SPF Framework

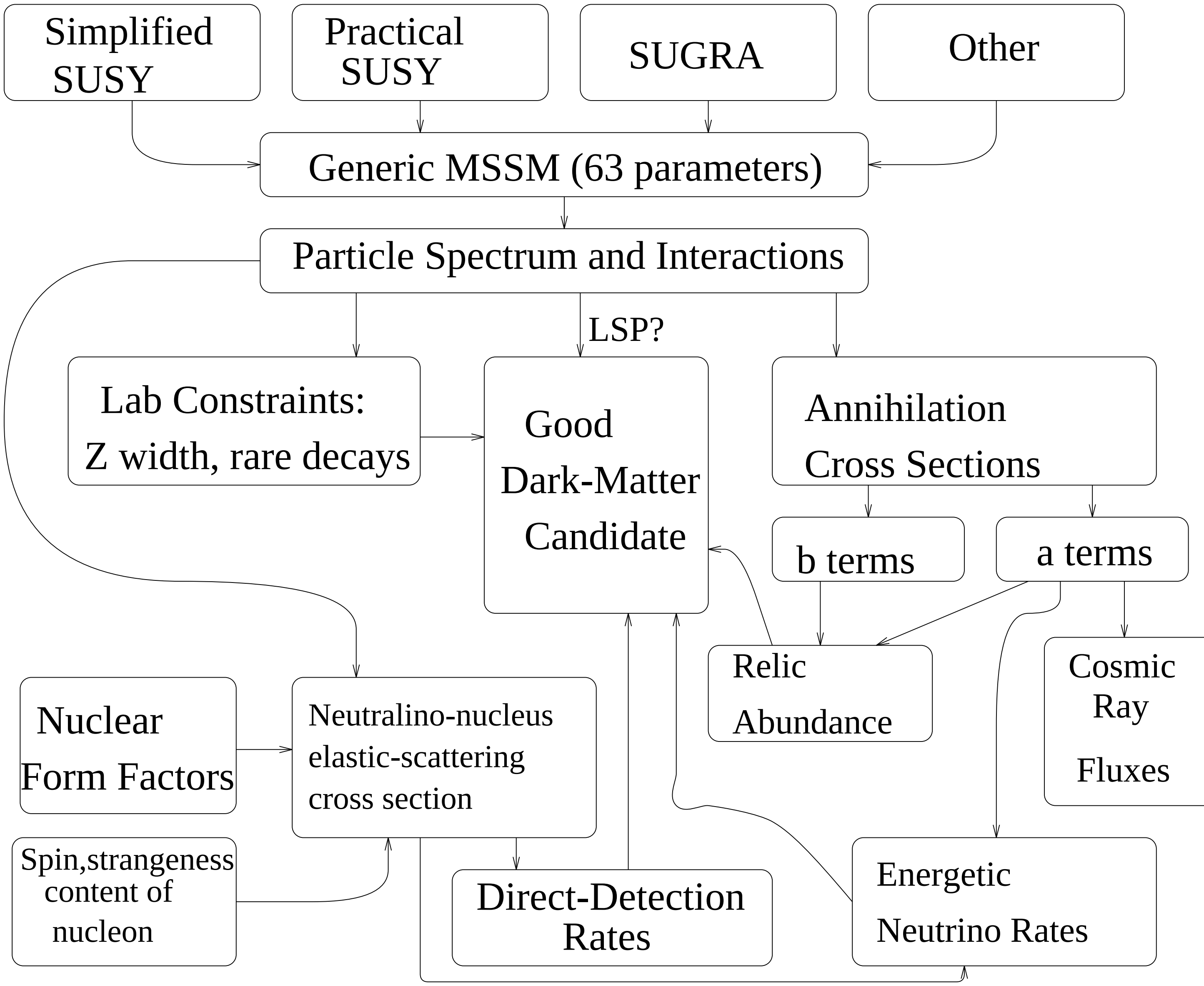
Gerard Jungman (Los Alamos)

http://t8web.lanl.gov/people/jungman

What is SPF?

- cosmology/phenomenology calculations for the MSSM
- replacement for NEUTDRIVER
  - same physics
  - new design
- testbed for more general frameworks

What does it do?



Goals for SPF Framework

- Reduce and/or eliminate monolithic nature of legacy code.

