Final Report
TR # GIT-CC-9549

The Detection and Extraction of Interleaved Code Segments

Spencer Rugaber, Kurt Stirewalt, and Linda M. Wills
College of Computing
Georgia Institute of Technology
{spencer, kurt, linda}@cc.gatech.edu

January 17, 1996

NASA Technical Contact: Michael Lowry, NASA Ames Research Center
Research Area: Artificial Intelligence and Software Engineering
Topic Area: Software Understanding and Reengineering
NASA Project Number: NAG 2-890
Georgia Tech Technical Contact:

Spencer Rugaber
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
(404) 894-8450 Fax: (404) 894-9442

Administrative Contact:

Janis L. Goddard
Contracting Officer
Georgia Tech Research Corporation
Centennial Research Bldg

Georgia Institute of Technology
Atlanta, GA 30332-0420
(404) 894-4817 janis.goddard@oca.gatech.edu

1 Introduction: Interleaving

This project is concerned with a specific difficulty that arises when trying to understand and modify computer programs. In particular, it is concerned with the phenomenon of "interleaving" in which one section of a program accomplishes several purposes, and disentangling the code responsible for each purposes is difficult. Unraveling interleaved code involves discovering the purpose of each strand of computation, as well as understanding why the programmer decided to interleave the strands. Increased understanding improve the productivity and quality of software maintenance, enhancement, and documentation activities.

1.1 Goals

It is the goal of the project to characterize the phenomenon of interleaving as a prerequisite for building tools to detect and extract interleaved code fragments.

1.2 Background

Because the project's approach was largely empirical, we tested it on actual software, the `NAIF SPICELIB` software library obtained from the Jet Propulsion Laboratory. The library consists of 170 thousand lines of Fortran code used by scientists to build software systems to analyze data returned from space missions.

We were introduced to this software by scientists at the NASA Ames laboratory who were familiar with it due to their participation in the Amphion project. It is the purpose of Amphion to improve the productivity of the space scientist by automatically generating software for them, starting from a high-level graphical schematic of a space mission. Amphion is able to do this by using a formal model of the domain (solar system kinematics) and then `proving` that the problem defined by the schematic is derivable from the domain model. In the course of the proof, actual Fortran code is generated which makes appropriate calls to the `SPICELIB` library routines.

The success of Amphion is highly dependent on the consistency and completeness of the domain model it uses, and our role was to use the results we learned about interleaving to improve this model.

2 Summary of Results

The interleaving project was in fact able to accomplish its goals. We have constructed a characterization of interleaving and used the characterization to build detection tools. These tools, in turn, were used to analyze `SPICELIB`, pointing out parts of the library that the domain model did not cover. The specific analysis routines that we built and the results they obtained are described in this section and included in the Appendices.

Our analyses fall into two categories: those that focus on detecting interleaving-related data from `SPICELIB` and those that focus on detecting incompletenesses in Amphion's domain model with respect to the library. The results of our empirical analyses on `SPICELIB` are the following.

- **Precondition extraction** – we have been able to extract preconditions on the use of the `SPICELIB` routines. It is often difficult to detect the code responsible for checking these preconditions because it is usually tightly interleaved with the code for the primary computation in order to take advantage of intermediate results computed for the primary computation.

  Using the Software Refinery, our tool detects precondition checks within error handlers and extracts the preconditions into a documentation form suitable for expression as a partial specification. The Refine code which implements this detection and extraction is found in Appendix 7.5. A table of preconditions that were extracted is given in Appendix 6. Our tool generates LaTeXoutput so as to easily produce nicely formatted reports. Our tool generated the LaTeXsource included in Appendix 6 without change.

- **Roots of the call graph** – as part of our analysis of the calling structure of the library (which subroutines call others), we have been able to identify which subroutines are not called by any other subroutines. These are listed in Appendix 4.2. Either these represent high-level routines (such as, `INELPL`) that are intended to be called externally by software that uses `SPICELIB`, or they are "dead" (no longer used). Perhaps they have been replaced by an optimized or more useful subroutine.

- **Constant parameters** – we have identified calls to library subroutines which supply a parameter to the subroutine that is called as opposed to a variable. The constant parameter may be a flag that is being used to choose among a set of possible computations to perform. The significance of constant parameters is that they often signal a form of interleaving, called *control coupling*. The use of control flags allows control conditions to be determined once but used to affect execution at more than one location in the program. We have discovered that the set of routines that are invoked with

a constant parameter accounts for 19 percent of the total routines in `SPICELIB`. They are listed in Appendix 4.3.

- **Indefinite loops** – Most of the analyses for detecting interleaving rely on some form of dataflow analysis. Dataflow analysis propagates data usage information along all possible *sequences* of statements in a subprogram. The accuracy or precision of a dataflow analysis depends upon the ability to cover all such sequences. Unfortunately, programs with loops can sometimes represent *indefinite* sequences of statements. When posed with such programs we must settle for only approximate dataflow information. On the other hand, subprograms whose loops are bounded by constants may be *unrolled* into code each of whose statement sequences may be determined statically. Such subprograms can be analyzed exactly with dataflow techniques.

  Since so many of our interleaving analyses depend upon the precision of dataflow analyses, we wanted some measure of our tool's maximum analysis potential over the library. We chose to measure this as the percentage of routines with no indefinite loops and developed a statistics gathering tool in Refine. The analysis showed that roughly 68 percent of the subprograms had no indefinite loops. This number indicates that we will be able to completely analyze two thirds of the routines. Moreover, in the other cases, approximate dataflow information may be good enough to capture the spirit of the analysis.

- **Multiple Outputs and Tupling** – One heuristic for finding instances of interleaving is to determine which subroutines compute more than one output. When this occurs, the subroutine is returning either the results of multiple distinct computations or a result whose type can not be directly expressed in the Fortran type system (e.g., as a data aggregate or tuple). In the former case, the subroutine is realized as the interleaving of multiple distinct plans. In the latter case, the subroutine may be implementing only a single plan, but a maintainer's conceptual categorization of the subroutine is still obscured by the appearance of some number of seemingly distinct outputs.

  We built a tool which uses Refine rules to analyze the direction of dataflow in parameters of `SPICELIB` functions and subroutines. A parameter's direction is either: **in** if the parameter is only read in the subprogram, **out** if the parameter is only written in the subprogram, or **in-out** if the parameter is both read and written in the subprogram.

  We then analyzed the subroutines in `SPICELIB`, focusing on the in/out directions of parameters: multiple output subprograms will have more than one parameter with direction out or in-out. Our analysis showed that 25 percent of the subprograms in `SPICELIB` had multiple output parameters. We were thus able to focus our work on these routines first, as they are likely to involve interleaving.

In addition to analyses for detecting interleaving in the `SPICELIB` software library, we also performed empirical analyses to determine the coverage provided by the Amphion domain model. In particular, we obtained data on the following.

- **Extent of coverage by routines** – which routines are either directly linked to some entity in the domain model or are invoked by another routine that is covered by the domain model? Based on the calling relationships between library routines and based on Amphion's mapping from domain theory to routines, this analysis revealed that only 35 percent of the library is covered by the domain model.
- **Dead end data flows** – For those routines that are covered by the domain model, which of their outputs are mapped to entities in the domain model? We refer to outputs that are not mapped to anything in the domain model as "dead end dataflows," since the programs that Amphion creates can never make use of these return values; they have not been associated with any meaning in the application domain. Dead end dataflows imply interleaving in the subprogram and/or an incompleteness in the domain model. Our analysis revealed that of the subroutines covered by the domain model, 30 percent have some output parameters that are dead end dataflows.

A paper describing our empirical findings won the best paper award at the International Conference on Software Maintenance this year. We also published our characterization of interleaving at the Working Conference on Reverse Engineering last July. A journal paper on this project has also been submitted to the journal of *Automated Software Engineering* and is currently under review. These papers have been attached to this report.

3 Future Work

There are several possible future directions for pursuing research on interleaving. The suggested directions are summarized here.

3.1 Architectural Interleaving

Thus far, our work has concentrated on detecting instances of interleaving that occur relatively late during design, usually appearing as code optimizations. Significant leverage would obtain if we were able to detect higher level instances of interleaving, specifically situations where several different styles of architecture were interleaved.

There are several promising approaches to this problem. A bottom-up approach tries to detect instances of particular types of modules. For example, the proponents for Structured Design describe module types such as transforming, afferent, efferent, and controlling, all of which can be readily identified due to their dataflow properties. It would be interesting to see the extent to which these types actually occur in their pure form, and, once detected, whether higher level constructs could be detected from them.

Another approach to architectural interleaving detection builds on the concepts of design patterns and frameworks now popular in the object oriented community. It is our belief that characteristic phenomenon of a framework (and possibly of a pattern) is the interaction protocol among its components. We plan to investigate this possibility by instrumenting existing code to produce event traces which can then be analyzed for specific interaction patterns.

The third appealing approach to architectural interleaving investigates the following phenomenon. It turns out that many instances of low level interleaving are really implementations of a more structured design decision, such as aggregation or specialization, made at a higher level in the design. If this is the case, then we might be able to look for explanations of particular patterns of interleaving by detecting the higher level decisions.

3.2 Exploiting Domain Knowledge

Another future direction for research on interleaving exploits domain information. As was the case with `SPICELIB` and Amphion, we expect that our understanding and ability to express domain models and programs will co-evolve. That is, domain models can be constructed or extended by analyzing existing instances of programs in the domain. And as the domain model matures, it will become more and more helpful in understanding programs. One possible application of this will be our ability to express what we learn about a program using a vocabulary that is closer to that of the domain expert than to the programmer.

3.3 Tool Building

The tools that we built for analyzing `SPICELIB` are, for the most part, tailored to it. For example, our detection of preconditions depended on the occurrences of calls to certain error handling routines. It should, however, be possible to build more general tools for detecting and extracting interleavings, based upon the characterization that we have made. Specifically, we need to develop some general tools for doing dataflow analysis in order to refine the approximate analyses that we are currently making. The dataflow tool can then serve as a basis for building tools to detect specific classes of interleavings.

3.4 Further Work with SPICELIB

Finally, there are directions that we could go to further understand `SPICELIB`. In particular, no tools were developed to detect certain instances of interleaving that we noticed, such as reformulation wrappers. A reformulation wrapper is used to transform one problem into another that is simpler to solve and then to transfer the solution back to the original situation. Some examples of reformulation wrappers in `SPICELIB` are: reducing a three-dimensional geometry problem to a two-dimensional one and mapping an ellipsoid to the unit sphere to make it easier to solve three-dimensional intersection problems. Once a reformulation wrapper is recognized, properties of the core ("wrapped") computation that are invariant over the reformulation transformation may be associated with the results transformed back to the original problem domain. This is important in recovering accurate post-conditions of `SPICELIB` subroutines to further elaborate the Amphion domain model. We believe that we could go further in our analysis by looking at some of these cases.

