measured} + \Delta_{servo} - \theta_{measured} = \Delta_{servo},$$
$$current\_command = P * \Delta_{servo} \qquad \textbf{Equations 5-3}$$

5.1.2 Motor conversion thread: Sensory Processing

The motor conversion thread actually just does low-level sensory processing and filtering on the raw A/D sensory data that the Motor Action Thread records in shared memory. Angle data are used both in raw 8-bit form and after having a calibration applied, but no sensor filtering is done. The current data are filtered using the following IIR (Infinite Impulse Response) filter (after initialization).

filt_current = 0.05*current + 0.95 * prev_filt_current **Equation 5-4**

This IIR filter is hard-coded at this point and unconfigurable; providing SCCILS hooks for specifying filtering for any sensory signal would be an excellent addition to the present capability.

5.1.3 State transition control thread: SCASM

The state transition thread sets the state at each joint by checking the sensory thresholds described in the Boolean state transition rules. In so doing, this is the thread that implements the action switching of SCASM in these programs. It also calculates and sets the activations for the muscles at each joint as specified by the current action. Though most configurations currently use constant muscle activations, they are dynamically calculated and can use any available sensor and any of the generic functions available in the *utility.c* library.

To simulate sensory-motor control loop delays, the state transition thread is run at less than the maximum possible. A period of 25 milliseconds is used in the cockroach robot, corresponding to a hypothesized sensory loop delay of about 7.9 ms in the animal. As discussed in the future work section of the dissertation, this method has drawbacks because it also affects the fineness of sensory event detection, and the speed (and therefore fidelity) at which muscle dynamics are calculated. Motion was not obviously changed by adding the 25 ms delay here, however.

5.1.4 Muscle conversion thread: Muscle modeling

The muscle conversion task handles muscle model calculations, and sets the Δ_{servo} values used by the Motor Action thread. Here, the term "muscle model" refers to a crude mathematical model of a biological muscle, where the muscle is represented as a contractile element whose force at a particular level of activation is scaled by its current length and velocity. The length of the element, in turn, is determined by modeling the joint as a constant-radius pulley; an inaccurate but computationally simple joint model. This model of muscle activation is simpler than those intended to accurately model muscle dynamics (e.g. that of (Hill, 1970)), which can include a more complex arrangement of passive nonlinear stiffness and damping elements.

The servo delta at each joint is computed using the following relationship:

$$\Delta_{servo} = \sum_{muscles} [r \cdot f]$$
$$f = PS + activation \cdot FL \cdot FV \cdot f_{max}$$

Equation 5-5

where r is the radius of the pulley associated with each muscle, f is the current force of that muscle, PS, FL and FV are the values of the muscle's functions for passive stiffness, force-length activation scaling, and force-velocity activation scaling respectively, and f_{max} is a parameter intended to represent the maximum force which can be exerted by each particular muscle. PS, FL and FV can be calculated using constant, linear, parabolic or hyperbolic relationships, as implemented in *generic_function* in *utility.c*. As an example, the linear force-length characteristic FL was a function of nondimensionalized muscle length l':

$$FL = a + b \cdot l'$$

$$l' = \frac{(\alpha - \theta) r + l_0}{l_0} \qquad \textbf{Equation 5-6}$$

with a and b constant parameters, θ the joint angle, l_0 the "resting" length of the muscle and α the joint angle at which the muscle has this length.

Though it is not modeled explicitly and has not been quantified, there is also a force-velocity dependence inherent in the force control method used in the Motor Action thread. As a joint moves more quickly in the direction it is being driven, the average distance between the commanded position and the joint's actual position will become less, reducing the torque applied to the joint. The inverse is also true; decreasing joint velocity increases force. In the limit of the static case, θ_{err} is always equal to Δ_{servo} and will therefore actually be exerting the associated torque. At high speeds the low-level θ_{err} may never equal Δ_{servo}, since the low-level $\theta_{measured}$ will have moved closer to the calculated $\theta_{command}$ by the time the Δ_{servo} command reaches the servo. Additionally, there is some speed-output torque relationship for any physically existing motor and transmission, which also reduces maximum torque output at higher speeds.

5.2 Scriptmod: The SCCILS Interaction Real-Time Module

The low-level hooks for online SCCILS configuration changes are in the HDSO data store in the RT-Linux shared memory, accessed by enginemod. Scriptmod provides timed read and write access to these data through its two threads: the script handler and the HDSO data server.

The script handler thread receives HDSOmod data change commands (already translated from XML to the native engine representation) and implements them as closely as possible to the times specified. Once an HDSOmod has been applied, the script handler puts that HDSOmod on the Scriptlog FIFOs, marked with the time of actual implementation.

The HDSO server thread receives HDSOspec specifiers for requested HDSO data. It then reads the associated data from the shared memory HDSO store and writes them to the HDSOlog FIFOs, marked with the current time.

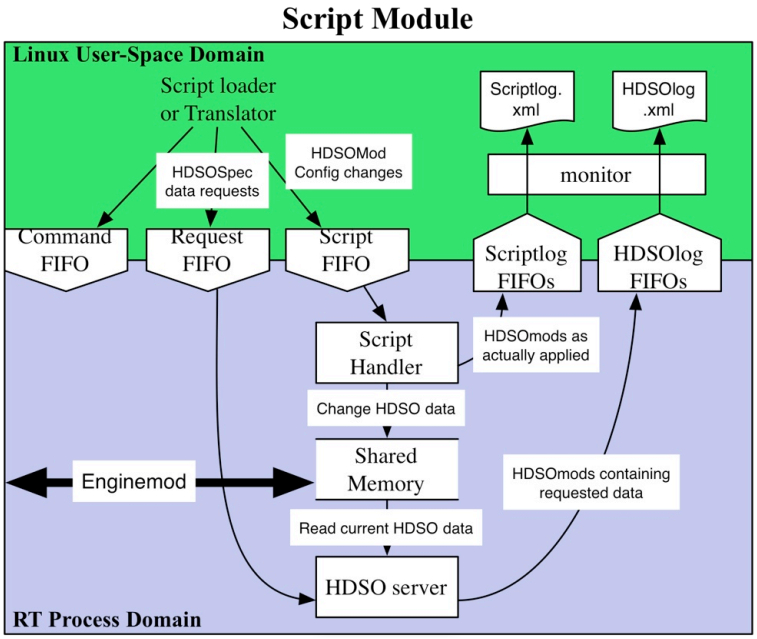


Figure 5-4 Scriptmod data flow. The script handler takes HDSOmod config changes via the Script FIFO, implements them at the desired time, and records actual implementation via the Scriptlog FIFOs (to Scriptlog.xml). The HDSO data server takes requests for HDSO data via the request FIFO, then looks up the specified data and writes it to the HDSOlog FIFOs in HDSOmod form. The "Shared Memory" in this figure is the shared memory HDSO store, a subset of the "Shared memory" block in the enginemod diagrams; thus the "connection" with enginemod.

5.3 User-space programs

Although the actual control computations take place in the real-time modules described above, there are a number of things that happen in programs in the "Linux User-Space Domain" elements of Figure 5-3 and Figure 5-4. This reflects the fact that user interaction, initial system configuration, and file parsing and I/O are better handled in the normal Linux programming environment.

Parsing and configuration are accomplished primarily by the *engineprog* command-line program. It is used to load and unload the RT-Linux modules, and makes heavy use of functions in the *userio.c* library. The *monitor* program logs data from RT data FIFOs to disk, and is run as a very high-priority user-space task. The *datalogging* command-line program is used to load and unload the daemon-like *monitor*, and it does some parsing of raw FIFO data written to disk by *monitor*.