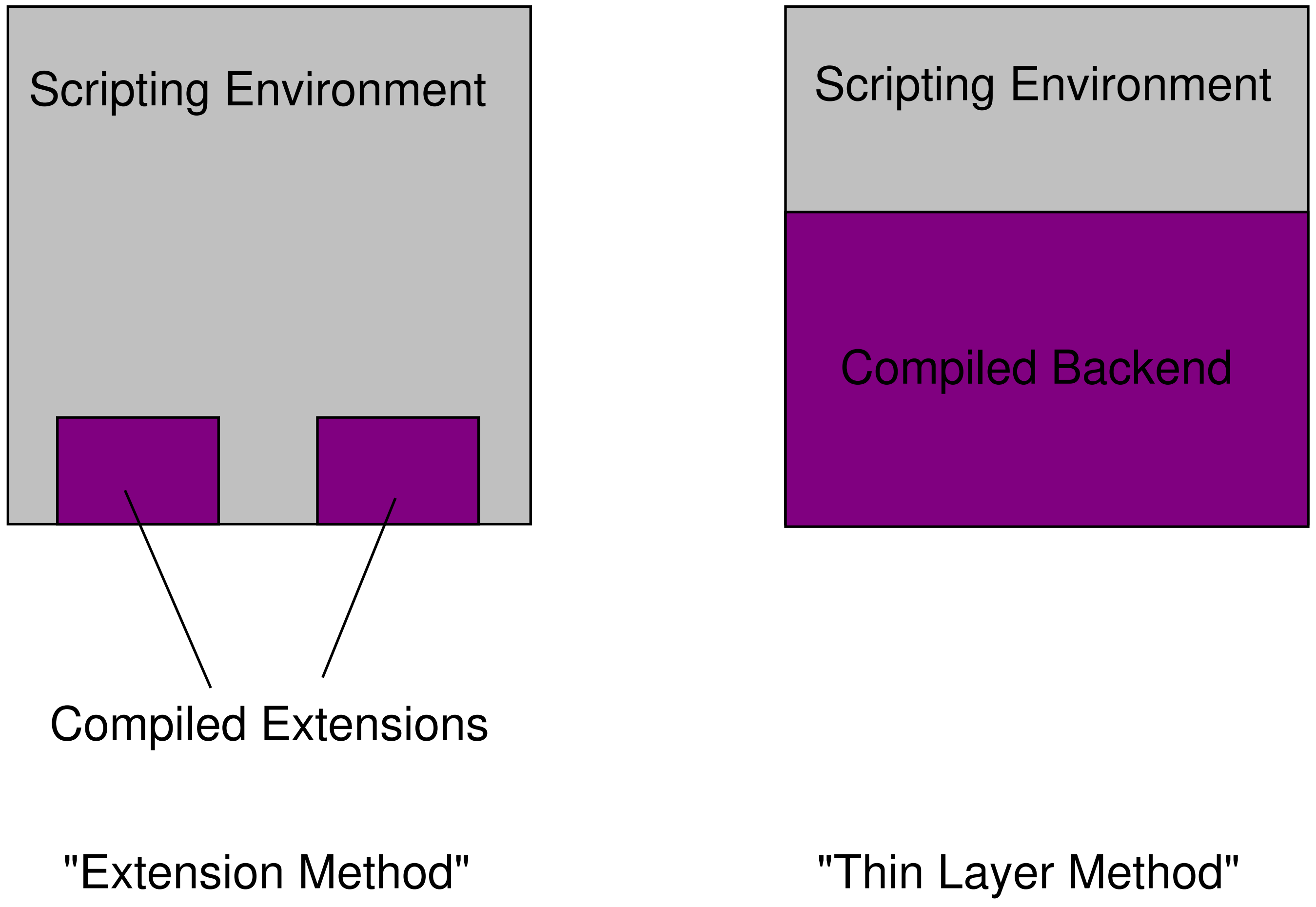
- Eliminate constraints on the end-user.

- Extensibility: an obvious buzzword, but here we can even see what needs to be done.

- Create as much language independence as possible.

... Scripting

Scripting Methodology Spectrum



What to do about scripting?

- Extensions
  - Write directly in the scripting language.
  - Rapid Development
  - No Static-Typing Headaches
  - How to achieve language independence?
- Thin Layer
  - Automatically generate scripting layer.
  - Extend easily to different languages.
  - C++ compiled backend is usable on its own, and is well understood technology.
  - Cannot avoid standard C++ design headaches.

Ideal Approach

- Start with scripting language.
- Rapid prototyping.
- Introduce extensions as needed.
- Keep the stable design that emerges.
- Throw away the prototype. [the hard part!]
- Start again with thin layer methodology, having accomplished the hardest part of the job (reaching a stable and workable design).