Citation Information for Attached Papers

1. S. Rugaber, K. Stirewalt, and L. Wills. Understanding interleaved code. *Journal of Automated Software Engineering.* Submitted.

2. S. Rugaber, K. Stirewalt, and L. Wills. Detecting interleaving. In *IEEE Conf. on Software Maintenance – 1995*, pages 265–274, Nice, France, September 1995. IEEE Computer Society Press.

3. S. Rugaber, K. Stirewalt, and L. Wills. The interleaving problem in program understanding. In *Proc. of the Second Working Conference on Reverse Engineering*, pages 166–175, Toronto, Ontario, July 1995. IEEE Computer Society Press.

4 Appendix: Empirical Data on NAIF Library

4.1 Preconditions

One of the more ambitious analyses that we perform is the automated detection of subprogram preconditions. Often exception handling behavior can be positively recognized using simple source patterns. We detect precondition checks in code by looking for exception handling code that is invoked under an `IF`-statement whose predicate consists entirely of unmodified subprogram parameters. When these are discovered, we know that part of the subprogram precondition is the logical negation of this predicate. This general procedure will work for any system whose exception handling mechanism is easily detected. Fortunately, `SPICELIB` satisfies this condition. The developers followed a strict discipline of exception propagation by registering an exception upon detection and then exiting the executing subprogram. Since their adherence to this policy was so rigorous, we found that we could detect such instances by checking for an invocation of the exception registry procedure `SIGERR`.

Our precondition extraction code encapsulates the exception recognition code into a single function `looks-like-exception` in the file `exception-pattern.re`. To port the code to a new application, one need merely rewrite this function. Our tool will apply the pattern to any block of code under a conditional whose truth or falsehood depends only upon parameter initial values. As we detect these blocks, we synthesize a partial subprogram precondition as the conjunction of the negation of the predicate of each such conditional. All of this analysis is done by propagating a set of unmodified input variables over the AST of the subprogram looking for exceptional cases. This process uses dataflow propagation techniques from the file `prop-vars.re` and predicate synthesis techniques from the file `precond.re`. (See Appendix 7.5.)

Once we have synthesized the precondition for a given subprogram, we simplify the expression via a predicate optimization pass and then write out the predicate in a LaTeXencoded formula. The accumulated output of all preconditions of all subprograms is then organized into a table via the LaTeXdescription environment. The code that manages this part of the process is in the file `expr-to-latex.re`.

4.2 Roots of the Call Graph

One of our analyses looked at the calling structure of routines in `SPICELIB`. In the course of this analysis, we were able to identify which subroutines are not called by any other subroutines. These are listed in the table on the next page. Either these represent high-level

routines that are intended to be called externally by software that uses SPICELIB, or they are "dead" (no longer used). Perhaps they have been replaced by an optimized or more useful subroutine.

ALLTRU	APPNDC	APPNDD	APPNDI	APPROX	B1900	B1950
BODEUL	CHBDER	CHGIRF	CKBSR	CKGP	CKGPAV	CKGR01
CKGR02	CKGR03	CKLPF	CKNR01	CKNR03	CKUPF	CKW01
CKW02	CKW03	CLLINE	CLPOOL	CONICS	COPYD	CPOSR
CYCLEC	CYLLAT	CYLREC	CYLSPH	DAFA2B	DAFAH	DAFANA
DAFB2A	DAFBT	DAFCAD	DAFCS	DAFFA	DAFFNH	DAFGH
DAFLUH	DAFNRR	DAFRA	DAFRS	DAFRWD	DAFTB	DATANH
DIFFC	DIFFD	DIFFI	DXTRCT	EDLIMB	ELEMD	ELLTOF
ERRACT	ERRDEV	ERRPRT	ET2UTC	EVEN	EXISTS	FETCHC
FETCHD	FETCHI	FILLC	FILLI	FRELUN	FRSTPC	GEOREC
GETMSG	HYPTOF	INELPL	INRYPL	INSRTD	INTERC	INTERD
INTERI	INVERT	INVSTM	IRFDEF	IRFNAM	ISRCHD	J1900
J1950	J2100	JYEAR	KXTRCT	LATCYL	LATSPH	LBUILD
LDPOOL	LPARSS	LSTCLI	LSTLTC	LSTLTI	M2Q	MATCHW
MAXAC	MAXAD	MAXAI	MEQU	MEQUG	MINAC	MINAD
MTXMG	MTXVG	MXMG	MXMTG	MXVG	NCPOS	NCPOSR
NOTRU	NPEDLN	NPLNPT	OPSGNI	ORDC	ORDD	ORDERC
ORDERI	ORDI	OSCELT	PACKAC	PACKAD	PACKAI	PARTOF
PCWID	POLYDS	POOL	POSR	PRODAD	PRODAI	PROMPT
PRTPKG	QCKTRC	QUOTE	RADREC	RDENCC	RDKER	RECCYL
RECGEO	RECRAD	RECSPH	REMOVD	REMSUB	REORDC	REORDI
REPLWD	REPMCT	REPMF	REPMOT	RESET	RESLUN	ROTVEC
SC01	SCE2S	SCLU01	SCS2E	SDIFFC	SDIFFD	SDIFFI
SETD	SETI	SIGDGT	SMSGNI	SOMTRU	SPCA2B	SPCB2A
SPCDC	SPCOPN	SPCRFL	SPCRNL	SPHCYL	SPHLAT	SPHREC
SPHSD	SPKBSR	SPKEZ	SPKLEF	SPKSUB	SPKUEF	SPKW05
SPKW08	SPKW09	SRFREC	SUMAD	SYDIMC	SYDIMI	SYDUPC
SYDUPD	SYDUPI	SYENQC	SYENQI	SYFETI	SYGETI	SYNTHC
SYNTHI	SYORDD	SYORDI	SYPOPC	SYPOPD	SYPOPI	SYPSHD
SYPSHI	SYPUTC	SYPUTD	SYPUTI	SYRENC	SYREND	SYRENI
SYSELC	SYSELD	SYSELI	SYTRNC	SYTRND	SYTRNI	TIPBOD
TISBOD	TPARCH	TRACEG	TRCOFF	TRCPKG	TYEAR	UNIOND
UNIONI	UNORMG	UTC2ET	VALIDD	VALIDI	VMINUG	VPROJG
VREL	VRELG	VSEPG	VTMV	VTMVG	VUPACK	WNCOMD
WNCOND	WNDIFD	WNELMD	WNEXTD	WNFETD	WNFILD	WNFLTD
WNINTD	WNRELD	WNSUMD	WNUNID	WNVALD	WRENCC	WRPOOL

4.3 Constant Parameters

Subprograms with input parameters that are constant at every call site often indicate the presence of *control coupling*. Routines that exhibit control coupling use the facility to interleave distinct plans with similar dataflow requirements. We want to be able to detect when a routine exhibits control coupling for several reasons:

1. the routine will no doubt be easier to understand if viewed as the interleaving of two distinct plans,

2. the precondition for the routine will probably need to be expressed as the conjunction of several disjoint cases. So, for example, if a routine uses control coupling to interleave two plans S_1 and S_2 with individual preconditions P_1 and P_2 respectively, then we would like the precondition of the routine to be of the form:

$$(CC_1 \Rightarrow P_1) \wedge (CC_2 \Rightarrow P_2)$$

   where CC_1 and CC_2 denote the two disjoint control cases.

3. if we are doing inter-procedural dataflow analysis, we might be able to partially evaluate the routine with the control parameter of interest in order to improve the precision of the analysis.

In Fortran, control coupling is typically implemented by using a subprogram formal parameter as a control flag. We can find possible control flags by finding subprograms having formal parameters whose actuals are constants at *every* call-site in the application. Our analysis shows that 19 percent of the routines in `SPICELIB` are of this form. The routines are listed in the table on the next page.

ACCEPT	ASTRIP	BODVAR	BRCKTD	BRCKTI	BSRCHC	BSRCHD
BSRCHI	CHKIN	CHKOUT	CLEARC	CLEARD	CLEARI	CMPRSS
CONVRT	CPOS	CVPOOL	DAFADA	DAFOPN	DAFPS	DAFRDR
DAFRWA	DAFSIH	DAFT2B	DAFUS	DELTET	DPSTR	DPSTRF
DROTAT	ENCHAR	EQSTR	ERRCH	ERRDP	ERRFNM	ERRINT
ESRCHC	EUL2M	EXACT	EXCESS	FILLD	INSLAC	INSLAD
INSLAI	INSRTC	INSSUB	IOERR	IRFNUM	IRFTRN	ISRCHC
ISROT	LATREC	LPARSE	LPARSM	LSTLTD	M2EUL	MOVEC
MOVED	MSGSEL	NEARPT	NVC2PL	ORDERD	OUTMSG	POS
PREFIX	PUTLMS	PUTSMS	RDENCI	REMLAC	REMLAD	REMLAI
REORDD	REPLCH	REPMC	REPMD	REPMI	REPSUB	ROTATE
RQUAD	RTPOOL	SCARDC	SCARDD	SCLD01	SCLI01	SETC
SETERR	SETMSG	SHIFTC	SHIFTL	SHIFTR	SIGERR	SOMFLS
SPCAC	SSIZEC	SSIZED	SSIZEI	SUFFIX	SUMAI	SWAPAC
SWAPAD	SWAPAI	SWPOOL	TWOVEC	UNITIM	VADDG	VALIDC
VEQUG	VHATG	VLCOM	VLCOM3	VLCOMG	VNORMG	VPACK
VSCLG	VSUBG	VZEROG	WRENCI	WRKVAR	WRLINE	XPOSBL
XPOSEG						

4.4 Indefinite Loops

A major source of analysis imprecision comes from loop statements. Dataflow analysis procedures often can not decide the number of iterations a loop will actually perform. This can come about either because there are branching statements within the loop body or because the loop has an indeterminate upper (or lower) bound. Typically, a dataflow analysis procedure will not distinguish data flow coming in the first iteration of the loop from data flowing from a previous iteration. This makes it difficult to precisely analyze the code within the loop. Loops whose upper and lower bounds are constant, however, may be *unrolled* into straight-forward code barring any unstructured control flow within the loop. Our first analysis discovers all such loops in `SPICELIB` and reports the percentage over all loops. We found that 68 percent of the loops in `SPICELIB` had constant upper and lower bounds. This number gives us confidence that we can statically analyze a large part of the library with high precision.