The Python SCCILS translator and the command-line enginecli program are higher-level programs for running the engine interactively or executing experimental scripts.

Each of these makes use of *engineprog* and *monitor/datalogging* for the services they provide. The Python translator, of course, interacts with the graphical user interface via SCCILS, and via that SCCILS communication is capable of interacting with higher-level control of any nature.

5.4 Shared Memory and FIFO Organization

Information of many kinds is shared and passed between various RT tasks and user-space tasks using both shared memory and RT-FIFOs. Nearly all the configuration and operational data is stored in shared memory, allowing any process to look at and modify it at any time. Of course, it generally makes sense for only the sensor-reading real-time tasks to modify the sensor data values (for example), but there is no mechanism enforcing this. In order to make the operation a little more strictly controlled, communication of data between RT and user-space programs only takes place through RT-FIFO during actual engine operation. The user space programs do manipulate and initialize nearly all the shared memory structures before starting the real-time modules, however.

Shared memory is handled using the mbuff driver; each shared memory variable must be allocated/attached using mbuff_alloc at the beginning and detached using mbuff_free at the end of each program that uses that variable. In many of the programs involved in the NeuRoMod engine, you will find a lot of `goto: quitting_stuff` statements; generally wherever a fatal error is detected. This goes to the end of the main function, which has all of the mbuff_free statements and occasionally some other cleanup necessary before the program exits. If the program hangs or exits without freeing those shared memory variables, they will persist as "zombie" variables and the mbuff driver kernel module will not unload. Occasionally running an unload version of a program (such as "`enineprog u`") will kill zombie variables, but sometimes this will not work. In such cases it is necessary to reboot the computer to kill the zombies. This is usually recommended, because although the initialization routines *should* give you a clean slate, it is possible (and this has happened) that in some cases particular zombie data might not be zapped and cause confusion. Usually, it is best to kill the zombies.

One important note to understand when using mbuff is that the pointer address space is different in the kernel and user-space environments. Therefore, shared pointer variables

will not point to the same place in both contexts. This is why many of the shared memory structures use array indices rather than pointers. It is considerably messier than using pointers would be, but it does work.

An overview of the shared memory structure is shown in Figure 5-5. No one program accesses all shared memory variables; these are all the variables used in NeuRoMod. There are a number of one-element variables, many of which specify the size of shared data arrays. The parameter struct arrays provide structures for accessing various configuration and control data, much of which is actually stored in the monolithic HDSO database's data storage block.

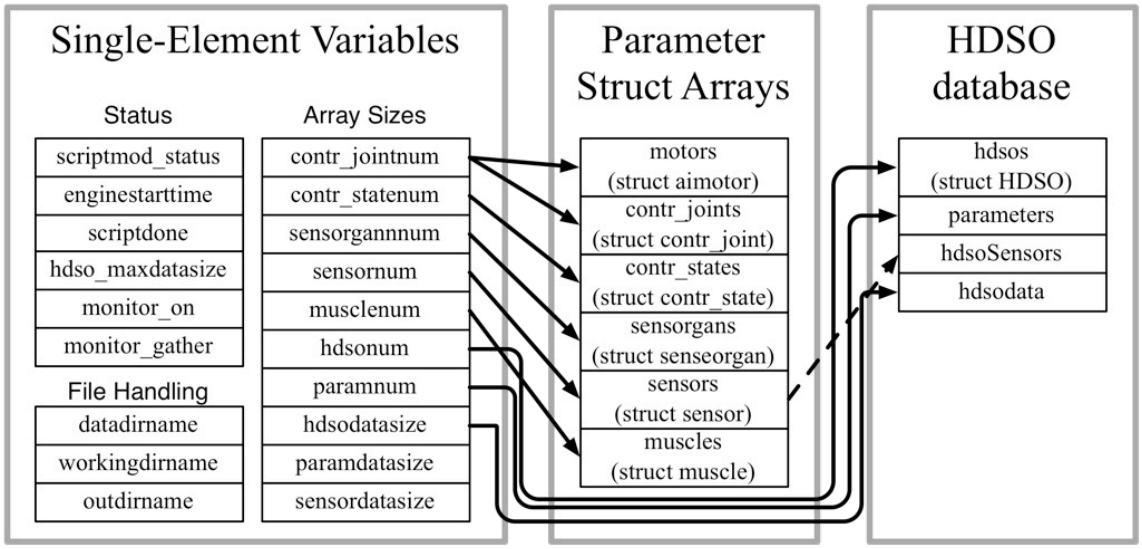


Figure 5-5 Shared memory overview for NeuRoMod. Status variables help to coordinate the function of different processes and programs, file handling variables are used primarily by enginecli, datalogging and monitor in user space. The array size variables are associated with the arrays they point to, and are necessary to allocate the proper array size in each program accessing shared memory. Param- and sensordatasize are only used in initialization to double-check consistency with hdsodatasize. The struct data type associated with each parameter struct array is given in parantheses in the parameter struct arrays block. The HDSO database really only consists of hdsos and hdsodata; parameters and hdsoSensors are convenience pointers into the hdsos array, but valid only in user space.

The FIFOs used in NeuRoMod are shown in Figure 5-6; note that the "input" FIFOs that carry info from user space to RT kernel space are all single-input-single-output since only one of enginecli or server.py is active at any given time. The output FIFOs can each carry information from multiple RT processes, and since they are duplicated (e.g. there

are INFO and INFO_2 FIFOs) they can carry information both to the local monitor data logger and through server.py to SCCILS interfaces.

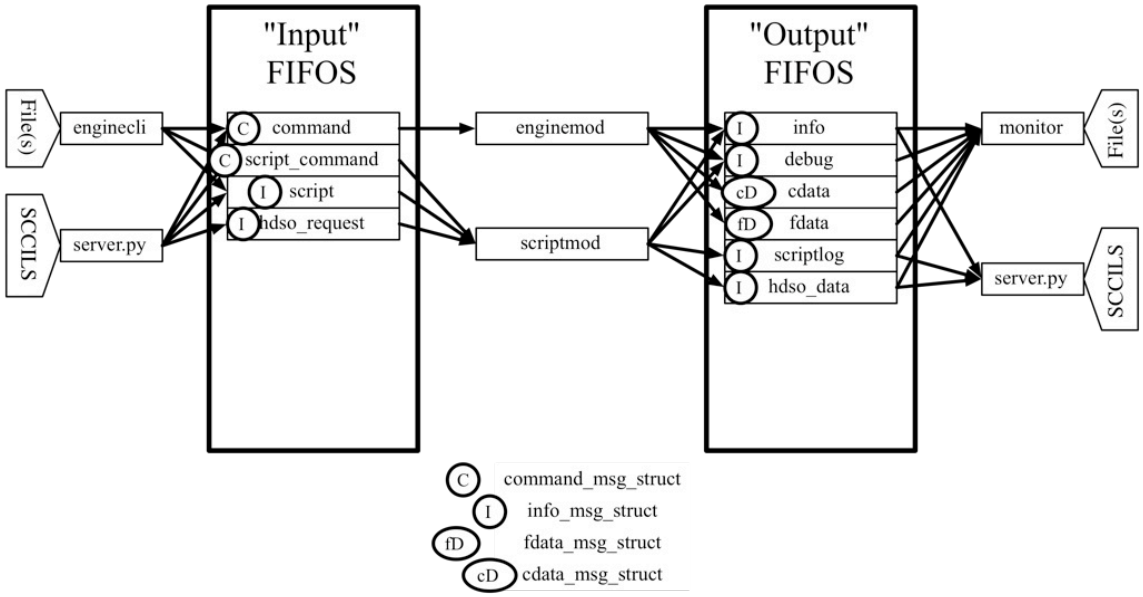


Figure 5-6 FIFOs used in NeuRoMod; the struct type on each FIFO is labeled according to the legend at the bottom. The script, hdso_request, scriptlog and hdso_data FIFOs each actually carry interleaved info_msg_structs and raw data associated with the HDSOmod defined in the preceding struct.

