Results from the Virtual Monte Carlo upgrade

B. Volkel

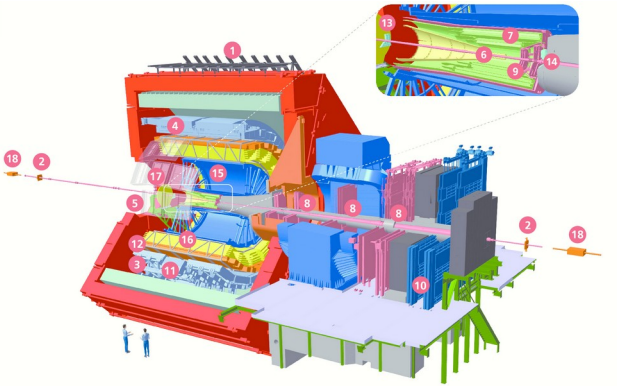
HighRR BiWeekly, 24.03.2021 - @remote

Overview

- Briefly: The principle of Monte Carlo detector simulations
- Introduction to the Virtual Monte Carlo (VMC) package
- Challenges for ALICE in LHC Run3 – limitations of the VMC package and desired extensions
- Examples
- Sketching the code extension
- Another overall design decision
- Conclusion

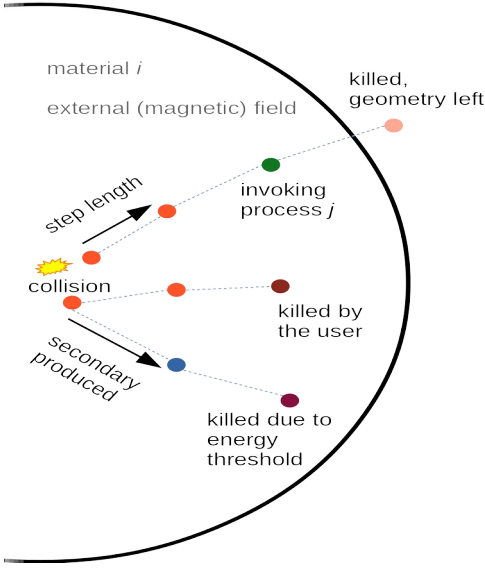
The principle of Monte Carlo detector simulations

- A Monte Carlo detector simulation transports particles through the detector geometry (e.g. GEANT)
- Involves different possible physics processes for given particles
  [Compton, Bremsstrahlung, nucleon-nucleon, decay and many more]



The principle of Monte Carlo detector simulations

- A Monte Carlo detector simulation transports particles through the detector geometry (e.g. GEANT)
- Involves different possible physics processes for given particles
  [Compton, Bremsstrahlung, nucleon-nucleon, decay and many more]
- Each particle is transported in single **steps**
  1) Compute mean free path per process
  2) Derive no-interaction probability
  3) Compute step length



mean free path in material i for process j

$$\lambda_j(E) = \left(\sum_i n_i \cdot \sigma_j(Z_i, E) \right)^{-1}$$

density

cross section

no interaction probability along Δx

$$P(\Delta x) = \exp\left[- \int_{x_0}^{x_1} \frac{dx}{\lambda(x)} \right]$$

roll dice (MC) and invoke process with smallest step $s(x)$

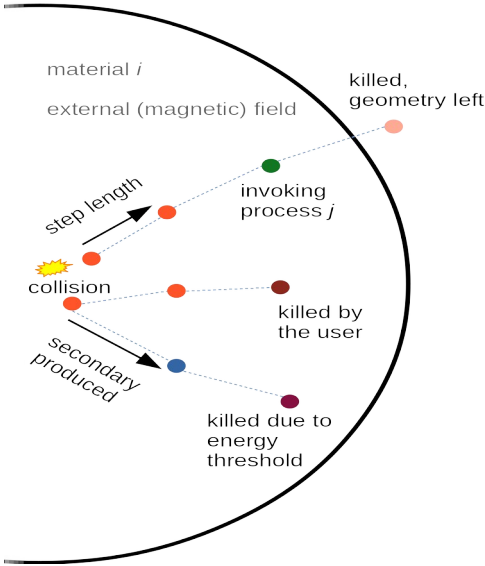
$$-\ln P(\Delta x) = \int_{x_0}^{x_1} \frac{dx}{\lambda(x)} = n_\lambda$$

$$\xrightarrow{evaluate} s(x) = n_\lambda \cdot \lambda(x)$$

number of mean free paths

The principle of Monte Carlo detector simulations

- A Monte Carlo detector simulation transports particles through the detector geometry (e.g. GEANT)
- Involves different possible physics processes for given particles
  [Compton, Bremsstrahlung, nucleon-nucleon, decay and many more]
- Each particle is transported in single **steps**
  1) Compute mean free path per process
  2) Derive no-interaction probability
  3) Compute step length



mean free path in material i for process j

$$\lambda_j(E) = \left(\sum_i n_i \cdot \sigma_j(Z_i, E) \right)^{-1}$$

density

cross section

no interaction probability along Δx

$$P(\Delta x) = \exp\left[- \int_{x_0}^{x_1} \frac{dx}{\lambda(x)} \right]$$

roll dice (MC) and invoke process with smallest step $s(x)$

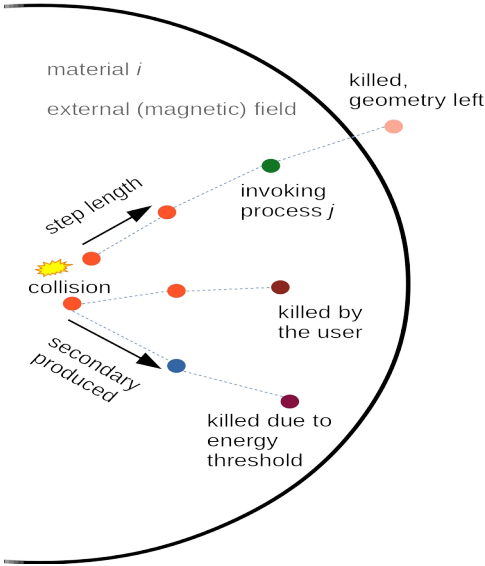
$$-\ln P(\Delta x) = \int_{x_0}^{x_1} \frac{dx}{\lambda(x)} = n_\lambda$$