4.5 Tupling/Pure Routines

Some subroutines in `SPICELIB` compute more than one output. When this occurs, the subroutine is returning either the results of multiple distinct computations or a result whose

type can not be directly expressed in the Fortran type system (e.g., as a data aggregate). In the former case, the subroutine is realized as the interleaving of multiple distinct plans. This interleaving complicates the task of understanding the code, clouds a maintainer's conceptual categorization of the subroutine, and often opens the door to dead-end dataflows (see section 5.2) in the domain theory..

If the result is a type that is not directly expressible in the Fortran type system, then the subroutine may be implementing only a single plan. Still a maintainer's conceptual categorization of the subroutine is obscured by the appearance of some number of seemingly *distinct* outputs. This opens up the potential for a domain theory specialist to *miss* relevant data by thinking that he or she was only interested in one of the outputs when in fact both outputs together determine the value of interest. A good example of this case occurs in the `SPICELIB` subroutine `SURFPT`:

```
SUBROUTINE SURFPT (POSITN,U,A,B,C,POINT,FOUND)
```

which conceptually returns the intersection of a vector with the surface of an ellipsoid. However, it is possible to give a vector and an ellipsoid that do not intersect. In such a situation the output parameter `POINT` will be undefined, but the Fortran type system cannot express the type: `DOUBLE PRECISION` $\vee$ *Undefined*. The programmer was forced to simulate a variable of this type using two variables, `POINT` and `FOUND`, adopting the convention that when `FOUND` is **false**, the return value is *Undefined*, and when `FOUND` is **true**, the return value is `POINT`.

Clearly subprograms with multiple outputs complicate program understanding and can lead to subtle bugs in the domain theory–library mapping. We built a tool that determines the multiple output subprograms in a library by analyzing the *direction* of dataflow in parameters of functions and subroutines. A parameter's direction is either: **in** if the parameter is only read in the subprogram, **out** if the parameter is only written in the subprogram, or **in-out** if the parameter is both read and written in the subprogram. Multiple output subprograms will have more than one parameter with direction out or in-out.

The resulting analysis showed that 25 percent of the subprograms had multiple output parameters. Several of these subprograms are explicitly reachable by sentences in the domain theory, namely:

CKGPAV	CYLLAT	CYLSPH	EL2CGV	INEDPL	INELPL	INRYPL
LATCYL	NEARPT	NPEDLN	NPELPT	NPLNPT	PL2NVC	RECCYL
RECLAT	RECRAD	RECSPH	SCE2T	SPHCYL	SPHLAT	SPKEZ
SPKSSB	SURFPT					

Domain theory specialists should probably check that their understanding of every output of these particular routines is correct. We refer domain theory specialists to the section of domain theory dead-end dataflows 5.2 so that they may check those against this list. Certainly as the domain theory is extended, this problem is more likely to occur. We, therefore, have included the complete list of routines with multiple output parameters:

BODEUL	CHBDER	CHBINT	CHGIRF	CKBSR	CKE01	CKE02
CKE03	CKGP	CKGPAV	CKPFS	CKR01	CKR02	CKR03
CKSNS	COPYC	CYLLAT	CYLSPH	DAFAH	DAFARW	DAFFA
DAFHLU	DAFHSF	DAFLUH	DAFNRR	DAFOPN	DAFPS	DAFRA
DAFRCR	DAFRDA	DAFRDR	DAFRFR	DAFRWA	DAFRWD	DAFWDA
DECHAR	DIAGS2	DIFFC	DP2HX	DXTRCT	EL2CGV	ENCHAR
FNDNWD	FRAME	HX2DP	HX2INT	INEDPL	INELPL	INRYPL
INSLAC	INSLAD	INSLAI	INT2HX	INTERC	IRFNAM	IRFNUM
IRFTRN	KXTRCT	LATCYL	LATSPH	LOCLN	LPARSE	LPARSM
M2EUL	MAXAC	MAXAD	MAXAI	MINAC	MINAD	MINAI
NEARPT	NEXTWD	NPARSD	NPARSI	NPEDLN	NPELPT	NPLNPT
NTHWD	PACKAC	PACKAD	PACKAI	PL2NVC	PL2NVP	PL2PSV
POOL	RAXISA	RDKDAT	RDKER	RDKVAR	RDTEXT	RECCYL
RECGEO	RECLAT	RECRAD	RECSPH	REMLAC	REMLAD	REMLAI
REORDC	REORDD	REORDI	RMDUPC	RMDUPD	RMDUPI	RQUAD
SAELGV	SC01	SCARDC	SCE2S	SCE2T	SCENCD	SCET01
SCFM01	SCFMT	SCLI01	SCLU01	SCPART	SCS2E	SCT2E
SCTE01	SCTIKS	SCTK01	SDIFFC	SETC	SPCRFL	SPCRNL
SPHCYL	SPHLAT	SPKAPP	SPKBSR	SPKE08	SPKE09	SPKEZ
SPKGEO	SPKPV	SPKSFS	SPKSSB	SSIZEC	STMP03	SURFPT
SWAPC	SWAPD	SWAPI	SYDELC	SYDELD	SYDELI	SYDUPC
SYDUPD	SYDUPI	SYENQC	SYENQD	SYENQI	SYFETC	SYFETD
SYFETI	SYGETC	SYGETD	SYGETI	SYNTHC	SYNTHD	SYNTHI
SYORDC	SYORDD	SYORDI	SYPOPC	SYPOPD	SYPOPI	SYPSHC
SYPSHD	SYPSHI	SYPUTC	SYPUTD	SYPUTI	SYRENC	SYREND
SYRENI	SYSELC	SYSELD	SYSELI	SYSETC	SYSETD	SYSETI
SYTRNC	SYTRND	SYTRNI	TIPBOD	TISBOD	TPARSE	TRCPKG
UNIONC	UNORM	UNORMG	VALIDC	VPRJPI	VUPACK	WNDIFD
WNFETD	WNSUMD					

5 Appendix: Empirical Data on Amphion Domain Model

Whenever a declarative system is used to generate code over a library, one would like to know whether or not the library is *covered* by the declarative domain. That is:

1. Can every routine in the library be invoked by *some* sentence in the domain theory, and
2. is every value returned by a routine in the library accounted for in the domain theory?

If the answer to either question is no, then either the library is superfluous or the domain theory is incomplete. We would like to know about either case.

5.1 Extent of Coverage by Domain Model

The first question is one of *coverage*. It can be rephrased as, "What percentage of the routines in the library are covered by the domain model?" Our analyses show that only 35 percent of the library is covered by the domain model.

5.2 Dead End Dataflows

Some routines that are covered by the domain model have outputs that are not mapped to anything in the domain model. We call these outputs "dead end dataflows," since the programs that Amphion creates can never make use of these return values; they have not been associated with any meaning in the application domain. Dead end dataflows imply interleaving in the subprogram and/or an incompleteness in the domain model. Our analysis revealed that of the subroutines covered by the domain model, 30 percent have some output parameters that are dead end dataflows. While is probably not surprising considering that many of the subprograms referenced in the domain model have multiple outputs (see section **??**), domain theory specialists should double check that they understand *why* these dead end dataflows exist. That is, is the information returned in these extraneous outputs superfluous? Do they represent values that were merely *convenient* to compute by the library subprogram but that are not pertinent to the computation of interest? If the answer to these questions is no, then the domain theory specialist should consult with a `SPICELIB` specialist to determine the *true* nature of the dataflows.

The dead end dataflows that we found follow:

CKGPAV contains four dead ends: `FOUND CLKOUT AV REF`

PL2NVC contains one dead end: `CONST`

INELPL contains two dead ends: `XPT2 NXPTS`

INEDPL contains one dead end: `FOUND`

NPEDLN contains one dead end: `DIST`

NPELPT contains one dead end: `DIST`

SCE2T contains one dead end: `ET`

NEARPT contains one dead end: `ALT`

INRYPL contains one dead end: `NXPTS`

NPLNPT contains one dead end: `DIST`

SPKSSB contains one dead end: `REF`

SPKEZ contains one dead end: `REF`

RECRAD contains one dead end: `RANGE`

6 Appendix: Preconditions Extracted

What follows is a table of subprogram preconditions indexed by subprogram name. The table was automatically generated from the `SPICELIB` source code using our precondition extraction code.

CARDD $\neg(INT(CELL(-1)) < 0) \wedge \neg(INT(CELL(0)) > INT(CELL(-1))) \wedge \neg(INT(CELL(0)) < 0)$

CARDI $\neg(CELL(-1) < 0) \wedge \neg(CELL(0) > CELL(-1)) \wedge \neg(CELL(0) < 0)$

CKW01 $\neg(SCLKDP(1) < 0.D0) \wedge \neg(ENDTIM < SCLKDP(NREC)) \wedge \neg(BEGTIM > SCLKDP(1)) \wedge \neg(NREC \leq 0)$

CKW02 $\neg(STOP(1) \leq START(1)) \wedge \neg(START(1) < 0.D0) \wedge \neg(ENDTIM < STOP(NREC)) \wedge \neg(BEGTIM > START(1)) \wedge \neg(NREC \leq 0)$

CKW03 $\neg(SCLKDP(1) < 0.D0) \wedge \neg(ENDTIM < SCLKDP(NREC)) \wedge \neg(BEGTIM > SCLKDP(1)) \wedge \neg(NINTS \leq 0) \wedge \neg(NREC \leq 0)$

COUNTC $\neg((BLINE > ELINE) \vee (BLINE \leq 0))$

CYCLAC $(DIR =' F') \vee (DIR =' f')$

CYCLAD $(DIR =' F') \vee (DIR =' F')$

CYCLAI $(DIR =' F') \vee (DIR =' F')$

CYCLEC $(DIR =' R') \vee (DIR =' r')$

DACOSH $\neg(X < 1.D0)$

DAFAH $\neg(FNAME =' ')$

DAFRA $\neg\neg ISORDV(IORDER, N)$

DAFRDA $\neg(BEGIN \leq 0) \wedge \neg(BEGIN > END)$

DAFRWA $\neg((RECNO \leq 0) \vee (WORDNO \leq 0))$

DAFWDA $\neg(BEGIN \leq 0) \wedge \neg(BEGIN > END)$

DATANH $\neg(DABS(X) \geq 1.0D0)$

DROTAT $\neg((IAXIS > 3) \vee (IAXIS < 1))$

EDLIMB $\neg((A \leq 0.D0) \vee (B \leq 0.D0) \vee (C \leq 0.D0))$

ELLTOF $\neg((ECC < 0.D0) \vee (ECC \geq 1.D0))$

ENCHAR $\neg(NUMBER < 0)$

GEOREC $\neg(F \geq 1) \wedge \neg(RE \leq 0.0D0)$

HYPTOF $\neg(ECC < 1.D0)$

INEDPL $\neg((A \leq 0.D0) \vee (B \leq 0.D0) \vee (C \leq 0.D0))$

INSSUB $\neg((LOC < 1) \vee (LOC > (LEN(IN) + 1)))$

ISROT $\neg(NTOL < 0.D0) \wedge \neg(DTOL < 0.D0)$

LGRESP $\neg(STEP = 0.D0) \wedge \neg(N < 1)$

LGRINT $\neg(N < 1)$

M2EUL $\neg((AXIS3 < 1) \vee (AXIS3 > 3) \vee (AXIS2 < 1) \vee (AXIS2 > 3) \vee (AXIS1 < 1) \vee (AXIS1 > 3)) \wedge \neg((AXIS3 = AXIS2) \vee (AXIS1 = AXIS2))$