6 Writing Programs to Interact with the Engine

6.1 Introduction

I have, for the most part, tried to document the code in the various NeuRoMod engine program and library files pretty well. So, many questions can be answered by looking at the code, and searching for something you want to find (use control-s to search for something in emacs or xemacs text editors). You may wish to have a place to start, however, to familiarize yourself with the structure of the programs and meanings of the variables. I have started using Doxygen (www.doxygen.org) to create documentation for the source code in HTML. The current documentation can be found in the repository; checked out in the installation as doc-romod/engine/html/index.html . An abbreviated version of this documentation is also contained in an appendix to Brandon Rutter's dissertation. The same information is available in the source code, but even I find the HTML pages generated by Doxygen useful, and the hyperlinks provided between functions and files make navigation considerably more pleasant than searching through code in a text editor. If you plan to work on the code and update the Doxygen documentation, note that the Doxyxfile configuration in neuromod/engine currently assumes that neuromod/doc is linked to doc-romod; however this is only the default if you have checked out the entire romod repository; it won't work out of the box with the installation instructions in this manual.

There are a number of ways to interact with the engine using a further program added to the system, and a number of things to do with this interaction:

1) Through SCCILS, using the Python translator. This is currently used for the UI, and could be used for higher-level control and optimization programs. Data bandwidth is limited in this method, however, since the SCCILS layer does add considerable communication overhead. This method is not covered here, but anything implementing SCCILS should be able to make use of this way to add function; consult the example of the Python UI and server, and the SCCILS specification. It may be wise to consider this

option first, since SCCILS provides a better-defined API than other, lower-level communication with the engine.

2) Through real-time FIFOs from user space. Monitor is a good example of this, though its function is essentially read-only. Such programs can be local, and high-bandwidth. One could use both SCCILS functionality and higher-bandwith (and currently broader kinds of) data in this kind of program.

3) Through shared memory; either user-space and higher-level programs, or replacing or augmenting engine-level function. Examples: engineprog for configuration/initialization/setup. Scriptmod for the addition of functionality.

6.2 Writing an RTLinux module

Anything dealing with additional hardware or requiring strict interactive scheduling should be done in RT. To write your own RTL control modules, I suggest first familiarizing yourself with some of the RTL functions and the hello_world example, by reading

/usr/src/rtlinux/rtldoc-3.2-pre1/doc/html/GettingStarted/index.html

The RTLinux examples are in /usr/src/rtlinux/rtlinux-3.2-rc1/examples . The frank example demonstrates the FIFO command handling used in NeuRoMod.

Once you are familiar with the basics (or perhaps before), you can start taking a look at scriptmod.c as an example of relatively simple function to extend the functionality of the already-loaded and already-running engine. Honestly, scriptmod is not all that simple but it can be significantly simplified by ignoring the HDSO data server FIFO handler and thread.

Essentially, though, an RTLinux kernel module consists of the following sections:

includes

module-global variables, including thread variables and mbuff shared memory pointers

FIFO handler functions (optional)

function(s) for RT thread(s) containing:

initialization of local variables; possibly setting the thread to periodic execution or allowing floating-point calculations.

thread execution loop; must include either sleep statement(s) (usually clock_nanosleep) or pthread_wait_np (if it's been set to automatic periodic execution) in order to allow Linux some execution time.

init_module function containing:

FIFO initialization

shared memory attachment

possible linking of kernel-space pointer variables to shared memory (see enginemod)

create and possibly start the thread(s)

cleanup_module function containing:

FIFO destruction

thread halting/destruction

shared memory unlinking

Of the RT modules in NeuRoMod, scriptmod is probably the easier to understand. When developing realtime modules, keep in mind that at the Linux command prompt you can always enter "lsmod" to see what modules are currently loaded. If things are acting strangely, sometimes it is because somebody left an external module running after they quit the main engine.

6.3 Writing a user space program

User and file I/O, as well as higher-level control that has less stringent scheduling requirments (and may use more complex, non-deterministic-time calculations), should be done in user space. The most straightforward examples of user space interaction with the engine are datalogging.c and monitor.c. A user space program that interacts with the engine will probably have the following elements:

includes

main function, including:

variable declarations & initialization

possible input parsing, mode determiniation, and environment checking

shared memory attachment/allocation

shared memory status variable initialization

Possible file parsing

RT-FIFO opening (note: in some cases it is better to only open FIFOs when necessary, and close them after each use)

Execution loop, possibly including:

- shared memory status updates
- user interaction
- FIFO data processing (from RT)
- File output
- Control computations and FIFO output to RT

quitting_stuff, including:

- file closing
- RT-FIFO closing
- shared memory status updating (to indicate this program is done)
- shared memory detachment/deallocation
- return status

7 Using Linux

7.1 Basic commands

Note on doing things as the super-user: On the RTLinux machine where you run the engine, a number of the commands will only work if you do them as the "super-user". In UNIX and similar systems, the super-user, usually named "root", is essentially capable of doing anything. It is therefore dangerous to do things as the super-user, since you can really mess things up!

The standard way to run a command as the super-user is 'sudo [command]'; this will work for commands you have been given the rights to run as super-user. For increased security you will often be required to type your password. Though it is routine to run a number of commands via 'sudo' to operate NeuRoMod, try not to become too comfortable doing this. Be careful with sudo, because it is possible to really mess things up if you run the wrong commands as super-user.

Basic UNIX commands:
Typing the "tab" key in a terminal will auto-complete a command or file name if it is unique.
The standard syntax for a UNIX command line program is
command -options [arguments]
When describing a set of commands, it is standard to have a new command on each line; i.e.
cd ~
ls
Are two separate commands; hit "return" after each. (Those two commands will take you to your home directory and show you what's in it).

man [command]	displays a manual page for command
info [command]	displays either more in-depth info for a command, if available, or the manual page for the command
apropos [keyword]	displays a list of commands dealing with keyword

cd [dir]	change directory to dir
pwd	print working directory to screen
ls	list the contents of the current directory
~	shortcut for your home directory
~rutter	shortcut for rutter's home directory
.	shortcut for working directory
..	shortcut for parent of working directory
mkdir [dir]	create directory [dir]
mv [path] [path]	move file or directory from one place to another
cp [path] [newpath]	copy a file
cp -r [path] [newpath]	copy directories recursively
rm [path]	delete a file
rm -r [path]	recursively delete a directory
rm -rf [path]	recursively delete a directory without asking for confirmation
*	wildcard character
rm -r *	something to type only if you know what you're doing – recursively removes all files and directories in the working directory
sudo	do something as the root user, or "superuser"
startx	start the X screen display server
passwd	change your password
emacs	a text editor
xemacs	same text editor, works better in X
halt	immediately shut down the system
reboot	immediately reboot
shutdown	a command that allows you to warn users who may be logged in before rebooting or shutting down
who	displays who is logged on to the machine, and from where

ssh user@hostname	secure shell – log into hostname as user
scp	secure copy – copy documents from one machine to another securely. to copy an entire data directory poo on exuberance to your home directory on your local machine, type "scp -r user@exuberance:~rutter/llcnew/data/poo ~/."
rsync	Remote sync