$$\xrightarrow{evaluate} s(x) = n_\lambda \cdot \lambda(x)$$

number of mean free paths

Brief overview of Monte Carlo detector simulations

- A Monte Carlo detector simulation transports particles through the detector geometry (e.g. GEANT)
- Involves different possible physics processes for given particles
  [Compton, Bremsstrahlung, nucleon-nucleon, decay and many more]
- Each particle is transported in single **steps**
  1) **Compute mean free path per process**
  2) Derive no-interaction probability
  3) Compute step length



mean free path in material i for process j

$$\lambda_j(E) = \left(\sum_i n_i \cdot \sigma_j(Z_i, E) \right)^{-1}$$

density

cross section

no interaction probability along Δx

$$P(\Delta x) = \exp\left[- \int_{x_0}^{x_1} \frac{dx}{\lambda(x)} \right]$$

roll dice (MC) and invoke process with smallest step $s(x)$

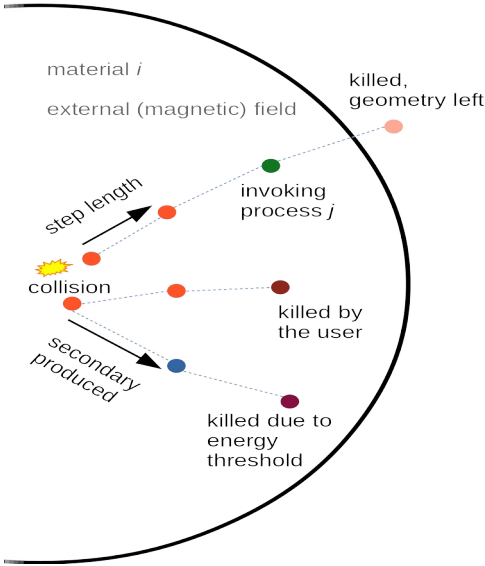
$$-\ln P(\Delta x) = \int_{x_0}^{x_1} \frac{dx}{\lambda(x)} = n_\lambda$$

$$\xrightarrow{evaluate} s(x) = n_\lambda \cdot \lambda(x)$$

number of mean free paths

Brief overview of Monte Carlo detector simulations

- A Monte Carlo detector simulation transports particles through the detector geometry (e.g. GEANT)
- Involves different possible physics processes for given particles
  [Compton, Bremsstrahlung, nucleon-nucleon, decay and many more]
- Each particle is transported in single **steps**
  1) Compute mean free path per process
  2) **Derive no-interaction probability**
  3) Compute step length



mean free path in material i for process j

$$\lambda_j(E) = \left(\sum_i n_i \cdot \sigma_j(Z_i, E) \right)^{-1}$$

density

cross section

no interaction probability along Δx

$$P(\Delta x) = \exp\left[- \int_{x_0}^{x_1} \frac{dx}{\lambda(x)} \right]$$

roll dice (MC) and invoke process with smallest step $s(x)$

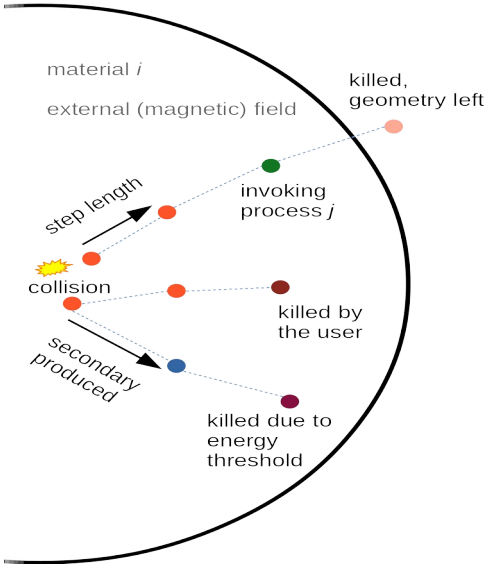
$$-\ln P(\Delta x) = \int_{x_0}^{x_1} \frac{dx}{\lambda(x)} = n_\lambda$$

$$\xrightarrow{evaluate} s(x) = n_\lambda \cdot \lambda(x)$$

number of mean free paths

Brief overview of Monte Carlo detector simulations

- A Monte Carlo detector simulation transports particles through the detector geometry (e.g. GEANT)
- Involves different possible physics processes for given particles
  [Compton, Bremsstrahlung, nucleon-nucleon, decay and many more]
- Each particle is transported in single **steps**
  1) Compute mean free path per process
  2) Derive no-interaction probability
  3) Compute step length



mean free path in material i for process j

$$\lambda_j(E) = \left(\sum_i n_i \cdot \sigma_j(Z_i, E) \right)^{-1}$$

density

cross section

no interaction probability along Δx

$$P(\Delta x) = \exp\left[- \int_{x_0}^{x_1} \frac{dx}{\lambda(x)} \right]$$

roll dice (MC) and invoke process with smallest step $s(x)$

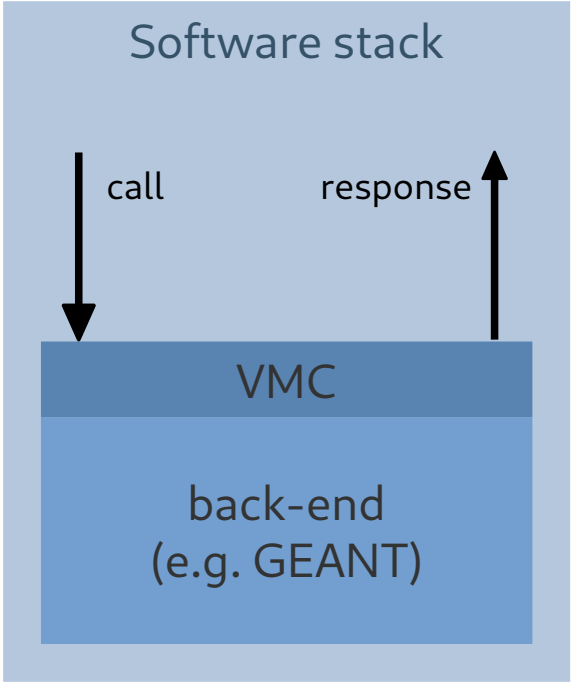
$$-\ln P(\Delta x) = \int_{x_0}^{x_1} \frac{dx}{\lambda(x)} = n_\lambda$$