NPEDLN $\neg((A \leq 0.D0) \vee (B \leq 0.D0) \vee (C \leq 0.D0))$

NPLNPT $\neg VZERO(LINDIR)$

NVP2PL $\neg VZERO(NORMAL)$

PROP2B $\neg(DT < 0) \wedge \neg(DT > 0)$

RDENCC $\neg(N < 1)$

RDENCD $\neg(N < 1)$

RDENCI $\neg(N < 1)$

RECGEO $\neg(F \geq 1) \wedge \neg(RE \leq 0.0D0)$

REMLAC $\neg((LOC < 1) \vee (LOC > NA)) \wedge \neg(NE > (NA - LOC + 1))$

REMLAD $\neg((LOC < 1) \vee (LOC > NA)) \wedge \neg(NE > (NA - LOC + 1))$

REMLAI $\neg((LOC < 1) \vee (LOC > NA)) \wedge \neg(NE > (NA - LOC + 1))$

REMSUB $\neg((LEFT > RIGHT) \vee (RIGHT < 1) \vee (LEFT < 1) \vee (RIGHT > LEN(IN)) \vee (LEFT > LEN(IN)))$

REPSUB $\neg(LEFT < 1) \wedge \neg(RIGHT < (LEFT - 1))$

RQUAD $\neg((A = 0.D0) \wedge (B = 0.D0))$

SC01 $\neg(CLKSTR = ' ')$

SCARDD $\neg((CARD < 0) \vee (CARD > INT(CELL(-1))))$

SCARDI $\neg((CARD < 0) \vee (CARD > CELL(-1)))$

SCE2T $SCTYPE(SC) = 1$

SCENCD $\neg(POS > 1) \wedge \neg(POS = 1)$

SCT2E $SCTYPE(SC) = 1$

SETC $OP = ' '$

SETD $OP = ' '$

SETI $OP = ' '$

SIZED $\neg(INT(CELL(-1)) < 0) \wedge \neg(INT(CELL(0)) > INT(CELL(-1))) \wedge \neg(INT(CELL(0)) < 0)$

SIZEI $\neg(CELL(-1) < 0) \wedge \neg(CELL(0) > CELL(-1)) \wedge \neg(CELL(0) < 0)$

SPHSD $\neg(RADIUS < 0)$

SPKW05 $\neg(FIRST > LAST) \wedge \neg(N \leq 0)$

SPKW08 $\neg(EPOCH1 > FIRST) \wedge \neg((EPOCH1 + (N - 1) * STEP) < LAST) \wedge \neg(STEP \leq 0.D0) \wedge \neg(FIRST \geq LAST) \wedge \neg(N \leq DEGREE)$

SPKW09 $\neg(EPOCHS(1) > FIRST) \wedge \neg(EPOCHS(N) < LAST) \wedge \neg(N \leq DEGREE) \wedge \neg(FIRST \geq LAST)$

SSIZEC $\neg(SIZE < 0)$

SSIZED $\neg(SIZE < 0)$

SSIZEI $\neg(SIZE < 0)$

SURFPT $\neg((U(1) = 0.0D0) \wedge (U(2) = 0.0D0) \wedge (U(3) = 0.0D0))$

SWAPAC $\neg(N < 0) \wedge \neg(LOCN < 1) \wedge \neg(LOCM < 1) \wedge \neg(M < 0)$

SWAPAD $\neg(N < 0) \wedge \neg(LOCN < 1) \wedge \neg(LOCM < 1) \wedge \neg(M < 0)$

SWAPAI $\neg(N < 0) \wedge \neg(LOCN < 1) \wedge \neg(LOCM < 1) \wedge \neg(M < 0)$

SYPUTC $\neg(N < 1)$

SYPUTD $\neg(N < 1)$

SYPUTI $\neg(N < 1)$

TWOVEC $\neg((MAX(INDEXP, INDEXA) > 3) \vee (MIN(INDEXP, INDEXA) < 1)) \wedge \neg(INDEXA = INDEXP)$

TXTOPN $\neg(FNAME = ' ')$

TXTOPR $\neg(FNAME = ' ')$

VALIDC $\neg(N > SIZE)$

VALIDD $\neg(N > SIZE)$

VALIDI $\neg(N > SIZE)$

WNCOMD $\neg(LEFT > RIGHT)$

WNINSD $\neg(LEFT > RIGHT)$

WNRELD $(OP = '>=') \vee (OP = '>')$

WNVALD $\neg ODD(N) \wedge \neg(N > SIZE)$

WRENCC $\neg(N < 1)$

WRENCD $\neg(N < 1)$

WRENCI $\neg(N < 1)$

XPOSBL $\neg((MOD(NCOL, BSIZE) \neq 0) \vee (MOD(NROW, BSIZE) \neq 0)) \wedge \neg(NCOL < 1) \wedge \neg(NROW < 1) \wedge \neg(BSIZE < 1)$

7 Appendix: Refine Code

7.1 The Precondition Extraction Code

The code to perform the precondition extraction follows. Note that the file `exception-pattern.re` is specialized to the `SPICELIB` but is easily ported to other applications.

The file exception-pattern.re

```
!! in-package("RU")
!! in-grammar('user)
#||
  This file encapsulates the application dependent exception handling
  recognition code. This particular instance is specialized to detecting
  exceptions in SPICELIB
||#
var *FOUND-RETURN*      : boolean = undefined
var *FOUND-SIGERR*         : boolean = undefined
rule IS-RETURN ( s : rf::program-unit-statement )
   ~*FOUND-RETURN* &
   rf::return-statement(s)
   -->
   *FOUND-RETURN*
rule IS-SIGERR ( s : rf::program-unit-statement )
   ~*FOUND-SIGERR* &
   rf::call-statement(s) &
   rf::identifier-name(rf::called-object(s)) =
                                 'RFU::SIGERR
   -->
   *FOUND-SIGERR*
"The rules that check for patterns indicating the application is raising
 an exception. We found that the JPL programmers used a very nice scheme
 for handling exceptions, and that a block of code calling the routine
 SIGERR followed by a RETURN statement denotes exception handling."
var *CHECK-EXCEPTION-RULES* : seq(symbol) =
  [ 'IS-RETURN,
    'IS-SIGERR ]
"Given a sequence of Fortran statements (assumed to be the body of an
 IF-THEN-ELSE block), report whether or not it appears to be raising
 an exception."
```

```
function looks-like-exception( stmt-seq : seq(rf::program-unit-statement) ) :
   boolean =
   let (*FOUND-RETURN*      : boolean = false,
        *FOUND-SIGERR* : boolean = false)
      (enumerate a-stmt over stmt-seq do
          preorder-transform(a-stmt, *CHECK-EXCEPTION-RULES*));
      *FOUND-RETURN* & *FOUND-SIGERR*
```