7.2 Online tutorial

http://www.linux.org/lessons/beginner/index.html

8 Installing RT-Linux

This consists merely of a copy of the installation log kept for Digistation, the most recently and straightforwardly configured NeuRoMod engine-capable machine. These directions provide a complete outline, but a substantial amount of ability to run Debian installers and do Linux kernel configuration is assumed. If you run into trouble, which is most likely in the kernel configuration stage in the selection of the drivers necessary for your machine, there is considerable help available on the internet- though it can take some time to dig up and understand properly. Be patient, and possibly enlist the help of a Linux/Unix geek who will think it's cool to configure a real-time kernel. In case it is useful, both this and the configuration log for minishadowfax (which is longer, and not quite so straightforward) are included in the doc-romod subversion repository.

digistation admin log

Intended system use: SCASM robot runner; sibling/update/replacement of minishadowfax

500 MHz Pentium III

Optical NIC: National Semiconductor DP83820 10/100/1000 driver

hda: IBM DTTA-371440 /, swap
hdc: BCD-48SB CD-ROM
hdd: WDC WD1200JB-00FUA0 /home

Video card:
agpgart: Detected an Intel 440BX Chipset.
agpgart: AGP aperture is 64M @ 0xe0000000

12/4/2007 BLR
Installed Debian Etch from net install image debian-40r1-i386-netinst.iso
Basic system
Also added basics and basic window manager: sshd, (x)emacs, xfce, sux, modutils
Also added Python+wxWidgets for the robot UI, and libxml2-dev,utils,doc
and setserial for engine

12/6/2007 BLR
Continuing basic setup before compile & install of RTLinux

configure sudoers file:

/etc/sudoers
#

```
# This file MUST be edited with the 'visudo' command as root.
#
# See the man page for details on how to write a sudoers file.
#

Defaults        env_reset

# Host alias specification

# User alias specification
User_Alias BOTRUNNERS = rutter, markw, bkt2, jbender

# Cmnd alias specification

# User privilege specification
BOTRUNNERS ALL=(ALL) ALL
root    ALL=(ALL) ALL
---------------------------------
```

That appears to work. Now disable root ssh login:

```
edit /etc/ssh/sshd_config
  Find the line in the "# Authenication:" section which reads:
    PermitRootLogin yes
  This is "yes" by default, and will allow someone to ssh into the
     machine using the root account.
  First change this option to "no".
Now the sshd service needs to be restarted, so that the config file
    is read off of the disk:
  /etc/init.d/ssh restart
Test to make sure it is doing the right thing, and that everyone
    who should be able can still get into the machine
```

All appears to work.

Now, enable X11 forwarding over SSH as default:

```
in /etc/ssh/sshd_config change X11Forwarding line to "yes"
(actually, the above now appears to be the default)
in /etc/ssh/ssh_config change ForwardX11 to "yes" and uncomment

use
ps aux | grep sshd
to find root sshd process, then kill -HUP that process

(this kill step actually unnecessary, since sshd was already correctly configured)


Edit /etc/apt/sources.list to remove the cdrom entry

===============
```

Now do the RTLinux thing.

1) compile and install gcc 2.95.3 from gcc.gnu.org

First, make sure you've got any compiler at all; I installed Etch gcc, which is 4.1

Also, install flex to work around a bug in binutils make process

install binutils 2.15, from http://ftp.gnu.org/pub/gnu/binutils/

tar -xjf binutils-2.15.tar.bz2 (might have to rename the tarball)
cd binutils-2.15
./configure
make
sudo make install

tar -xvzf gcc-2.95.3.tar.gz
follow directions in gcc-2.95.3/install/index.html

mkdir gcc_objdir
cd gcc_objdir
../gcc-2.95.3/configure --with-as=/usr/local/bin/as --with-ld=/usr/local/bin/ld

make bootstrap

sudo make install

This puts gcc 2.95.3 (as "gcc") in /usr/local/bin, so it's the default (since this is
before /usr/bin in the default path) Yay!

=====
2) compile custom kernel, basing config choices on those from minishadowfax.

First get the sources in the right places:

put the following in /usr/src:
linux-2.4.29 (vanilla from kernel.org) (symlink "linux" to this)
tarball from http://www.rtlinux-gpl.org/cgi-bin/viewcvs.cgi/
(saved and unpacked in /usr/src/rtlinux/)
rt_math-1.0 (from tarball from minishadowfax, originally from http://sourceforge.net/projects/mca2)

Now configure and compile the custom kernel (as root)
cd /usr/src/linux

make clean
make mrproper
make xconfig

do not prompt for alpha modules/code
enable loadable module support
do not set version information on module symbols
yes: kernel module loader

Processor type/features
PentiumIII/Celeron(Coppermine)
Machine Check Exception
(turned off SMP support)
General Setup disabled support for hot-pluggable devices (includes PCMCIA
enabled kernel support
 for all available binaries
disabled power management support
RTC stores time in GMT

NO parallel port support

No Plug and Play support

Block devices only Normal Floppy (Though Ram disk might be useful at some point)

Network: Packet socket: mmaped IO
Network packet filtering
Socket filtering

IDE,ATA...
IDE,ATA, and ATAPI block devices
turned off use multi-mode by default
turned on CMD640 bugfix/support and enhanced support
 left on: use PCI DMA by default when available
 turned off INtel PIIXn chipsets support
turned off RZ1000 chipset bugfix/support
SCSI support off

Network devices:
Dummy net driver as module
Enable 10/100
      DECchip Tulip support
Ethernet 1000
            National Semiconductor DP83820 support

Character Devices: Module Enhanced Real Time clock Support rtc.o
turned off Direct Rendering Manager turned off all AGP chipset support except
 Intel 440LX/BX/GX

File systems
Reiserfs
EXT3
DOS FAT
MSDOS
VFAT turned off Virtual memory file system support
NTFS
XFS
Network all but SMB (unix extensions enabled) off
UDF read-only

Sound support off
USB support off

make dep

edit EXTRAVERSION in Makefile = pre-rtl_12-6-07
make bzImage (or nohup make bzImage &)

Crap, that didn't work; got a bunch of errors starting with:
{standard input}: Assembler messages:
{standard input}:730: Error: suffix or operands invalid for 'mov'

Apparently this has to do with:
"...The new i386/x86_64 assemblers no longer accept instructions for moving

between a segment register and a 32bit memory location,..."
(from http://kerneltrap.org/node/5785)
and indeed minishadowfax has gnu assembler version 2.15, whereas here it's 2.17

Okay, so install that version of binutils and re-compile GCC 2.95.3 with it...
This is actually included above, so that this day is represented as a single
successful series of events.

After having done that and (successfully!) re-done all of the above:

make modules
make modules_install
copy the config file for records
copy the image
 /usr/src/linux/arch/i386/boot/bzImage to /boot/vmlinuz-2.4.29_pre-rtl_12-6-07
copy the .config to /boot/config-imagename (i.e. config-2.4....)

edit /boot/grub/menu.lst to have a 20 second timeout for boot loading
update-grub
reboot

all appears to work, except the ethernet interface didn't come up automatically.
Apparenlty this is one of the things the 2.6 kernel supports- in interfaces eth0
was listed as allow-hotplug, and that was enough. but not so for a 2.4 kernel.

To fix, edit /etc/network/interfaces

modify the following line to be:
auto lo eth0

And now it works.

but whoops, x doesn't work on the console.

dpkg-reconfigure xserver-xorg, leave everything the same but change the
mouse port to /dev/psaux (/dev/mice apparently not supported by 2.4 kernel)

yup, that works.