$$\xRightarrow{evaluate} s(x) = n_\lambda \cdot \lambda(x)$$

number of mean free paths

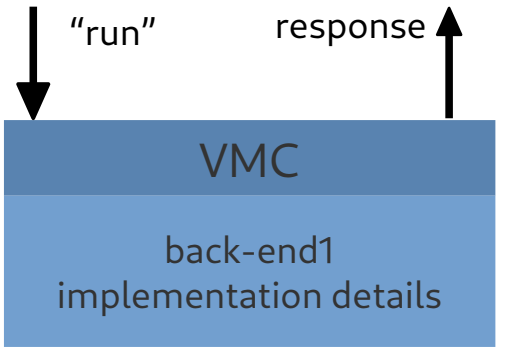
Now what is the Virtual Monte Carlo (VMC) package?

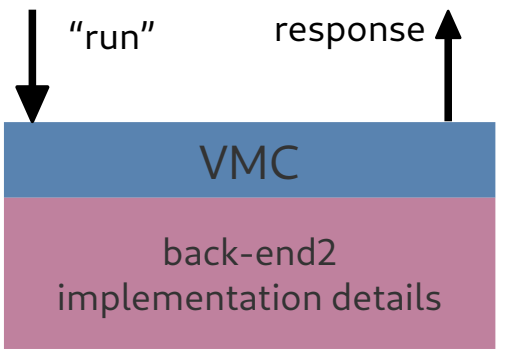
- It is a framework/interface layer
  As such, it is (re-)**usable** within different software stacks and provides common interfaces to use its functionality.
- It is an **abstraction layer** "hiding" the details of **different back-ends** which are in this case different **Monte Carlo detector simulation/transport engines**
- **Unify and make usage of different back easier**
- Interfaces for GEANT4, GEANT3 and Fluka transport engines
  [GEANT4 is the state-of-the art and successor of GEANT3, Fluka more specialised for nuclei simulation]
- An interface can also be provided for any user-specific simulation engine to be accessed via common VMC interfaces



What is the purpose of using such a framework?

- The only things you need to know are the abstract interfaces, “how to talk to it”, say ”run”, “stop”, “pause”, “useGeometry”, “changePosition”, “howManySecondaries”
- A good framework keeps the interfaces intact, back-end details might change
  → no need to worry about that as a user
- Demands on developers
  - Make sure interfaces trigger expected behaviour
  - Hide implementation details as much as possible
  - Ensure backward compatibility when new features are implemented
    [that might be broken sometimes e.g. for major updates or depends on the user community]
- A framework such as the VMC package hence ensures the reliable and unified usage of detector simulation engines in a big collaboration.





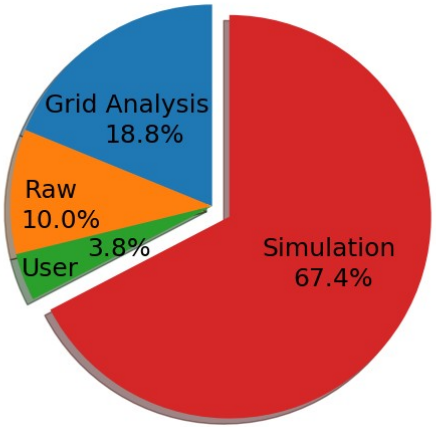
Why different detector simulation frameworks?

- Different simulation frameworks are used by different (CERN) collaborations, but why?
  - Historical reasons (once a big software stack/framework is in place, it is used and further developed
  - Specific needs: frameworks can be more or less tightly bound to a specific simulation engine
    → ATLAS for instance relies heavily on GEANT4 by default
    → ALICE was using GEANT3 during LHC Run2, GEANT4 will most likely become the default in the future, but possible usage of GEANT3 and Fluka maintained (more on that later)
- VMC is also used by the Fair collaboration, indeed the ALICE O^2 software stack interfaces with VMC in a very similar way → VMC is not only an ALICE effort

Challenges for ALICE in LHC Run 3 (similar for other experiments)

- Up to **~100 times more data** expected compared to Run 2
  - Not possible to increase simulation production by comparable factor
    [during Run 2 ~2/3 of the resource were dedicated to simulation]
  - Need smarter, **software-based solutions** to make the detector simulation more efficient
- Need for what is generically called fast simulation and **hence interfacing**
- **GEANT will still be considered to be the default**, BUT identify most demanding
  - …sub-detectors,
  - …phase-space regions,
  - …particle types

…and introduce more efficient simulation process if possible
[such as parametrisation, Machine-Learning based approaches]

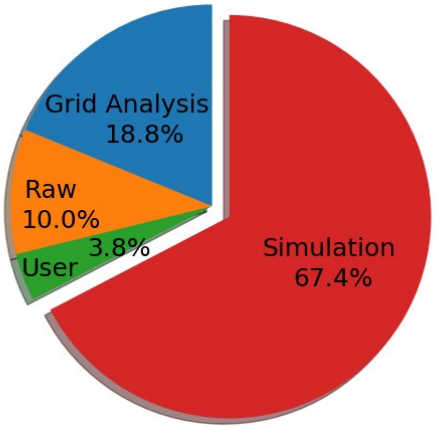


ALICE resource demands during LHC Run 2

Challenges for ALICE in LHC Run 3

- Up to ~100 times more data expected compared to Run 2
  - Not possible to increase simulation production by comparable factor
    [during Run 2 ~2/3 of the resource were dedicated to simulation]
  - Need smarter, software-based solutions to make the detector simulation more efficient
- Need for what is generically called fast simulation and hence interfacing
- GEANT will still be considered to be the default, BUT identify most demanding
  - ...sub-detectors,
  - ...phase-space regions,
  - ...particle types