The file prop-vars.re

```
!! in-package("RU")
!! in-grammar('user)
#||
   This file contains code to propagate a list of variables which have not
   been modified through operations which may modify them. The outcome
   of this analysis is a list of variables *still* not modified after
   passing through the given operations.
   Entry Point:
      Propagate-Vars-Through-Expression(expression, set of variables)
      returns set of variables
||#
var *ANALYSIS-SC* : rlt::structure-chart = undefined
function find-in-structure-chart( i : rf::identifier-or-sref ) : rlt::sc-node =
  if(rf::identifier(i))
  then some (sc)(sc in rlt::sc-nodes(*ANALYSIS-SC*) &
                rlt::sc-node-name(sc) = symbol-to-string(
                                                    rf::identifier-name(i)))
  else undefined
function compare-by-position(params   : seq(rf::expression-or-special),
                             idparams : set( rf::expression-or-special ),
                             sc-node  : rlt::sc-node ) : set(symbol) =
  let (out-formals : seq( rlt::sc-formal ) =
                    [ fp | (fp) fp in rlt::sc-node-formals(sc-node) &
                                rlt::sc-formal-direction(fp) ~= 'rlt::in ] )
     let (modifiable-actuals : set(rf::expression-or-special) =
           ac | (ac,fp,i) fp in out-formals &
                          ac in idparams &
                          i in [1..size(rlt::sc-node-formals(sc-node))] &
                          fp = rlt::sc-node-formals(sc-node)(i) &
                          ac = params(i)  )
        image(lambda (a : rf::expression-or-special)
                    let ( a : rf::identifier-or-sref =
                              rf::id-or-indexee(a))
                       if (rf::identifier(a)) then rf::identifier-name(a)
                       else undefined ,
              modifiable-actuals)
"Given a list of parameters to a function or CALL statement, return the
 subset of this list which are simple variable names as opposed to more
```

```
 complex expressions."
function get-identifier-parameters( el : seq( rf::expression-or-special ) ) :
  set(rf::expression-or-special) =
   i | (i : rf::expression-or-special)
         i in el &
         rf::indexable-name(i) &
         ~rf::indexable-name-has-params?(i)
"Given a structure chart node sc-node which represents a called function
 or subroutine, a sequence params of actual parameters to this call, and
 a set vars of immutable variable names, return the subset of vars that
 is still immutable after the execution of the call."
function safe-vars-through-call(sc-node : rlt::sc-node,
                                params  : seq(rf::expression-or-special),
                                vars    : set(symbol) ) :
        set(symbol) =
  let ( ident-params : set( rf::expression-or-special ) =
                       get-identifier-parameters( params ) )
     let ( expr-params : set(rf::expression-or-special) =
                           setdiff(seq-to-set(params),
                                   ident-params))
        let ( newvars : set(symbol) =
                        if defined?(sc-node)
                        then setdiff(vars, compare-by-position(params,
                                                               ident-params,
                                                               sc-node))
                        else vars)
           if empty(expr-params) then
             newvars
          else reduce(intersect,
                      image(lambda (e : rf::expression-or-special)
                              check-expression-or-special(e,
                                                          newvars),
                              expr-params))
#||
  Rationale: Given something that looks like an array ref or a function call,
    there are two cases in which variables can be modified:
       1) The actual parameter is a variable, and the entity is a function
          which modifies it's formal parameter of this position, or
       2) The actual parameter is an expression which itself is a function
          call that modifies parameters.
```

```
  This recursive definition allows us to manage the problem by:
     1) Split the actual parameter list into 2 sets, one of which is
        lone variables, and the other expressions.
     2) Go look up the suspected subprogram in the structure chart. If not
        found, then it was an array ref and we're done with set 1, else
        it was a function call, so we match actuals in set 1 with formals
        in the structure chart node and report cases which can be modified.
     3) Apply a recursive call to check-expression to each member
        of set 2 getting a set of sets of modified vars. Reduce this set
        by append to get a flattened set.
     4) Return the union of the sets output in steps 2 and 3.
||#
function check-indexable-name( fc   : rf::indexable-name,
                               vars : set(symbol) ) :
  set(symbol) =
  if (~rf::indexable-name-has-params?(fc)) then
     vars
  else let( params : seq(rf::expression-or-special) =
                          rf::function-params-or-indices(fc),
            sc-info : rlt::sc-node = find-in-structure-chart(
                                                    rf::id-or-indexee(fc)))
          if defined?(sc-info)
          then safe-vars-through-call(sc-info, params, vars)
          else vars
function check-expression( e    : rf::expression,
                           vars : set(symbol) ) : set(symbol)
  computed-using
    rf::indexable-name(e) => check-expression(e,vars) =
                                  check-indexable-name(e,vars),
    rf::binary-expression(e) => check-expression(e,vars) =
                                reduce(intersect,
                                       image(lambda (x)
                                                check-expression(x,vars),
                                             rf::exp-list(e))),
    rf::unary-expression(e) => check-expression(e,vars) =
                                check-expression(rf::exp-value(e), vars),
    rf::literal-constant(e) or
    rf::open-key-parameter(e) or
    rf::special-intrinsic-call(e) => check-expression(e,vars) = vars
function check-expression-or-special( e    : rf::expression-or-special,
```

```
                                        vars : set(symbol) ) :
  set(symbol)
  computed-using
     rf::expression(e) => check-expression-or-special(e,vars) =
                          check-expression(e,vars),
     rf::label-parameter(e) or rf::null-ex(e) =>
                     check-expression-or-special(e,vars) = vars
"Given an expression e and a set vars of immutable variables, return the
 subset of vars which are still immutable."
function Propagate-Vars-Through-Expression( e    : rf::expression-or-special,
                                            vars : set(symbol) ) :
     set(symbol) =
     check-expression-or-special(e,vars)
function get-vars-from-indexable-name( fc   : rf::indexable-name ) :
  set(symbol) =
  let( sc-info : rlt::sc-node = find-in-structure-chart(rf::id-or-indexee(fc)),
       flat-pars : set(symbol) =
                   (if (rf::indexable-name-has-params?(fc))
                    then let( params : seq(rf::expression-or-special) =
                           rf::function-params-or-indices(fc) )
                         reduce(union, image(get-vars-from-expr, params))
                    else ))
  if(defined?(sc-info))
  then flat-pars
  else flat-pars with rf::identifier-name(rf::id-or-indexee(fc))
function get-vars-from-expr( e    : rf::expression ) : set(symbol)
  computed-using
    rf::indexable-name(e) => get-vars-from-expr(e) =
                                get-vars-from-indexable-name(e),
    rf::binary-expression(e) => get-vars-from-expr(e) =
                                reduce(union,
                                       image(get-vars-from-expr,
                                             rf::exp-list(e))),
    rf::unary-expression(e) => get-vars-from-expr(e) =
                                get-vars-from-expr(rf::exp-value(e)),
    rf::literal-constant(e) or
    rf::open-key-parameter(e) or
    rf::special-intrinsic-call(e) => get-vars-from-expr(e) =
function get-vars-from-expr-or-special( e    : rf::expression-or-special ) :
  set(symbol)
```

```
computed-using
   rf::expression(e) => get-vars-from-expr-or-special(e) =
                        get-vars-from-expr(e),
   rf::label-parameter(e) or rf::null-ex(e) =>
                   get-vars-from-expr-or-special(e) =
```

The file precond.re

```
!! in-package("RU")
!! in-grammar('user)
#||
  Entry Point:
      compute-preconditions( this-sys : rf::system-analysis ) :
        string
  Returns:
      Description of all detectable  preconditions (detected by explicit
        checks whose failure causes an exception). The description is
        in the form of a LaTeX description environment.
||#
type condition-tuple-type = tuple(rf::expression-or-special,symbol)
type condition-tuple-set = set(condition-tuple-type)
"Given a system-analysis, return the cumulative sequence of subroutines
 that are contained in this analysis."
function convert-analysis-into-subr-seq( this-system : rf::system-analysis ) :
  seq(rf::program-unit) =
  analysis::maybe-wrapup-system-analysis(this-system);
  let (exec-progs : seq(rf::executable-program) =
          [ e | (file-an : rf::file-analysis,
                 e       : rf::executable-program)
                 file-an in rf::file-analyses-for-system(this-system) &
                 e = rf::file-analysis-file-ast(file-an) ])
     reduce(append, [ s | ( p : rf::executable-program,
                            s : seq(rf::program-unit) )
                          p in exec-progs &
                          s = rf::program-units(p) &
                          defined?(s) &
                          ~empty(s) ] )
#||
  EXPLICIT DATAFLOW PROPAGATION ROUTINES.
||#
function prop-assign-statement( stmt  : rf::assignment-statement,
                                vars  : set(symbol) ) :
                 set(symbol) =
  let (lhs : rf::indexable-name = rf::assignee(stmt))
     let (simple-lhs : symbol = rf::identifier-name(rf::id-or-indexee(lhs)))
        let (complex-lhs : set(symbol) =
```

```
                        if (rf::indexable-name-has-params?(lhs) and-then
                            defined?(rf::function-params-or-indices(lhs))
                            and-then
                            ~empty(rf::function-params-or-indices(lhs)))
                        then reduce(intersect,
                                    image(lambda (e)
                                       Propagate-Vars-Through-Expression(e,
                                                                         vars),
                                       rf::function-params-or-indices(lhs)))
                        else vars)
          let( rhs : rf::expression-or-special = rf::exp-value(stmt) )
             let (after-exc  : set(symbol) =
                  intersect( Propagate-Vars-Through-Expression(rhs,
                                                               vars),
                             complex-lhs)
                  less simple-lhs )
             after-exc
function prop-call( stmt : rf::call-statement,
                    vars : set(symbol) ) : set(symbol) =
  let (call-pars  : seq(rf::expression-or-special) = rf::call-params(stmt),
       sc-node    : rlt::sc-node = find-in-structure-chart(
                                          rf::called-object(stmt)))
  safe-vars-through-call(sc-node,
                         call-pars,
                         vars)
function run-through-stmt-seq(
                       stmts : seq(rf::program-unit-statement),
                       vars  : set(symbol)) : set(symbol) =
  let (newvars : set(symbol) = vars)
    (enumerate s over stmts do
       newvars <- Propagate-Vars-Through-Executable-Statement(s,newvars));
    newvars
function prop-do( stmt : rf::do-statement,
                  vars : set(symbol) ) : set(symbol) =
  let (do-stms : seq(rf::program-unit-non-enddo-statement) =
                 rf::do-actions(stmt))
     run-through-stmt-seq(do-stms, vars)
function prop-arith-if( stmt : rf::arithmetic-if-statement,
                        vars : set(symbol) ) : set(symbol) =
  Propagate-Vars-Through-Expression( rf::exp-value(stmt), vars )
```

```
function prop-logical-if-statement( stmt : rf::logical-if-statement,
                                    vars : set(symbol) ) :
             set(symbol) =
  Propagate-Vars-Through-Executable-Statement(
           rf::logical-if-stmt(stmt),
           Propagate-Vars-Through-Expression(
                rf::exp-value(stmt),
                vars))
function prop-if-then-else( stmt : rf::if-then-else-statement,
                            vars : set(symbol) ) :
             set(symbol) =
  let (exp-val : rf::expression-or-special = rf::exp-value(stmt))
  let ( exp-vars : set(symbol) = Propagate-Vars-Through-Expression(
                                              exp-val, vars))
     let ( then-vars : set(symbol) =
                       run-through-stmt-seq(rf::then-part(stmt), exp-vars))
        let( else-vars : set(symbol) =
                         if defined?(rf::else-part(stmt)) then
                            run-through-stmt-seq(rf::else-part(stmt), exp-vars)
                         else then-vars )
           let( elsif-vars : set(symbol) =
                             if defined?(rf::elseif-part(stmt)) then
                                prop-if-then-else( rf::elseif-part(stmt),
                                                   exp-vars )
                             else else-vars )
              intersect(then-vars, intersect(else-vars, elsif-vars))
function Propagate-Vars-Through-Executable-Statement(
                                      stmt : rf::executable-statement,
                                      vars : set( symbol ) ) :
                          set( symbol )
  computed-using
  rf::assignment-statement(stmt) =>
                   Propagate-Vars-Through-Executable-Statement(stmt,vars) =
                        prop-assign-statement(stmt,vars),
  rf::call-statement(stmt) =>
                   Propagate-Vars-Through-Executable-Statement(stmt,vars) =
                        prop-call(stmt,vars),
  rf::arithmetic-if-statement(stmt) =>
                   Propagate-Vars-Through-Executable-Statement(stmt,vars) =
                        prop-arith-if(stmt, vars),
```

```
  rf::block-if-statement(stmt) =>
                    Propagate-Vars-Through-Executable-Statement(stmt,vars) =
                         prop-if-then-else(rf::conditional-if(stmt),
                                           vars),
  rf::logical-if-statement(stmt) =>
                    Propagate-Vars-Through-Executable-Statement(stmt,vars) =
                         prop-logical-if-statement(stmt, vars),
  rf::do-statement(stmt) =>
                    Propagate-Vars-Through-Executable-Statement(stmt,vars) =
                         prop-do(stmt,vars),
%%
%% VERY conservative dataflow here. We could do better.
%%
  rf::include-statement(stmt) or
  rf::io-statement(stmt) => Propagate-Vars-Through-Executable-Statement(
                                                          stmt, vars ) = ,
%%
%% These statements can not affect mutation of variables.
%%
  rf::declaration-statement(stmt) or
  rf::entry-statement(stmt) or
  rf::format-statement(stmt) or
  rf::label-definition(stmt) or
  rf::continue-statement(stmt) or
  rf::end-do-statement(stmt) or
  rf::goto-statement(stmt) or
  rf::label-assignment-statement(stmt) or
  rf::pause-statement(stmt) or
  rf::return-statement(stmt) or
  rf::save-statement(stmt) or
  rf::stop-statement(stmt) =>
           Propagate-Vars-Through-Executable-Statement(stmt,vars) = vars
function depends-only-upon-vars( e    : rf::expression-or-special,
                                 vars : set(symbol) ) : boolean =
  let(used-vars : set(symbol) = get-vars-from-expr-or-special(e))
     used-vars subset vars
#||
  PATTERN MATCHING CODE FOLLOWS
||#
function block-if-matches(the-if : rf::if-then-else-statement,
```

```
                          vars   : set(symbol)) :
  condition-tuple-set =
  let (exp-val : rf::expression-or-special = rf::exp-value(the-if))
  if (defined?(the-if))
  then union(
            (if (depends-only-upon-vars(exp-val,vars))
             then (if looks-like-exception(rf::then-part(the-if))
                   then  <exp-val,'NEGATIVE>
                   elseif defined?(rf::else-part(the-if)) &
                          looks-like-exception(rf::else-part(the-if))
                   then  <exp-val,'POSITIVE>
                   else )
            else ),
           (if defined?(rf::elseif-part(the-if))
            then block-if-matches(rf::elseif-part(the-if), vars)
            else  ))
  else
"At the present, logical if's can not match as precondition checks because
 they can not execute both a call to signal and exception and a return."
function logical-if-matches(s    : rf::logical-if-statement,
                            vars : set(symbol)) :
         condition-tuple-set =
function get-vars-from-subroutine( pu : rf::program-unit )  : string =
  let ( vars : set(symbol) =  rf::identifier-name(
                                     rf::id-name(rf::parameter-name(p))) |
                                            (p) p in rf::formals(
                                                   rf::unit-heading(pu)) )
    let (newvars    : set(symbol) = vars,
         candidates : condition-tuple-set = )
       (enumerate s over rf::unit-body(pu) do
           (if rf::logical-if-statement(s)
            then candidates <- union(candidates,
                                     logical-if-matches(s,newvars))
            elseif rf::block-if-statement(s)
                then candidates <- union(candidates,
                                         block-if-matches(
                                            rf::conditional-if(s),newvars)));
          newvars <- Propagate-Vars-Through-Executable-Statement(s,newvars));
       if (empty(candidates))
       then ""
```

```
       else let(seed : condition-tuple-type = arb(candidates))
              let (out-str : string =
                          precond-to-latex( seed.1,
                                            (seed.2 = 'NEGATIVE)))
               (enumerate i over (candidates less seed) do
                  out-str <- concat(out-str,
                                    " \wedge ",
                                    precond-to-latex(
                                              i.1,
                                              (i.2 = 'NEGATIVE))));
               out-str
"Main entry point. Given a system-analysis, computes the preconditions
 and displays them in a LaTeX suitable form."
function compute-preconditions( this-sys : rf::system-analysis ) =
       let( *ANALYSIS-SC* : rlt::structure-chart =
               fret::extract-structure-chart-from-symbol-table(this-sys))
        let (program-units : seq(rf::program-unit) =
                            convert-analysis-into-subr-seq(this-sys))
           format(true, "\begindescription~%");
           (enumerate p over program-units do
              let ( pred-str : string = get-vars-from-subroutine(p))
                 if (~empty(pred-str))
                 then format(true, "\item[");
                      format(true,
                             symbol-to-string(
                                rf::identifier-name(
                                   rf::id-name(rf::unit-heading(p)))));
                      format(true, "] $");
                      format(true, pred-str);
                      format(true, "$~%"));
           format(true, "\enddescription~%")
"A specialization of compute-preconditions to the NAIF library"
function naif-preconditions () =
    let(system-name : string =
           "/users/projects/groups/nasa-jpl/naif/analysis/naif-ast.analysis")
     let (this-sys : rf::system-analysis = analysis::load-system-analysis(
                                                              system-name))
       analysis::maybe-wrapup-system-analysis(this-sys);
       compute-preconditions(this-sys)
```