Okay, now do rtlinux

cd /usr/src/linux
change EXTRAVERSION back to blank
make clean
make mrproper
patch -p1 < ../rtlinux/rtlinux-3.2-rc1/patches/kernel_patch-2.4.29-rtl3.2-rc1
copy config file from above to .config
make xconfig

not enabling RTLinux HAL
leave everything else as before. Save and exit.

make dep

edit EXTRAVERSION = -rtl3.2-rc1_12-6-07
make bzImage

make modules
make modules_install
copy image and config to /boot/ with version name 2.4.29-rtl3.2-rc1_12-6-07
update-grub

reboot

All appears to work! Now compile rtlinux modules

cd to /usr/src/rtlinux/rtlinux-3.2-rc1

make checklinux
succeeded

make xconfig
turned off RTLinux V1 API support
left everything else as default

make dep
make
make install

make regression - all OK, no warnings!

now install rt-math
cd /usr/src/rtlinux/rt-math-1.0
edit makefile to set include paths correctly:
"rtl" -> "rtlinux/rtlinux-3.2-rc1"
"rtlinux" -> "linux"

make

cp rt_math.o /usr/rtlinux/modules
cp rt_math.o /lib/modules/2.4.29-rtl3.2-rc1_12-6-07/misc
cp include/rt_math.h /usr/local/include/

and while we're at it, copy rtl_debug module from rtlinux-3.2-rc1/debugger/rtl_debug.o
to these same places (not include)

Do the same with rtlinux-3.2-rc1/drivers/rt_com.0.5.5/

now attempting to make /romod/trunk/engine

compiler isn't finding libxml/*.h header files

-ah yes, make the necessary symlink:

ln -s /usr/include/libxml2/libxml /usr/include/libxml

now scriptmod make complaining that ld can't find -lc; added -L/usr/lib to fix this.
not sure what was making this work on minishadowfax.

Also getting ": undefined reference to 'errno'" errors from make of enginecli --appears new
libc doesn't like old "extern int errno" usage, needs #include <errno.h> instead

This compiles!

hrm. having troubles getting rt_com to work, and FIFOs don't appear to be working right

This was due to rt_com.o not loading, due to it attempting to gain control of more than just the first serial port. Need to figure out what exactly is necessary to get this to work (I did setserial /dev/ttyS[0-3] uart none)

12/11/2007 BLR+JAB
==========
To get the Python GUI working, changed symlink for python in /usr/bin to point to /usr/bin/python2.5

Also, needed to get a more up-do-date wxWidgets to work with Python2.5:

add

deb http://apt.wxwidgets.org/ etch-wx main
deb http://debs.astraw.com/ gutsy/

to /etc/apt/sources.list

apt-get update
apt-get install python-wxgtk2.8 python-wxvalidatedtext python-setuptools

Also had to make symlinks in /usr/lib/python2.5/site-packages to the above packages, which otherwise installed in /usr/lib/python2.4/site-packages

cd /usr/lib/python2.5/site-packages
ln -s ../../python2.4/wx-2.8-gtk2-unicode
ln -s ../../python2.4/wxversion.py
ln -s ../../python2.4/pkg_resources.py
ln -s ../../python2.4/setuptools
ln -s ../../python2.4/wx.pth
ln -s ../../python2.4/python-support.pth
ln -s ../../python2.4/setuptools.pth
ln -s ../../python2.4/wxvalidatedtext

3/15/08 JAB
==========
changed python-wxvalidatedtext package:

apt-get remove python-wxvalidatedtext
apt-get install python-motmot-wxvalidatedtext
cd /usr/lib/python2.5/site-packages
rm wxvalidatedtext
ln -s ../../python2.4/motmot
ln -s ../../python2.4/motmot.wxvalidatedtext.egg-info
ln -s /usr/share/pycentral/python-motmot-wxvalidatedtext/site-packages/motmot.wxvalidatedtext-0.5.1.dev_r409-py2.4-nspkg.pth
ln -s /usr/share/pycentral/python-motmot-wxvalidatedtext/site-packages/motmot.wxvalidatedtext-0.5.1.dev_r409-py2.5-nspkg.pth

11 Bibliography

Abbas, J., & Full, R. J. (2000). Neuromechanical interaction in cyclic movements. In J. M. Winter & P. E. Crago (Eds.), *Biomechanics and neural control of posture and movement* (pp. 177-191). New York: Springer-Verlag.

Akay, T., Bässler, U., Gerharz, P., & Büschges, A. (2001). The role of sensory signals from the insect coxa-trochanteral joint in controlling motor activity of the femur-tibia joint. *Journal of neurophysiology, 85*(2), 594-604.

Akay, T., & Büschges, A. (2006). Load signals assist the generation of movement-dependent reflex reversal in the femur-tibia joint of stick insects. *J Neurophysiol, 96*(6), 3532-3537.

Akay, T., Haehn, S., Schmitz, J., & Büschges, A. (2004). Signals from load sensors underlie interjoint coordination during stepping movements of the stick insect leg. *Journal of neurophysiology, 92*, 42-51.

Akay, T., Ludwar, B. C., Goritz, M. L., Schmitz, J., & Büschges, A. (2007). Segment specificity of load signal processing depends on walking direction in the stick insect leg muscle control system. *Journal of Neuroscience, 27*(12), 3285-3294.

Allen, T. J., Quinn, R. D., Bachmann, R. J., & Ritzmann, R. E. (2003, Oct). *Abstracted Biological Principles Applied with Reduced Actuation Improve Mobility of Legged Vehicles.* Paper presented at the IEEE/RSJ international conference on intelligent robots and systems (IROS 2003), Las Vegas, NV.

Alsop, D. W. (1978). Comparative analysis of the intrinsic leg musculature of the American cockroach, *Periplaneta americana* (L.). *Journal of Morphology, 158*(2), 199-241.

Altendorfer, R., Moore, N., Komsuolu, H., Buehler, M., Brown, H. B., McMordie, D., Saranli, U., Full, R., & Koditschek, D. E. (2001). RHex: A biologically inspired hexapod runner. *Autonomous Robots, 11*(3), 207-213.

Asada, H., & Slotine, J.-J. E. (1986). *Robot Analysis and Control*. New York: Wiley-Interscience.

Bachmann, R. J., Nelson, G. M., Flannigan, W. C., Quinn, R. D., Watson, J. T., & Ritzmann, R. E. (1997, May). *Design of a Cockroach-Like Hexapod Robot.* Paper presented at the 11th VPI & SU Symposium on Structural dynamics and control, Blacksburg, VA.

Bässler, U. (1988). Functional Principles of Pattern Generation for Walking Movements of Stick Insect Forelegs: The Role of the Femoral Chordotonal Organ Afferences. *J Exp Biol, 136*(1), 125-147.

Bässler, U. (1993). The femur-tibia control system of stick insects – a model system for the study of the neural basis of joint control. *Brain Research Reviews, 18*, 207-226.

Bässler, U., & Büschges, A. (1998). Pattern generation for stick insect walking movements – multisensory control of a locomotor program. *Brain Research Reviews, 27*, 65-88.

Beer, R. D., & Gallagher, J. C. (1992). Evolving Dynamical Neural Networks for Adaptive Behavior. *Adaptive Behavior, 1*(1), 91-122.

Bender, J. A., Rutter, B. L., Simpson, E. M., Papay, T., Parker, D. A., Taylor, B. K., Quinn, R. D., & Ritzmann, R. E. (2008, June 1-6). *Robotic Control based on Cockroach 3D Leg Kinematics.* Poster presented at the Fourth International Symposium on Adaptive Motion of Animals and Machines Cleveland, OH.