...and introduce more efficient simulation process if possible
[such as parametrisation, Machine-Learning based approaches]



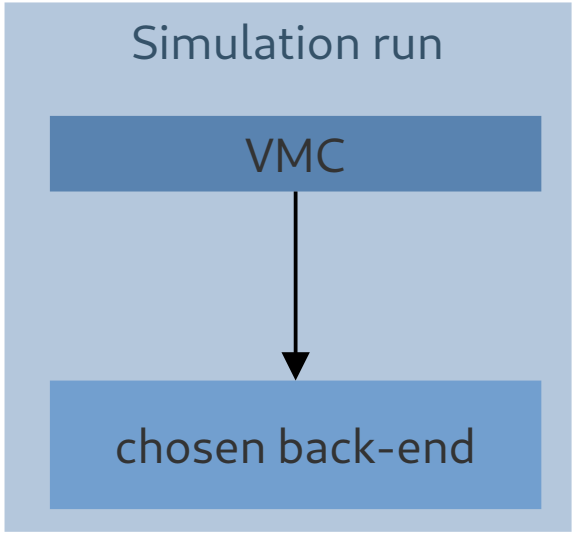
ALICE resource demands during LHC Run 2

How can that be covered by the VMC package?

Previous limitations of the VMC package

Not capable of partitioning the event simulation among multiple different engines
→ no interaction with any other simulation/fast simulation possible during a full simulation

previous



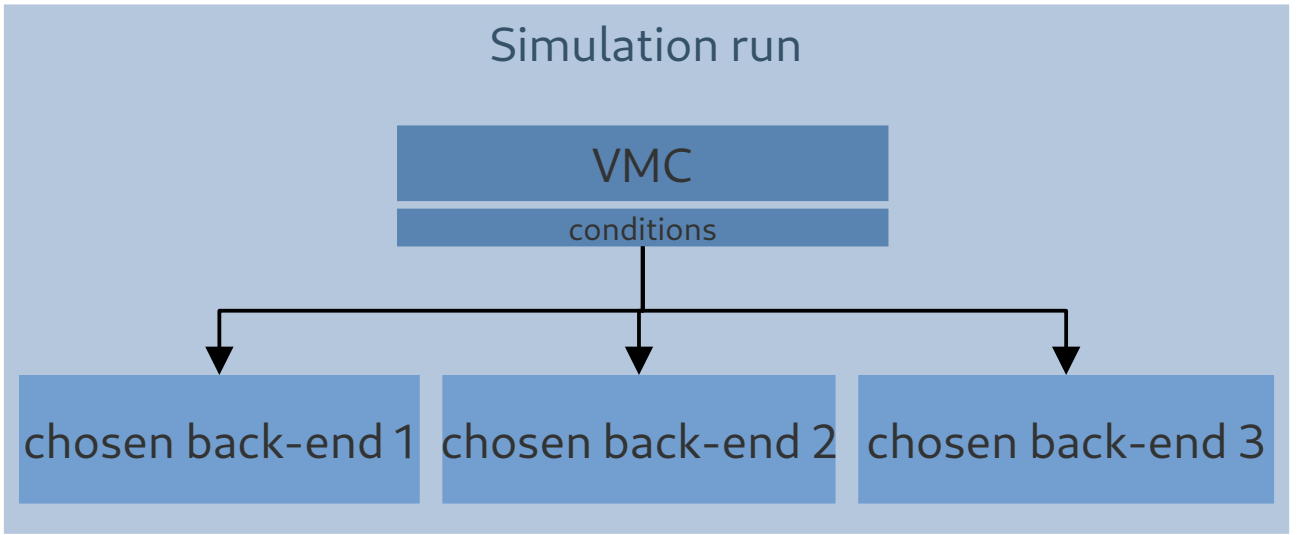
Previous limitations of the VMC package

Now capable of partitioning the event simulation among multiple different engines and is therefore capable of running full and fast simulation engines together

- Conditions can be based on
  - Geometry
  - Particle type
  - Phase space
  - Any combination of those