Time-Constrained False Economy Approach

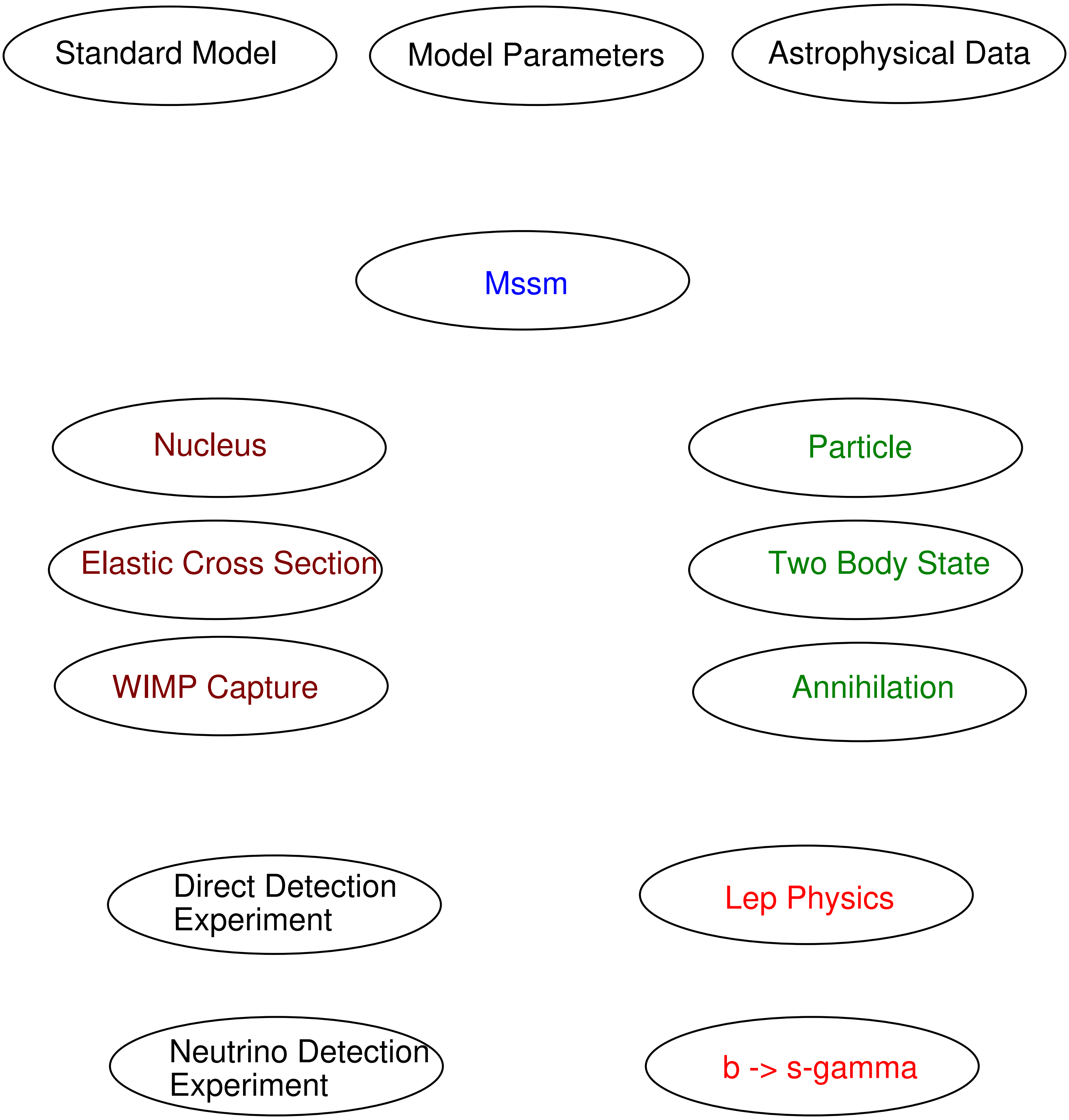
- Worry alot.
- Decide to build the C++ library first (thin layer immediately) since you know you can deliver *something* that way.
- Go through two or three redesigns in C++.
- Realize that you probably should have chosen the "Ideal Approach" since you are effectively trying to do rapid protoyping anyway, and in the hostile C++ environment.
- Worry some more.
- Get something that works, with one week to spare, and declare yourself done.
- Worry about the fact that it could have turned out better.

The SPF C++ Library

- Identification of Abstractions

- Dreaded Double-Dispatch...