Bender, J. A., Simpson, E. M., & Ritzmann, R. E. (2008). Increased stereotypy of leg movement patterns with increased walking speed in the cockroach. Program no. 198.6. *2008 Neuroscience Meeting Planner*.

Bender, J. A., Simpson, E. M., & Ritzmann, R. E. (2009 (In Prep)). Quantitative behavioral analysis reveals a distinction between trotting and ambling gaits in the cockroach, *Blaberus discoidalis*. *to be submitted to Arthropod Struct Dev*.

Black, P. E. (2008). finite state machine. *Dictionary of Algorithms and Data Structures* Retrieved July 5, 2009, from http://www.itl.nist.gov/div897/sqg/dads/HTML/finiteStateMachine.html

Boggess, M., Schroer, R., Quinn, R., & Ritzmann, R. (2004, Apr). *Mechanized Cockroach Footpaths Enable Cockroach-like Mobility.* Paper presented at the International conference on robotics and automation; 2004 IEEE, New Orleans, La.

Borgmann, A., Hellekes, K., & Büschges, A. (2009). Characterization of intersegmental sensory influences in the stick insect walking system. Program no. 564.15. *2009 Neuroscience Meeting Planner*.

Branicky, M. S. (1995). *Studies in hybrid systems: modeling, analysis and control.* Massachusetts Institute of Technology, Cambridge.

Branicky, M. S. (1997, 09-04-1999). Hybrid Systems Retrieved July, 2009, from http://dora.cwru.edu/msb/personal/hybrid.html

Brezina, V., Orekhova, I. V., & Weiss, K. R. (2000). The Neuromuscular Transform: The Dynamic, Nonlinear Link Between Motor Neuron Firing Patterns and Muscle Contraction in Rhythmic Behaviors. *Journal of neurophysiology, 83*(1), 207-232.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of, 2*(1), 14-23.

Brooks, R. A. (1989). A Robot that Walks; Emergent Behaviors from a Carefully Evolved Network. *Neural computation, 1*(2), 253-262.

Brooks, R. A. (1991). New Approaches to Robotics. *Science, 253*(5025), 1227-1232.

Bucher, D., Akay, T., DiCaprio, R. A., & Büschges, A. (2003). Interjoint coordination in the stick insect leg-control system: The role of positional signaling. *Journal of neurophysiology, 89*, 1245-1255.

Büschges, A. (2005). Sensory Control and Organization of Neural Networks Mediating Coordination of Multisegmental Organs for Locomotion. *Journal of neurophysiology, 93*(3), 1127-1135.

Büschges, A., Ludwar, C., Bucher, D., J, S., & DiCaprio, R. A. (2004). Synaptic drive contributing to rhythmic activation of motorneurons in the deafferented stick insect walking system. *European Journal of Neuroscience, 19*, 1-7.

Büschges, A., Schmitz, J., & Bässler, U. (1995). Rhythmic patterns in the thoracic nerve cord of the stick insect induced by pilocarpine. *Journal of Experimental Biology, 198*, 453-456.

Buschmann, A. (2000a, March 2000). Home of Tarry I & II: design of the walking machine Tarry II, from http://www.tarry.de

Buschmann, A. (2000b, March 2000). Home of Tarry I & II: frequently asked questions about Tarry, from http://www.tarry.de

Chan, K. H., & Tidwell, P. M. (1993). *The reality of Artificial Life: Can computer simulations become realizations?* Paper presented at the Third International Conference on Artificial Life (submitted).

Choi, J.-u., Rutter, B. L., Kingsley, D. A., Ritzmann, R. E., & Quinn, R. D. (2005, Jul). *A Robot with Cockroach Inspired Actuation and Control.* Paper presented at the IEEE / ASME international conference on advanced intelligent mechatronics, Monterey, CA.

Collins, S. H., Wisse, M., & Ruina, A. (2001). A Three-Dimensional Passive-Dynamic Walking Robot with Two Legs and Knees. *The International Journal of Robotics Research, 20*(7), 607-615.

Crespi, A., Badertscher, A., Guignard, A., & Ijspeert, A. J. (2004). AmphiBot I: an amphibious snake-like robot. *Robotics and autonomous systems, 50*(4), 163-175.

Cruse, H. (1976). The function of the legs in the free walking stick insect, *Carausius morosus*. *Journal of Comparative Physiology A: Neuroethology, Sensory, Neural, and Behavioral Physiology, 112*(2), 235-262.

Cruse, H. (1985). Which Parameters Control the Leg Movement of a Walking Insect?: II. The Start of the Swing Phase. *J Exp Biol, 116*(1), 357-362.

Cruse, H. (1990). What Mechanisms Coordinate Leg Movement in Walking Arthropods. [Review]. *Trends in Neurosciences, 13*(1), 15-21.

Cruse, H., & Bartling, C. (1995). Movement of Joint Angles in the Legs of a Walking Insect, Carausius-Morosus. *Journal of Insect Physiology, 41*(9), 761-771.

Cruse, H., Dean, J., Muller, U., & Schmitz, J. (1991). *The stick insect as a walking robot.* Paper presented at the Fifth International Conference on Advanced Robotics, 'Robots in Unstructured Environments' (91 ICAR).

Dickinson, M. H., Lehmann, F. O., & Sane, S. P. (1999). Wing rotation and the aerodynamic basis of insect flight. *Science, 284*(5422), 1954-1960.

Dresden, D., & Nijenhuis, E. D. (1953). On the anatomy and mechanism of motion of the mesothoracic leg of *Periplaneta americana*. *Proc. Acad. Sci. Amster., 56*, 39-47.

Dürr, V., Krause, A. F., Schmitz, J., & Cruse, H. (2003). Neuroethological Concepts and their Transfer to Walking Machines. *The International Journal of Robotics Research, 22*(3-4), 151-167.

Dürr, V., Schmitz, J., & Cruse, H. (2004). Behaviour-based modelling of hexapod locomotion: linking biology and technical application. *Arthropod structure & development, 33*(3), 237-251.

Ekeberg, Ö., Blümel, M., & Büschges, A. (2004). Dynamic simulation of insect walking. *Arthropod structure & development, 33*(3), 287-300.

Ekeberg, O., & Grillner, S. (1999). Simulations of neuromuscular control in lamprey swimming. *Philosophical Transactions of the Royal Society of London Series B-Biological Sciences, 354*(1385), 895-902.

Espenschied, K. S., & Quinn, R. D. (1994, March 20-24). *Biologically-Inspired Hexapod Robot Design and Simulation.* Paper presented at the AIAA Conference on Intelligent Robots in Field, Factory, Service and Space, Houston, Texas.

Espenschied, K. S., Quinn, R. D., Beer, R. D., & Chiel, H. J. (1996). Biologically based distributed control and local reflexes improve rough terrain locomotion in a hexapod robot. *Robotics and autonomous systems, 18*(1-2), 59-65.

Espenschied, K. S., Quinn, R. D., Chiel, H. J., & Beer, R. D. (1993). Leg coordination mechanisms in stick insect applied to hexapod robot locomotion. *Adaptive Behavior, 1*(4), 455-468.

Full, R. J., & Tu, M. S. (1990). Mechanics of 6-Legged Runners. *Journal of Experimental Biology, 148*, 129-146.

Garcia, M., Chatterjee, A., Ruina, A., & Coleman, M. (1998). The simplest walking model: stability, complexity, and scaling. *Journal of Biomechanical Engineering, 120*(2), 281-288.

Gassmann, B., Scholl, K.-U., & Berns, K. (2001, September, 2001). *Behavior control of LAURON III for walking in unstructured terrain.* Paper presented at the Intl. Conference on Climbing and Walking Robots (CLAWAR '01), Karlsruhe, Germany.