→ different tracks are transported by different engines

now



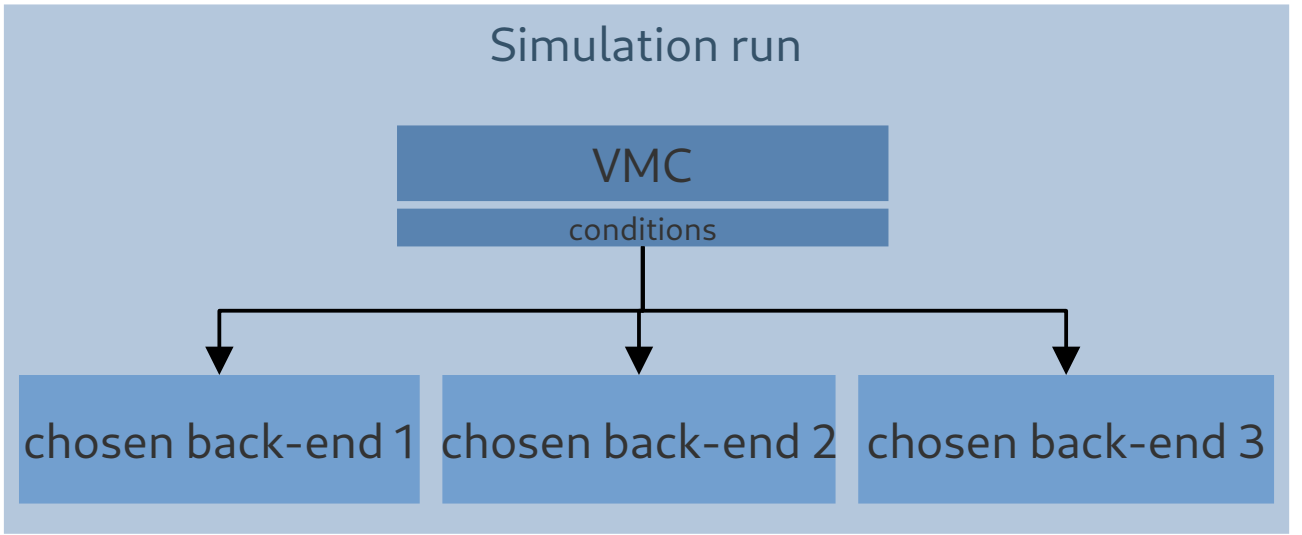
Previous limitations of the VMC package

Now capable of partitioning the event simulation among multiple different engines and is therefore capable of running full and fast simulation engines together

- Conditions can be based on
  - Geometry
  - Particle type
  - Phase space
  - Any combination of those

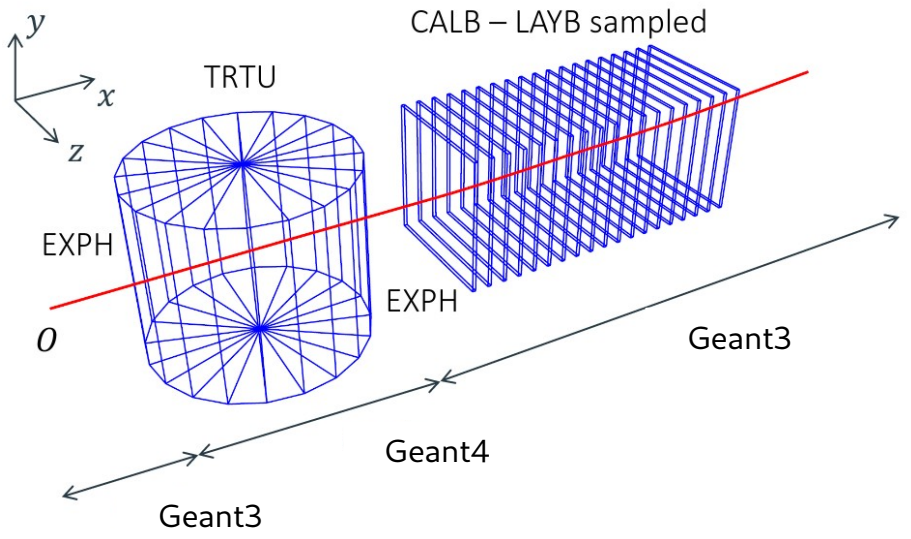
→ different tracks are transported by different engines

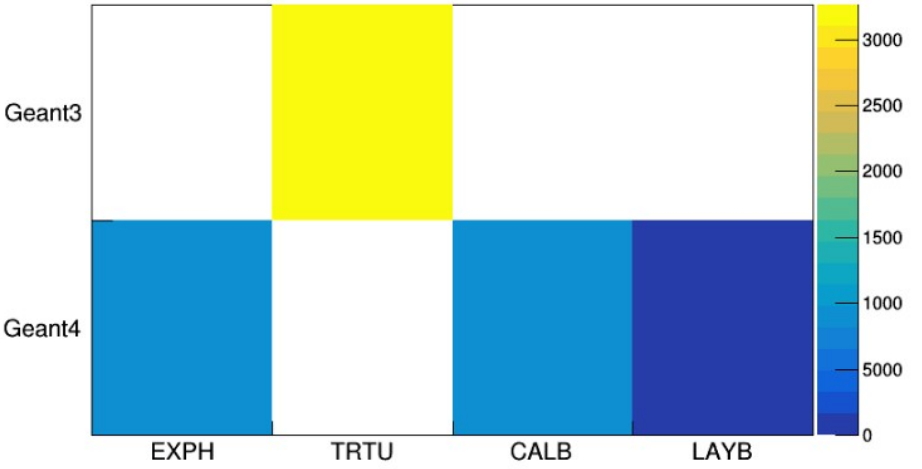
now



Done, thank you for your attention…

Test whether the partitioning works in principle

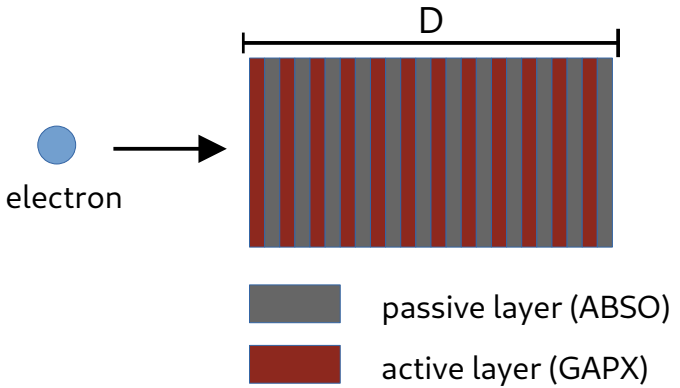




Steps per volume and engine

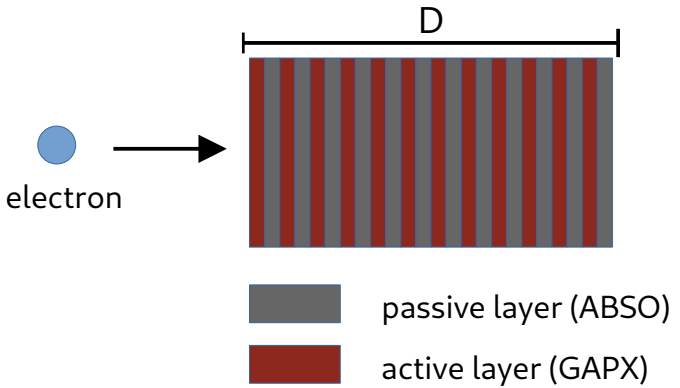
- Shoot particles through a geometry and register where each engine produces steps
- Steps are made according to the desired partitioning