The file expr-to-latex.re

```
!! in-package("RU")
!! in-grammar('user)
#||
  This file contains code for outputting expressions in the Fortran language
  model in a form suitable for processing by LaTeX.
||#
var *dummy-negate-for-test* : rf::negate-expression =
                              make-object('rf::negate-expression)
function literal-to-string( e : rf::expression ) : string =
  let(newstr  : string = [],
      the-lit : string =
                if (rf::literal-true(e))
                then " \bf true "
                elseif (rf::literal-false(e))
                then " \bf false "
                else format(false," ~\pp\ ", e))
%%
%% Go replicate all occurrences of ~ in strings.
%%
       (enumerate i over [1..length(the-lit)] do
          (if(the-lit(i) = #\~)
          then newstr <- append(newstr,#\~));
          newstr <- append(newstr, the-lit(i)));
       newstr
function id-to-string( id : rf::identifier ) : string =
  symbol-to-string(rf::identifier-name(id))
function expr-to-string( e : rf::expression ) : string
  computed-using
  rf::add-expression(e) => expr-to-string(e) = " + ",
  rf::and-expression(e) => expr-to-string(e) = " \wedge ",
  rf::concatenate-expression(e) => expr-to-string(e) = " . ",
  rf::divide-expression(e) => expr-to-string(e) = " / ",
  rf::eq-expression(e) => expr-to-string(e) = " = ",
  rf::eqv-expression(e) => expr-to-string(e) = " = ",
  rf::exponentiate-expression(e) => expr-to-string(e) = "^",
  rf::ge-expression(e) => expr-to-string(e) = " \geq ",
  rf::gt-expression(e) => expr-to-string(e) = " > ",
  rf::le-expression(e) => expr-to-string(e) = " \leq ",
```

```
  rf::lt-expression(e) => expr-to-string(e) = " < ",
  rf::multiply-expression(e) => expr-to-string(e) = " \ast ",
  rf::ne-expression(e) => expr-to-string(e) = " \neq ",
  rf::neqv-expression(e) => expr-to-string(e) = " \neq ",
  rf::or-expression(e) => expr-to-string(e) = " \vee ",
  rf::string-subrange-extraction(e) => expr-to-string(e) = " .. ",
  rf::subtract-expression(e) => expr-to-string(e) = " - ",
  rf::literal-constant(e) => expr-to-string(e) = literal-to-string(e),
  rf::open-key-parameter(e) => expr-to-string(e) = " ???? ",
  rf::special-intrinsic-call(e) => expr-to-string(e) = " ?? ",
  rf::identity-expression(e) => expr-to-string(e) = "",
  rf::negate-expression(e) => expr-to-string(e) = " - ",
  rf::not-expression(e) => expr-to-string(e) = " \neg "
function power-expr( e : rf::expression ) : boolean =
  rf::exponentiate-expression(e)
function multiplicative-expr( e : rf::expression ) : boolean =
  rf::divide-expression(e) or
  rf::concatenate-expression(e) or
  rf::string-subrange-extraction(e) or
  rf::multiply-expression(e)
function additive-expr( e : rf::expression ) : boolean =
  rf::add-expression(e) or
  rf::subtract-expression(e)
function atomic-expr( e : rf::expression ) : boolean =
  rf::identity-expression(e) or
  rf::literal-constant(e)
function eqv-expr(e : rf::expression ) : boolean =
  rf::eqv-expression(e) or
  rf::neqv-expression(e)
function eq-expr(e : rf::expression ) : boolean =
  rf::eq-expression(e) or
  rf::ne-expression(e) or
  rf::ge-expression(e) or
  rf::gt-expression(e) or
  rf::le-expression(e) or
  rf::lt-expression(e)
function and-or-expr( e : rf::expression ) : boolean =
  rf::and-expression(e) or
  rf::or-expression(e)
function not-expr( e : rf::expression ) : boolean =
```

```
  rf::negate-expression(e) or
  rf::not-expression(e)
function unknown-expr( e : rf::expression ) : boolean =
  rf::open-key-parameter(e) or
  rf::special-intrinsic-call(e)
#||
  Implements a total ordering over the types of expressions.
||#
function precedes-eq(e1 : rf::expression,
                     e2 : rf::expression ) : boolean =
  if ( atomic-expr(e1) )
  then not-expr(e2) or
       and-or-expr(e2) or
       eqv-expr(e2) or
       eq-expr(e2) or
       power-expr(e2) or
       multiplicative-expr(e2) or
       additive-expr(e2) or
       atomic-expr(e2)
  elseif ( additive-expr(e1) )
  then not-expr(e2) or
       and-or-expr(e2) or
       eqv-expr(e2) or
       eq-expr(e2) or
       power-expr(e2) or
       multiplicative-expr(e2) or
       additive-expr(e2)
  elseif ( multiplicative-expr(e1) )
  then not-expr(e2) or
       and-or-expr(e2) or
       eqv-expr(e2) or
       eq-expr(e2) or
       power-expr(e2) or
       multiplicative-expr(e2)
  elseif ( power-expr(e1) )
  then not-expr(e2) or
       and-or-expr(e2) or
       eqv-expr(e2) or
       eq-expr(e2) or
       power-expr(e2)
```

```
  elseif ( eq-expr(e1) )
  then not-expr(e2) or
       and-or-expr(e2) or
       eqv-expr(e2) or
       eq-expr(e2)
  elseif ( eqv-expr(e1) )
  then not-expr(e2) or
       and-or-expr(e2) or
       eqv-expr(e2)
  elseif ( and-or-expr(e1) )
  then not-expr(e2) or
       and-or-expr(e2)
  elseif ( not-expr(e1) )
  then not-expr(e2)
  else undefined
function need-parens?( e1 : rf::expression,
                       e2 : rf::expression ) : boolean =
  ~atomic-expr(e2) &
  ~rf::indexable-name(e2) &
  precedes-eq(e2,e1) &
  ~precedes-eq(e1,e2)
function construct-bin-expr-list( e : rf::expression,
                                  es : seq(rf::expression)) : string =
  if(empty(es))
  then undefined     %% This should never happen
  else let(head-xpr  : rf::expression = first(es),
           rest-xpr  : seq(rf::expression) = rest(es))
        let (head-p-str : string = expr-to-latex(head-xpr))
         let(head-str : string = if (need-parens?(e,head-xpr))
                                  then concat( "(", head-p-str, ")" )
                                 else head-p-str)
            if (~empty(rest-xpr))
            then let(rest-str : string = construct-bin-expr-list(e,rest-xpr))
                   (if power-expr(e)
                    then concat(head-str, expr-to-string(e), "",rest-str,"")
                    else concat(head-str, expr-to-string(e), rest-str))
            else head-str
function construct-binary-expr(e : rf::expression) : string =
  construct-bin-expr-list(e, rf::exp-list(e))
function construct-unary-expr(e : rf::expression) : string =
```

```
  if atomic-expr(e)
  then expr-to-string(e)
  else let( val   : rf::expression = rf::exp-value(e),
            x-str : string = expr-to-string(e))
          let ( val-str : string = expr-to-latex(val) )
             if (need-parens?(e,val))
             then concat(x-str,"(",val-str,")")
             else concat(x-str,val-str)
function construct-indexable-name(e : rf::expression) : string =
  let ( ret-str : string = id-to-string(rf::id-or-indexee(e)))
     (if (rf::indexable-name-has-params?(e))
      then let (params : seq(rf::expression-or-special) =
                         rf::function-params-or-indices(e))
             let (len : integer = length(params))
                if(len > 0) then
                  (ret-str <- concat(ret-str,"(",expr-to-latex(first(params)));
                   (enumerate i over [2..len] do
                       ret-str <- concat(ret-str, ",",
                                         expr-to-latex(params(i))));
                   ret-str <- concat(ret-str, ")")));
      ret-str
#||
  This function is grouped by binding precedence of the operators
  (to determine if parenthesis insertion is necessary).
||#
function expr-to-latex( e : rf::expression ) : string
  computed-using
  rf::indexable-name(e) => expr-to-latex(e) = construct-indexable-name(e),
  rf::binary-expression(e) => expr-to-latex(e) = construct-binary-expr(e),
  rf::literal-constant(e) => expr-to-latex(e) = literal-to-string(e),
  rf::open-key-parameter(e) => expr-to-latex(e) = "",
  rf::special-intrinsic-call(e) => expr-to-latex(e) = "",
  rf::unary-expression(e) => expr-to-latex(e) = construct-unary-expr(e)
function precond-to-latex( e       : rf::expression,
                           negate? : boolean ) : string =
  let (xpr-str : string = expr-to-latex(e))
     if (negate?)
     then (if (need-parens?(*dummy-negate-for-test*,e))
           then concat("\neg (", xpr-str, ")")
           else concat("\neg ", xpr-str))
```

```
    else xpr-str
```