Gat, E. (1998). Three-layer architectures. In D. Kortenkamp, R. P. Bonasso & R. Murphy (Eds.), *Artificial intelligence and mobile robots: case studies of successful robot systems* (pp. 195-210). Menlo Park: AAAI Press.

Gollu, A., & Varaiya, P. (1989). *Hybrid dynamical systems.* Paper presented at the 28th IEEE Conference on Decision and Control.

Greenwood, D. T. (1965). *Principles of Dynamics*. Englewood Cliffs, NJ: Prentice-Hall.

Grillner, S. (1975). Locomotion in vertebrates: central mechanisms and reflex interaction. *Physiol Rev, 55*(2), 247-304.

Gruhn, M., Hoffmann, O., Dübbert, M., Scharstein, H., & Büschges, A. (2006). Tethered stick insect walking: A modified slippery surface setup with optomotor stimulation and electrical monitoring of tarsal contact. *Journal of neuroscience methods, 158*(2), 195-206.

Guschlbauer, C., Scharstein, H., & Büschges, A. (2007). The extensor tibiae muscle of the stick insect: biomechanical properties of an insect walking leg muscle. *Journal of Experimental Biology, 210*(6), 1092-1108.

Hess, D., & Büschges, A. (1999). Role of proprioceptive signals from an insect femur-tibia joint in patterning motoneuronal activity of an adjacent leg. *Journal of neurophysiology, 81*, 1856-1865.

Hill, A. V. (1970). *First and last experiments in muscle mechanics*. Cambridge [Eng.]: University Press.

Hofmann, T., & Bässler, U. (1982). Anatomy and physiology of trochanteral campaniform sensilla in the stick insect, *Culiculina impigra. Physiological Entomology, 7*(4), 413-426.

Hooper, S. L., Brezina, V., Cropper, E. C., & Weiss, K. R. (1999). Flexibility of muscle control by modulation of muscle properties. In P. S. Katz (Ed.), *Beyond neurotransmission: neuromodulation and its importance for information processing* (pp. 241-274). New York: Oxford University Press.

Hooper, S. L., Guschlbauer, C., Blümel, M., Rosenbaum, P., Gruhn, M., Akay, T., & Büschges, A. (2009). Neural Control of Unloaded Leg Posture and of Leg Swing in Stick Insect, Cockroach, and Mouse Differs from That in Larger Animals. *Journal of Neuroscience, 29*(13), 4109-4119.

Horchler, A. D., Reeve, R. E., Webb, B., & Quinn, R. D. (2004). Robot phonotaxis in the wild: a biologically inspired approach to outdoor sound localization. *Advanced Robotics, 18*(8), 801-816.

Ijspeert, A. J. (2008). Central pattern generators for locomotion control in animals and robots: a review. *Neural Netw, 21*(4), 642-653.

Ijspeert, A. J., Crespi, A., Ryczko, D., & Cabelguen, J. M. (2007). From swimming to walking with a salamander robot driven by a spinal cord model. *Science, 315*(5817), 1416-1420.

Jindrich, D. L., & Full, R. J. (1999). Many-legged maneuverability: Dynamics of turning in hexapods. *The journal of experimental biology, 202*(12), 1603-1625.

Jindrich, D. L., & Full, R. J. (2002). Dynamic stabilization of rapid hexapedal locomotion. *The journal of experimental biology, 205*(18), 2803-2825.

Josephson, R. K. (1993). Contraction Dynamics and Power Output of Skeletal Muscle. *Annual Review of Physiology, 55*, 527-546.

Kingsley, D. A., Quinn, R. D., & Ritzmann, R. E. (2003). *A cockroach inspired robot with artificial muscles*. Paper presented at the International symposium on adaptive motion of animals and machines (AMAM '03).

Koditschek, D. E., Full, R. J., & Buehler, M. (2004). Mechanical aspects of legged locomotion control. *Arthropod structure & development, 33*(3), 251-272.

Laurent, G., & Richard, D. (1986). The Organization and Role During Locomotion of the Proximal Musculature of the Cricket Foreleg: I. Anatomy and Innervation. *Journal of Experimental Biology, 123*(1), 255-283.

Lewinger, W. A. (2005). *Insect-Inspired, Actively Compliant Robotic Hexapod.* Case Western Reserve University, Cleveland, OH.

Lewinger, W. A., Branicky, M. S., & Quinn, R. D. (2005, Sept. 13-15). *Insect-inspired, actively compliant robotic hexapod.* Paper presented at the International Conference on Climbing and Walking Robots (CLAWAR 05), London, U.K.

Lewinger, W. A., & Quinn, R. D. (2009). *A Small, Autonomous, Agile Robot with an On-board, Neurobiologically-based Control System*. Video Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'09).

Lewinger, W. A., Rutter, B. L., Blümel, M., Büschges, A., & Quinn, R. D. (2006). *Sensory Coupled Action Switching Modules (SCASM) generate robust, adaptive stepping in legged robots*. Paper presented at the 9th International Conference on Climbing and Walking Robots (CLAWAR 06).

Lewinger, W. A., Rutter, B. L., & Quinn, R. D. (2008). *Irregular Terrain Navigation and Leg Coordination Improve Walking Behavior for Small Legged Robots*. Paper presented at the Fourth International Symposium on Adaptive Motion of Animals and Machines from http://amam.case.edu/AMAM%202008%20Abstracts.pdf

Maxwell, B. A., Meeden, L. A., Addo, N. S., Dickson, P., Fairfield, N., Johnson, N., Jones, E. G., Kim, S., Malla, P., Murphy, M., Rutter, B. L., & Silk, E. (2001). REAPER: A Reflexive Architecture for Perceptive Agents. *AI Magazine, 22*(1), 53-66.

Mu, L. (2007). *Interaction between descending input and local thoracic reflexes for joint coordination in cockroach turning.* Case Western Reserve University, Cleveland.

Mu, L., & Ritzmann, R. E. (2005). Kinematics and motor activity during tethered walking and turning in the cockroach, *Blaberus discoidalis*. *J Comp Physiol A Neuroethol Sens Neural Behav Physiol, 191*(11), 1037-1054.

Mu, L., & Ritzmann, R. E. (2008a). Interaction between descending input and thoracic reflexes for joint coordination in cockroach: I. descending influence on thoracic sensory reflexes. *J Comp Physiol A Neuroethol Sens Neural Behav Physiol, 194*(3), 283-298.

Mu, L., & Ritzmann, R. E. (2008b). Interaction between descending input and thoracic reflexes for joint coordination in cockroach. II comparative studies on tethered turning and searching. *J Comp Physiol A Neuroethol Sens Neural Behav Physiol, 194*(3), 299-312.

Mu, L., Taylor, B. K., Rutter, B. L., Ritzmann, R. E., & Quinn, R. D. ((in preparation)). A Kinematic Model Study on the Spatial and Temporal Coordination of the Joints of the Cockroach Mesothoracic Leg.

Nelson, G. M. (2002). *Learning about control of legged locomotion using a hexapod robot with compliant pneumatic actuators.* Case Western Reserve University, Cleveland.

Nelson, G. M., & Quinn, R. D. (1999). Posture Control of a Cockroach-like Robot. *IEEE control systems magazine, 19*(2), 9.

Nelson, G. M., Quinn, R. D., Bachmann, R. J., & Flannigan, W. C. (1997). *Design and Simulation of a Cockroach-Like Hexapod Robot.* Paper presented at the International Conference of Robotics and Automation (ICRA '97), Albuquerque, NM.