Test potential overhead introduced by track transfer



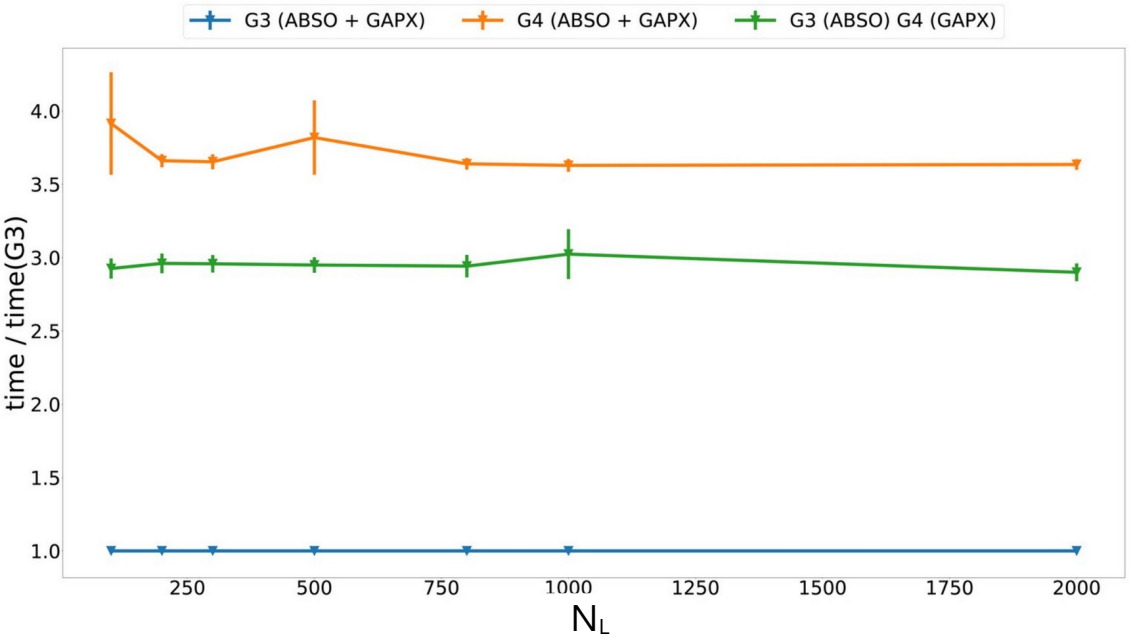
- Consider 3 scenarios, running with
  1) ...only GEANT3
  2) ...only GEANT4
  3) ...GEANT3 for passive and GEANT4 for active layers
     → partitioning based on geometry conditions

Test potential overhead introduced by track transfer



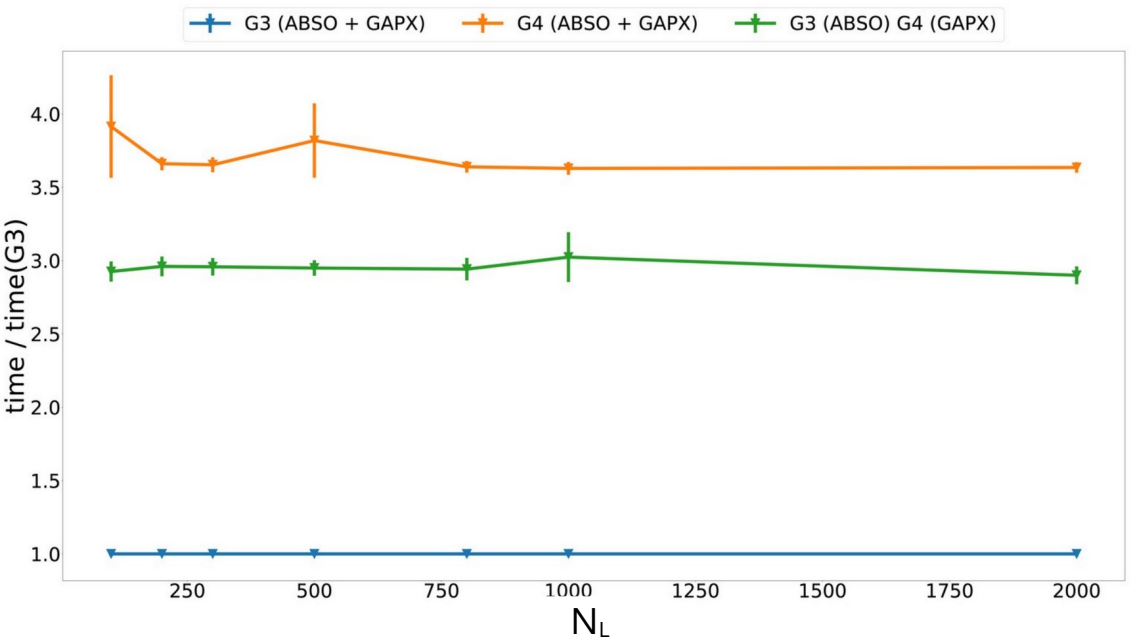
- Consider 3 scenarios, running with
  1) ...only GEANT3
  2) ...only GEANT4
  3) ...GEANT3 for passive and GEANT4 for active layers
     → partitioning based on geometry conditions

- Simulation time normalised to G3
- Increasing the number of layers N_L while keeping depth D constant
  → increasing number of passing the simulation back and forth between the back-ends



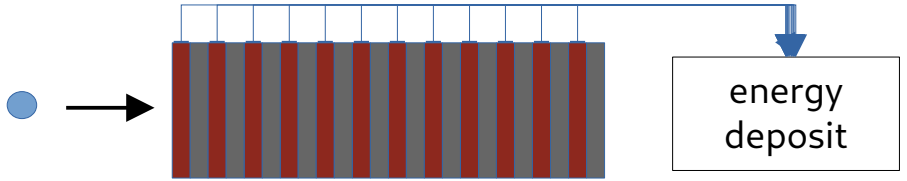
Test potential overhead introduced by track transfer