7.2 Definite Loops

Definite loops are easily detected using Refine rule construct and a tree walking traversal.

The file def-loops.re

```
!! in-package("RU")
!! in-grammar('user)
var *number-of-do-loops* : integer = 0
var *number-of-definite-loops* : integer = 0
var *visited* : set(object) =
rule TALLY-STANDARD-DO-LOOP( a : object )
  ~(a in *visited*) &
  rf::do-statement(a) &
  rf::literal-constant(rf::base-value(a)) &
  rf::literal-constant(rf::exp-value(a))
    -->
    (*number-of-definite-loops* = *number-of-definite-loops* + 1) &
    (*number-of-do-loops* = *number-of-do-loops* + 1) &
    a in *visited*
rule COUNT-DO-LOOP( a : object )
  ~(a in *visited*) &
  rf::do-statement(a) &
  ~( rf::literal-constant(rf::base-value(a)) &
     rf::literal-constant(rf::exp-value(a)))
  -->
  (*number-of-do-loops* = *number-of-do-loops* + 1) &
  a in *visited*
var *LOOP-COUNT-RULES* : seq(symbol) =
  ['TALLY-STANDARD-DO-LOOP,
   'COUNT-DO-LOOP]
"Given a system-analysis, return the cumulative sequence of subroutines
 that are contained in this analysis."
function convert-analysis-into-subr-seq( this-system : rf::system-analysis ) :
  seq(rf::program-unit) =
  analysis::maybe-wrapup-system-analysis(this-system);
  let (exec-progs : seq(rf::executable-program) =
```

```
          [ e | (file-an : rf::file-analysis,
                 e       : rf::executable-program)
                 file-an in rf::file-analyses-for-system(this-system) &
                 e = rf::file-analysis-file-ast(file-an) ])
    reduce(append, [ s | ( p : rf::executable-program,
                           s : seq(rf::program-unit) )
                         p in exec-progs &
                         s = rf::program-units(p) &
                         defined?(s) &
                         ~empty(s) ] )
"Given a system-analysis, return a tuple containing the number of subroutines
 in the system and the number of subroutines which contain either:
  1) no loops, or
  2) only loops with literal upper and lower bounds."
function compute-definite-loops( this-system : rf::system-analysis ) :
  tuple(integer,integer) =
  let (program-units : seq(rf::program-unit) =
                  convert-analysis-into-subr-seq(this-system),
       number-of-subrs      : integer = 0,
       number-of-unrollables : integer = 0)
  (enumerate ourguy over program-units do
   (let (*number-of-do-loops* : integer = 0,
         *number-of-definite-loops* : integer = 0,
         *visited* : set(object) = )
      preorder-transform(ourguy, *LOOP-COUNT-RULES*);
      number-of-subrs <- number-of-subrs + 1;
      *number-of-do-loops* = *number-of-definite-loops*
      -->
      number-of-unrollables = number-of-unrollables + 1));
   <number-of-unrollables, number-of-subrs>
function count-naif-loops() : tuple(integer,integer) =
  let(system-name : string =
           "/users/projects/groups/nasa-jpl/naif/analysis/naif-ast.analysis")
     let (this-sys : rf::system-analysis = analysis::load-system-analysis(
                                                              system-name))
        let( dl-tup : tuple(integer,integer) =
                       compute-definite-loops(this-sys))
        format(true,
               "~% Found ~\pp\ subroutines of which ~\pp\",
               dl-tup.2,
```

```
            dl-tup.1);
        format(true,
            " had no or only definite loops~% for a ratio of ~\pp\~%",
            (integer-to-real(dl-tup.1) / integer-to-real(dl-tup.2)))
```

7.3 Detecting Constant Parameters

The structure chart object provided by Refine/Workbench contains information about subprograms and formal parameters. It also contains link to invocations of a particular procedure. To detect subprograms with constant parameters, simply enumerate over all subprograms and examine the sequence of call sites to see if a particular actual parameter is constant in every case.

The file const-pars.re

```
!! in-package("RU")
!! in-grammar('user)
#||
  $Id: const-pars.re,v 1.1 1995/12/19 19:05:03 kurt Exp $
  This code searches for routines in SPICELIB that are invoked somewhere
  else in the library with a "constant" parameter. Constant here means
  either a literal or a Fortran PARAMETER. The motivating hypothesis here
  is that invocations of routines with non-varying parameters may
  indicat control coupling.
  $Log: const-pars.re,v $
# Revision 1.1  1995/12/19  19:05:03  kurt
# Initial revision
#
||#
%%
%% We want to return true this call parameter is a constant
%% (literal or Fortran PARAMETER). To recognize this in the structure
%% chart is actually somewhat difficult. There are 2 cases:
%%    1) If a parameter is a literal, then its corresponding
%%       rf::call-argument must be undefined [indicating that it's
%%       not a variable] and an  empty set of rf::call-uses [indicating
%%       that it is not an expression].
%%
%%    2) If a parameter is a Fortran PARAMETER, then its corresponding
```

```
%%       rf::call-argument must have a "mode" rf::unit-mode of
%%       'FRET::CONSTANT [which I found out indicates that it's
%%       a PARAMETER].
%%
%%    3) If a parameter is an expression whose leaves are one of the
%%       above. rf::call-argument-descriptors represent only the
%%       leaves of the expressions in the set rf::call-uses.
%%
%%
function refs-only-constants?( r : rf::call-argument-descriptor ) : boolean =
  let (se : rf::symbol-entry = rf::call-argument(r))
    if ( se = undefined )
    then
       let (constant-refs : boolean = true)
          (enumerate i over rf::call-uses(r) do
             constant-refs <- (constant-refs & refs-only-constants?(i)));
          constant-refs
    else (rf::unit-mode(se) = 'FRET::CONSTANT)
function constant-params?( s : rf::call-descriptor ) : boolean =
  let (cargs : seq(rf::call-argument-descriptor) = rf::call-arguments(s))
     reduce(or,  refs-only-constants?(x) | (x) x in cargs )
"Returns a sequence of booleans, one for each invocation of the given node,
 that represent whether or not the node was invoked with constant parameters.
 The interpretation of this sequence is that its or reduction indicates
 that the node is called somewhere with constant parameters, and its and
 reduction indicates that it is called everywhere with constant parameters."
function calls-with-const-params( node : rlt::sc-node ) : seq(boolean) =
  let ( in-edges : set(rlt::sc-edge) = rlt::sc-node-in-edges(node) )
     [ constant-params?(y) | (x : rlt::sc-edge,
                              y : rf::program-unit-statement)
                             (x in in-edges) &&
                                  (y in rlt::sc-edge-call-points(x)) ]
"Gathers the set of routines that are invoked with constant parameters"
function gather-routines-called-constant-params (
        this-sys : rf::system-analysis ) :
                tuple(set(rlt::sc-node), set(rlt::sc-node)) =
  let (chart : rlt::structure-chart =
              fret::extract-structure-chart-from-symbol-table(this-sys))
    let (scnds           : set(rlt::sc-node) =
                  rlt::sc-nodes(chart),
```

```
        some-set : set(rlt::sc-node) = ,
        all-set : set(rlt::sc-node) =  )
      (enumerate i over scnds do
          if(rlt::sc-node-user-defined?(i)) then
             let(calls-with-consts : seq(boolean) =
                                     calls-with-const-params(i))
                (if(reduce(and, calls-with-consts)) then
                    (all-set <- all-set with i;
                     some-set <- some-set with i)
                 elseif(reduce(or, calls-with-consts)) then
                    some-set <- some-set with i));
     <all-set, some-set>
"The constant parameter to subprogram analysis specialized to the
 NAIF library"
function naif-const-param-analysis() =
  let (sys-name : string =
       "/users/projects/groups/nasa-jpl/naif/analysis/naif-ast.analysis")
    let (this-sys : rf::system-analysis =
                    analysis::load-system-analysis(sys-name))
      analysis::maybe-wrapup-system-analysis(this-sys);
      let ( nodes : tuple(set(rlt::sc-node),set(rlt::sc-node)) =
            gather-routines-called-constant-params(this-sys) )
        format(true,"~% SUBPROGRAMS ALWAYS INVOKED WITH CONSTANT PARAMS~%");
        (enumerate i over nodes.1 do
            format(true, "~% ~\pp\~%", rlt::sc-node-name(i)));
        format(true,"~% SUBPROGRAMS INVOKED WITH CONSTANT PARAMS~%");
        (enumerate i over nodes.2 do
            format(true, "~% ~\pp\~%", rlt::sc-node-name(i)))
```