Nilsson, N. J. (1980). *Principles of Artificial Intelligence*. Palo Alto: Tioga.

Petri nets. 2009, from http://www.petrinets.info/

Pfeiffer, F., Weidemann, H. J., & Eltze, J. (1994). The TUM Walking Machine. - In: Intelligent Automation and Soft Computing *Trends in Research, Development and Applications* (Vol. 2, pp. 167-174): TSI Press.

Pfeiffer, F., & Zielinska, T. (Eds.). (2003). *Walking: biological and technological aspects*. Udine, Italy: Springer.

Prochazka, A., Clarac, F., Loeb, G. E., Rothwell, J. C., & Wolpaw, J. R. (2000). What do "reflex" and "voluntary" mean? Modern views on an ancient debate. *Experimental brain research, 130*(4), 417-432.

Prochazka, A., & Yakovenko, S. (2002). Locomotor control: from spring-like reactions of muscles to neural prediction. In R. J. Nelson (Ed.), *The somatosensory system: deciphering the brain's own body image* (pp. 141-181). Boca Raton, FL: CRC Press.

Raibert, M., Blankespoor, K., Nelson, G. M., Playter, R., & the Big Dog Team (2008, July 6-11). *BigDog, the Rough-Terrain Quaduped Robot.* Paper presented at the World Congress of the International Federation of Automatic Control, Seoul, Korea.

Raibert, M. H. (1986). *Legged robots that balance*. Cambridge: Massachusetts Institute of Technology.

Reeve, R., Webb, B., Horchler, A., Indiveri, G., & Quinn, R. (2005). New technologies for testing a model of cricket phonotaxis on an outdoor robot. *Robotics and autonomous systems, 51*(1), 41-54.

Ritzmann, R. E., Gorb, S., & Quinn, R. D. (2004). Arthropod locomotion systems: from biological materials and systems to robotics. *Arthropod structure & development, 33*(3), 183-187.

Ritzmann, R. E., & Quinn, R. D. (2003, September 8-12). *Locomotion in Complex Terrain.* Paper presented at the Advanced Class on Walking: biological and technological aspects, Udine, Italy.

Rutter, B. L., Bender, J. A., Ritzmann, R. E., & Quinn, R. D. (2009). Descending commands modify local feedback circuitry to effect behavioral transitions in a robotic neuromechanical cockroach leg model. Program no. 287.3. *2009 Neuroscience Meeting Planner*.

Rutter, B. L., Bender, J. A., Taylor, B. K., Ritzmann, R. E., & Quinn, R. D. (2008). Experiments in locomotion with neuromechanically based robotic insect models. Program no. 198.7. *2008 Neuroscience Meeting Planner*.

Rutter, B. L., Lewinger, W. A., Blümel, M., Büschges, A., & Quinn, R. D. (2007). *Simple Muscle Models Regularize Motion in a Robotic Leg with Neurally-Based Step Generation.* Paper presented at the International Conference on Robotics and Automation (ICRA '07), Rome.

Rutter, B. L., Lewinger, W. A., Taylor, B. K., Wilson, M., Blümel, M., Ekeberg, Ö., Büschges, A., Ritzmann, R. E., & Quinn, R. D. (2006). Neurally-based robot control for neuromechanical modeling of insect stepping. Program No. 449.13. *2006 Neuroscience Meeting Planner*.

Rutter, B. L., Mu, L., Ritzmann, R. E., & Quinn, R. D. (2007). *Transforming Insect Electromyograms into Pneumatic Muscle Control*. Paper presented at the International Conference on Robotics and Automation (ICRA '07), Rome

Rutter, B. L., Taylor, B. K., Mu, L., & Ritzmann, R. E. (2007). *A Functional Kinematic Model of the Cockroach Mesothoracic Leg*. Poster presented at the Eighth International Congress of Neuroethology.

Shirley, D., & Matijevic, J. (1995). Mars Pathfinder Microrover. *Autonomous Robots, 2*(4), 283-289.

Taylor, B. K., Rutter, B. L., & Quinn, R. D. (2009). *A biologically inspired sensory driven method for tracking wind-borne odors*. Paper presented at the Workshop on Performance Metrics for Intelligent Systems (PerMIS '09).

van Heesch, D. (2009). Doxygen: online.

Von Békésy, G. (1960). *Experiments in hearing*. New York,: McGraw-Hill.

Wadden, T., & Ekeberg, O. (1998). A neuro-mechanical model of legged locomotion: single leg control. *Biological cybernetics, 79*(2), 161-175.

Wadden, T., & Ekeberg, O. (1999, Sep). *A neuro-mechanical model of legged locomotion: quadruped control.* Paper presented at the International Conference on Climbing and Walking Robots, Portsmouth.

Waibel, M., Brooks, R. A., Hauert, S., Duerr, P., & Marbach, D. (2007). Rodney Brooks - The Past and Future of Behavior Based Robotics. On *Talking Robots - The Podcast on Robotics and A.I.* [Podcast]: Laboratory of Intelligent Systems, EPFL.

Waibel, M., Ijspeert, A. J., Hauert, S., Duerr, P., & Marbach, D. (2007). Auke Ijspeert - Salamander robot swims and walks. On *Talking Robots - The Podcast on Robotics and A.I.* [Podcast]: Laboratory of Intelligent Systems, EPFL.

Watson, J. T., & Ritzmann, R. E. (1998a). Leg kinematics and muscle activity during treadmill running in the cockroach, *Blaberus discoidalis* : I. Slow running. *Journal of Comparative Physiology a-Neuroethology Sensory Neural and Behavioral Physiology, 182*(1), 11-22.

Watson, J. T., & Ritzmann, R. E. (1998b). Leg kinematics and muscle activity during treadmill running in the cockroach, *Blaberus discoidalis*: II. Fast running. *Journal of Comparative Physiology a-Neuroethology Sensory Neural and Behavioral Physiology, 182*(1), 23-33.

Watson, J. T., Ritzmann, R. E., & Pollack, A. J. (2002). Control of climbing behavior in the cockroach, *Blaberus discoidalis*. II. Motor activities associated with joint movement. *Journal of Comparative Physiology a-Neuroethology Sensory Neural and Behavioral Physiology, 188*(1), 55-69.

Watson, J. T., Ritzmann, R. E., Zill, S. N., & Pollack, A. J. (2002). Control of obstacle climbing in the cockroach, *Blaberus discoidalis*. I. Kinematics. *Journal of Comparative Physiology a-Neuroethology Sensory Neural and Behavioral Physiology, 188*(1), 39-53.

Webb, B. (2001). Can robots make good models of biological behaviour? *Behavioral and Brain Sciences, 24*(06), 1033-1050.

Webb, B., & Scutt, T. (2000). A simple latency-dependent spiking-neuron model of cricket phonotaxis. *Biol Cybern, 82*(3), 247-269.

Wei, T. E., Quinn, R. D., & Ritzmann, R. E. (2004). *A CLAWAR That Benefits From Abstracted Cockroach Locomotion Principles.* Paper presented at the Climbing and walking robots Conference, Madrid, Spain.

Wiener, N. (1948). *Cybernetics; or, Control and communication in the animal and the machine*. Paris; Cambridge, Mass: Hermann; Technology Press.

Yodaiken, V., & Barabanov, M. (1997). *A real-time Linux.* Paper presented at the Linux Applications Development and Deployment Conference (USELINUX), Anaheim, CA.

Zill, S., Schmitz, J., & Büschges, A. (2004). Load sensing and control of posture and locomotion. *Arthropod structure & development, 33*(3), 273-287.