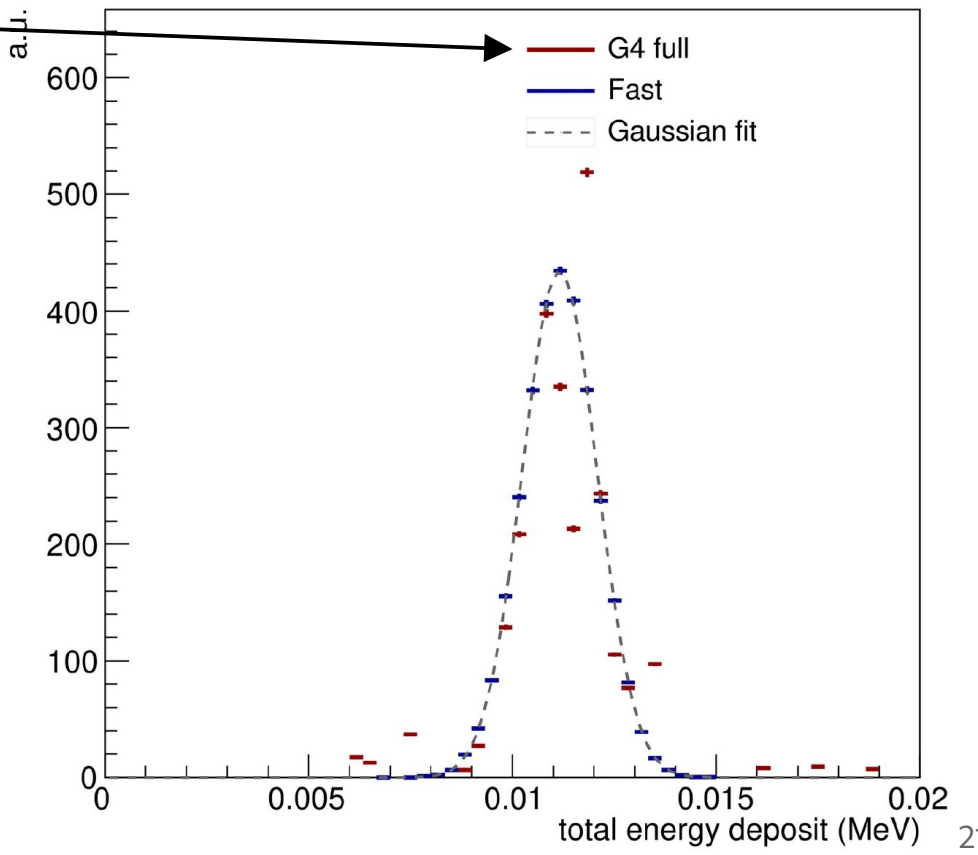
- GEANT4 simulation takes ~3.5 times as long as GEANT3 (more detailed and higher complexity of physics simulation)
- Let GEANT3 simulate passive volumes while keeping higher accuracy with GEANT4 in active layers decreases performance significantly
- The fact that the overall ratio of the partitioned scenario is flat indicates that the transferring of tracks between the engines does not introduce significant overhead during the computation
- This can be generalised to more involved scenarios…



Partitioning with a custom “fast simulation” (proof-of-principle)

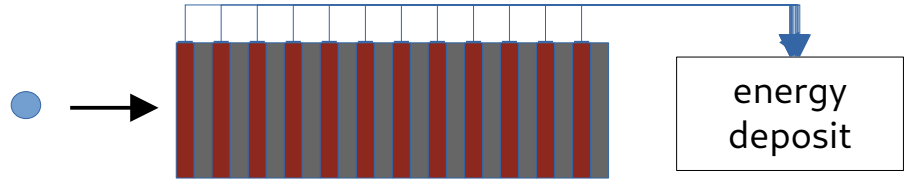
- Use the same calorimeter setup as before and run it with GEANT4 to obtain the energy deposit distribution (red)
- Fit Gaussian to distribution
  [of course, this is very simplified]
- In a second run, let GEANT4 do the transport up to the calorimeter
  [hence in the WORLD volume]
- As soon as the particle reaches it, dispatch to a custom “fast simulation” that draws the energy deposit from the aforementioned distribution and produces the energy deposit
- This straightforward approach is already more than 10 times faster