SPF Abstractions



Annihilation Cross-Section

$\sigma(IS(\chi\chi) \to FS(ZZ))$

```
cross_section(const IS *, const FS *);
IS * initial_state = new StateNeutNeut(0,0);
FS * final_state   = new StateVV(ZBOSON,ZBOSON);
```

- Abstract arguments.
- Double-dispatch... or worse?
- Cross section can vary too.

```
Operation::doOp(const IS *, const FS *) = 0;
Operation * op;
op->doOp(initial_state, final_state);
```

- Triple dispatch. ACK! Where will it end?

Multiple Dispatch

- Standard Methods
  - RTTI... sparingly
  - home-grown (table lookups)... too painful
- Chosen Solution
  - Parameterize on first type: reduces to (run-time) double dispatch.
  - Natural asymmetry between initial and final states makes this reasonable.

```
template <class IS>
class Annihilation {
  virtual void do_something(const FS *) = 0;
}
```

How It Looks: Two Body States

```
class TwoBodyStateObject {
  virtual void accept(TwoBodyStateTask &) = 0;
};

template <class Particle1, class Particle2>
class TwoBodyState : public TwoBodyStateObject {
  void accept(TwoBodyStateTask & t)
  {
    t.doTask(*this)
  };
}

typedef TwoBodyState<Fermion,Fermion> StateFF;
// etc

class TwoBodyStateTask {
  virtual void doTask(const StateFF &) = 0;
  virtual void doTask(const StateVV &) = 0;
  // etc
}
```

How It Looks: Annihilation Processes

```
template <class IS>
class ProcessTask : public TwoBodyStateTask {
}

template <class IS>
class CSProcessTask : public ProcessTask<IS> {
  void doTask(const StateFF & s)
  {
    /* do something here */
  };
  void doTask(const StateVV & s)
  {
    /* do something here */
  };
  // etc
}
```

Relic Abundance

Particle Spectrum

Particle	Mass	Multiplicity
P0	m0	g0
P1	m1	g1
...	...	...

Reaction Network

$$\begin{array}{ccc} \sigma_{00} & \sigma_{01} & \cdots \\ \sigma_{01} & \sigma_{11} & \cdots \\ \cdots & \cdots & \cdots \end{array}$$

- Handle general particle spectra.
- Handle general reaction networks.
- Transparent model independence.

Relic Abundance Object

```
class RelicAbundance {
  HANDLE registerParticle(Particle &, ...) = 0;
  void   registerProcess(HANDLE, HANDLE,
                               Annihilation &) = 0;
}

Neutralino p0(0);
Neutralino p1(1);
TwoBodyState<Fermion,Fermion> tt(QUARK_T,QUARK_T);
Annihilation<Neutralino,Neutralino> cs00(p0,p0);
Annihilation<Neutralino,Neutralino> cs01(p0,p1);
cs00.addFinalState(tt);
cs01.addFinalState(tt);
RelicAbundance * relic;
handle0 = relic->registerParticle(p0, m0, g0);
handle1 = relic->registerParticle(p1, m1, g1);
relic->registerProcess(handle0, handle0, cs00);
relic->registerProcess(handle0, handle1, cs01);
// ...
result = relic.lightParticleAbundance();
```

Something Better?

Would it be better to parameterize on all the types?

```
template <class IS, class FS>
class Annihilation {
  virtual void accept(Task<IS,FS> &) = 0;
}

template <class IS, class FS>
class ATask {
  void doTask(void);
}

class IS_A : public IS { ... };
class IS_B : public IS { ... };
  ...
ATask<IS_A,FS_A>::doTask() { do something };
ATask<IS_B,FS_B>::doTask() { something else };
```

Can you solve multiple-dispatch this way in general?

The Scripting Layer

- SWIG
- choose Python as initial target language
- Technicalities
  - static objects: who creates them?
  - cannot export templates
  - cannot export overloaded functions

SWIG

- Simplified Wrapper and Interface Generator [www.swig.org]
  - automatic generation of scripting layer from interface specification
  - Python, Perl, TCL

module.i $\Rightarrow$module.c $\Rightarrow$module.so

Future Needs and Wants

- exploration / "mining"
  - adaptive scans
  - computer-aided "proofs" of exhaustion
- visualization
  - guide exploration
  - manage model data, input and output
- not programs but *programming interfaces*

Schedule

- Short Term (July ...?)
  - stabilize C++ library
  - stabilize scripting
- Longer Term
  - physics enhancements
  - interaction with other tools