7.4 Multiple Outputs

The Software Refinery provides a package to create structure chart (call graph) objects. The nodes of these structure charts are annotated with direction information about parameters. We were thus able to detect subprograms with multiple outputs using only the structure chart object. This greatly simplified the implementation. The file `tuple.re` contains this analysis. Note that this detection returns the subprograms with multiple outputs that are invoked according to the domain theory.

```
!! in-package("RU")
!! in-grammar('user)
"This is the set of routines that we identified as directly reachable from
 the domain theory."
var *calls* : set(string) =
                                "BODMAT",
                                "BODVAR",
                                "CGV2EL",
                                "CKGPAV",
                                "CYLLAT",
                                "CYLREC",
                                "CYLSPH",
                                "EDLIMB",
                                "EL2CGV",
                                "EL2PL",
                                "ELPROJ",
                                "FINDPV",
                                "I2SC",
                                "INEDPL",
                                "INELPL",
                                "INRYPL",
                                "LATCYL",
                                "LATREC",
                                "MTXV",
                                "MXV",
                                "NEARPT",
                                "NPEDLN",
                                "NPELPT",
                                "NPLNPT",
                                "NVP2PL",
                                "PJELPL",
```

```
                                "PL2NVC",
                                "PSV2PL",
                                "RADREC",
                                "RECCYL",
                                "RECD",
                                "RECLAT",
                                "RECRAD",
                                "RECSPH",
                                "SCE2T",
                                "SENT",
                                "SPHCYL",
                                "SPHLAT",
                                "SPHREC",
                                "SPKEZ",
                                "SPKSSB",
                                "SRFREC",
                                "SURFNM",
                                "SURFPT",
                                "TIKTOL",
                                "VADD",
                                "VDIST",
                                "VEQU",
                                "VHAT",
                                "VMINUS",
                                "VPRJP",
                                "VSEP",
                                "VSUB"
function detect-mult-outs( cg : rlt::structure-chart ) : set(rlt::sc-node) =
   s | (s : rlt::sc-node)
         s in rlt::sc-nodes(cg) &
         ex(x : rlt::sc-formal,
            y : rlt::sc-formal)
           ( x in rlt::sc-node-formals(s) &
             y in rlt::sc-node-formals(s) &
             x ~= y &
             rlt::sc-formal-direction(x) in 'rlt::out,
                                            'rlt::in-out &
             rlt::sc-formal-direction(y) in 'rlt::out,
                                            'rlt::in-out)
function compute-mouts( this-sys : rf::system-analysis ) : set(string) =
```

```
#||
  let (this-sys : rf::system-analysis = analysis::load-system-analysis(
        "/users/projects/groups/nasa-jpl/naif/analysis/naif-ast.analysis"))
||#
    analysis::maybe-wrapup-system-analysis(this-sys);
    let (cg : rlt::structure-chart =
              fret::extract-structure-chart-from-symbol-table(this-sys))
       let(multiple-outs : set(string) = image(rlt::sc-node-name,
                                               detect-mult-outs(cg)))
          intersect(multiple-outs, *calls*)
```

7.5 Amphion Domain Theory Coverage

Whenever a declarative system is used to generate code over a library, one would like to know whether or not the library is *covered* by the declarative domain. That is, can every routine in the library be invoked by *some* sentence in the domain theory, and is everything returned by a routine in the library accounted for in the domain theory? If the answer to either question is no, then either the library is superfluous or the domain theory is incomplete. We would like to know about either case.

Subprogram Reachability Detection

```
!! in-package("RU")
!! in-grammar('user)
%%
%% Want to conver the call graph (structure chart) into a relation so
%% that we can compute its transitive closure. Also, we want to remove
%% any edges to non user defined nodes.
%%
function call-graph-to-relation( cg : rlt::structure-chart ) :
  set(tuple(rlt::sc-node, rlt::sc-node)) =
  let(this-set : set(tuple(rlt::sc-node, rlt::sc-node)) = ,
      edges    : set(rlt::sc-edge) = rlt::sc-edges(cg))
     (e in edges &
      rlt::sc-node-user-defined?(rlt::sc-edge-to(e))
      -->
      <rlt::sc-edge-from(e), rlt::sc-edge-to(e)> in this-set);
     this-set
function to-string( s : set(tuple(rlt::sc-node, rlt::sc-node) ) ) :
  set(tuple(string, string)) =
   <rlt::sc-node-name(z.1), rlt::sc-node-name(z.2)> |
     (z : tuple(rlt::sc-node, rlt::sc-node)) z in s
"This is the set of routines that we identified as directly reachable from
 the domain theory."
var *calls* : set(string) =
                              "BODMAT",
                              "BODVAR",
                              "CGV2EL",
                              "CKGPAV",
                              "CYLLAT",
                              "CYLREC",
```

```
"CYLSPH",
"EDLIMB",
"EL2CGV",
"EL2PL",
"ELPROJ",
"FINDPV",
"I2SC",
"INEDPL",
"INELPL",
"INRYPL",
"LATCYL",
"LATREC",
"MTXV",
"MXV",
"NEARPT",
"NPEDLN",
"NPELPT",
"NPLNPT",
"NVP2PL",
"PJELPL",
"PL2NVC",
"PSV2PL",
"RADREC",
"RECCYL",
"RECD",
"RECLAT",
"RECRAD",
"RECSPH",
"SCE2T",
"SENT",
"SPHCYL",
"SPHLAT",
"SPHREC",
"SPKEZ",
"SPKSSB",
"SRFREC",
"SURFNM",
"SURFPT",
"TIKTOL",
"VADD",
```

```
                                "VDIST",
                                "VEQU",
                                "VHAT",
                                "VMINUS",
                                "VPRJP",
                                "VSEP",
                                "VSUB"
function get-results() : set(string) =
  let (this-sys : rf::system-analysis = analysis::load-system-analysis(
        "/users/projects/groups/nasa-jpl/naif/analysis/naif-ast.analysis"))
       analysis::maybe-wrapup-system-analysis(this-sys);
       let (cg : rlt::structure-chart =
         fret::extract-structure-chart-from-symbol-table(this-sys))
         let ( closed-set : set(tuple(rlt::sc-node, rlt::sc-node)) =
              tclosure(call-graph-to-relation(cg)))
            let (closed-str-set : set(tuple(string, string)) =
                 to-string(closed-set))
            (enumerate i over image(closed-str-set, *calls*) do
              format(true,"~\pp\~%", i))
```

Dead End Dataflow Detection

```
!! in-package("RU")
!! in-grammar('user)
#||
  Given a set *COVERED-ROUTINES* of names of covered routines in the library,
  and given a set *COVERED-DATAFLOWS* of routine/parameter tuples, compute
  the dead-end dataflows in *COVERED-ROUTINES* with respect to
  *COVERED-DATAFLOWS*.
||#
var *empty-cp-desc* : covered-parameter-desc = <"", >
#||
  Should load this input from inputs.re (automatically generated by snark
                                         analysis).
var *COVERED-DATAFLOWS* : set(covered-parameter-desc) =
   <"LATCYL", "LONGC" >
||#
"Given a set of names of routines in the library and the call graph,
 return the set covered-parameter-descriptors that summarize these entities"
function expand-routine-names-from-cg( routine-names : set(string),
```

```
                                        cg            : rlt::structure-chart ) :
  set(covered-parameter-desc) =
  image( lambda (v : rlt::sc-node)
               < rlt::sc-node-name(v),
                  rlt::sc-formal-name(f) | (f : rlt::sc-formal)
                                           f in rlt::sc-node-formals(v) &
                                           rlt::sc-formal-direction(f) in
                                              'rlt::out, 'rlt::in-out  >,
          n | (n : rlt::sc-node) n in rlt::sc-nodes(cg) &
                                    rlt::sc-node-name(n) in routine-names )
"Display the pairs of routines with dataflows not covered."
function display-dead-end-dataflows(dead-ends : set(covered-parameter-desc)) =
  enumerate de over dead-ends do
     if( ~empty(de.out-params) ) then
        (format(true,
                "~\pp\ ",
                de.routine-name);
         (enumerate p over de.out-params do
             format(true,
                    "~\pp\ ", p));
         format(true, "~%"))
"Of the routines that do not seem to cover any outputs, go see if there
 are any outputs to cover."
function
compute-dead-end-dataflows-from-totals(covset : set(covered-parameter-desc))
  : set(covered-parameter-desc) =
   p | (p : covered-parameter-desc) p in covset & ~empty(p.out-params)
"Of the routines that cover some outputs, detect those outputs that are
 not covered."
function
compute-dead-end-dataflows-from-partials(covset : set(covered-parameter-desc))
  : set(covered-parameter-desc) =
  let( retset        : set(covered-parameter-desc) =  )
  %%
  %% Author's note. You might think a transform (-->) would be more
  %% appropriate here, and you would be right. Unfortunately, the
  %% transform equivalent of this unearthed a bug in Refine 4.0-Beta.
  %%
  fa(p : covered-parameter-desc)
     (p in covset =>
```

```
      fa(q : covered-parameter-desc)
         (q in *COVERED-DATAFLOWS* =>
            (p.routine-name = q.routine-name) =>
               retset <- retset with <p.routine-name,
                                      setdiff(p.out-params,q.out-params)>));
          retset
function analyze-dead-end-dataflows() =
  let (this-sys : rf::system-analysis = analysis::load-system-analysis(
        "/users/projects/groups/nasa-jpl/naif/analysis/naif-ast.analysis"))
    analysis::maybe-wrapup-system-analysis(this-sys);
    let (cg : rlt::structure-chart =
              fret::extract-structure-chart-from-symbol-table(this-sys),
         routine-names : set(string) =
                         image( lambda (x : covered-parameter-desc)
                                      x.routine-name,
                                *COVERED-DATAFLOWS* ))
      let( partial-routines : set(covered-parameter-desc) =
                              expand-routine-names-from-cg(routine-names, cg),
           total-routines   : set(covered-parameter-desc) =
                              expand-routine-names-from-cg(
                                  setdiff(*COVERED-ROUTINES*,routine-names),
                                  cg) )
         let(partials : set(covered-parameter-desc) =
                        compute-dead-end-dataflows-from-partials(
                                                      partial-routines),
             totals   : set(covered-parameter-desc) =
                        compute-dead-end-dataflows-from-totals(
                                                      total-routines))
             display-dead-end-dataflows(union(partials, totals))
```