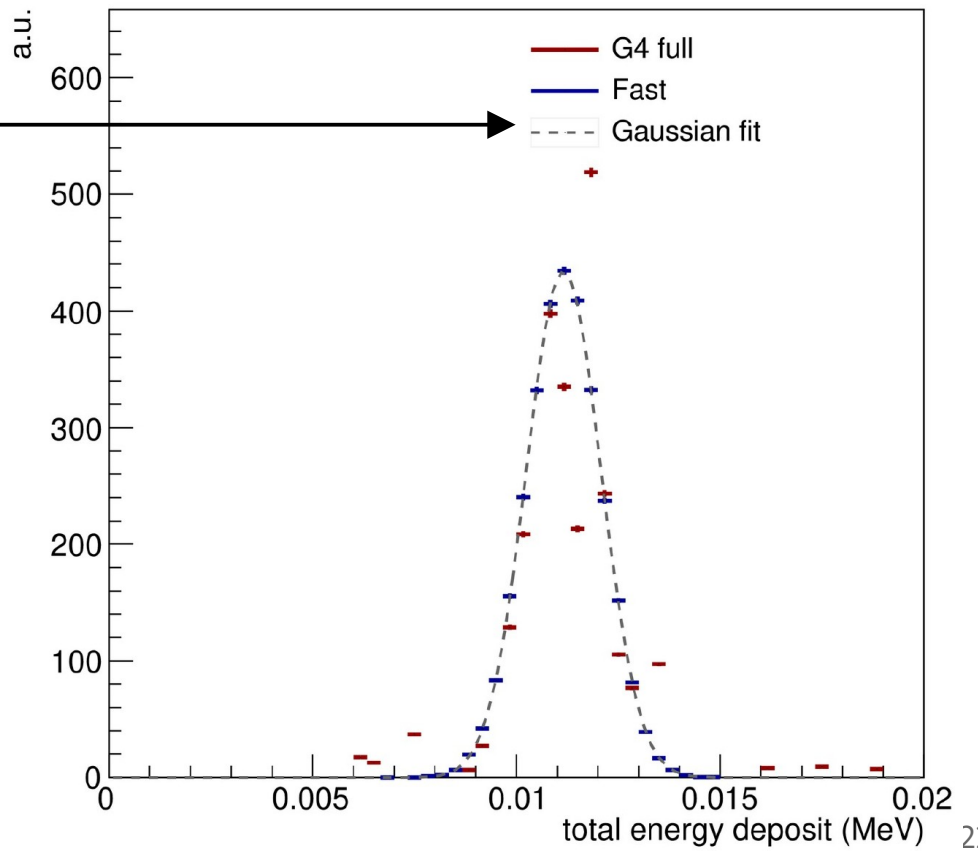




Partitioning with a custom “fast simulation” (proof-of-principle)

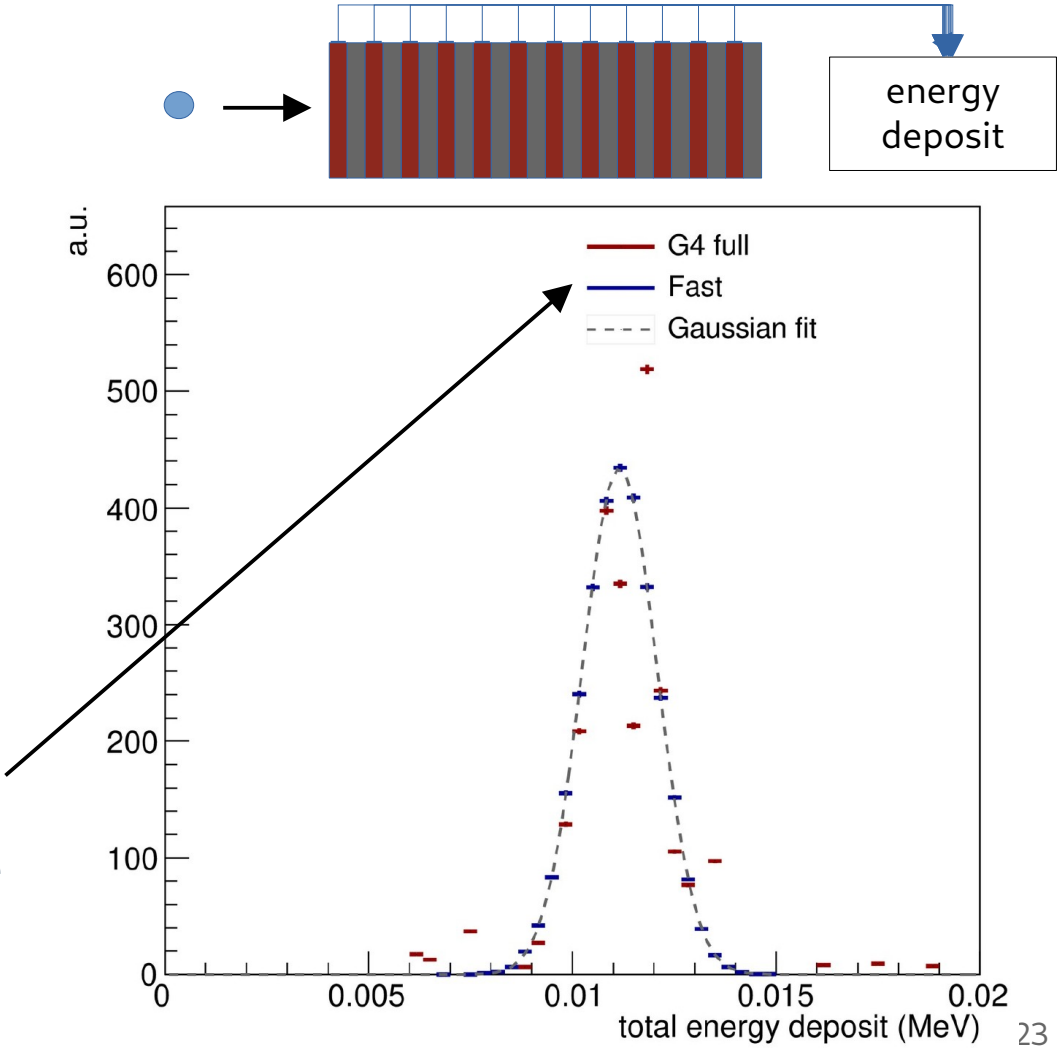
- Use the same calorimeter setup as before and run it with GEANT4 to obtain the energy deposit distribution (red)
- Fit Gaussian to distribution
  [of course, this is very simplified]
- In a second run, let GEANT4 do the transport up to the calorimeter
  [hence in the WORLD volume]
- As soon as the particle reaches it, dispatch to a custom “fast simulation” that draws the energy deposit from the aforementioned distribution and produces the energy deposit
- This straightforward approach is already more than 10 times faster





Partitioning with a custom “fast simulation” (proof-of-principle)

- Use the same calorimeter setup as before and run it with GEANT4 to obtain the energy deposit distribution (red)
- Fit Gaussian to distribution
  [of course, this is very simplified]
- In a second run, let GEANT4 do the transport up to the calorimeter
  [hence in the WORLD volume]
- As soon as the particle reaches it, dispatch to a custom “fast simulation” that draws the energy deposit from the aforementioned distribution and produces the energy deposit
- This straightforward approach is already more than 10 times faster

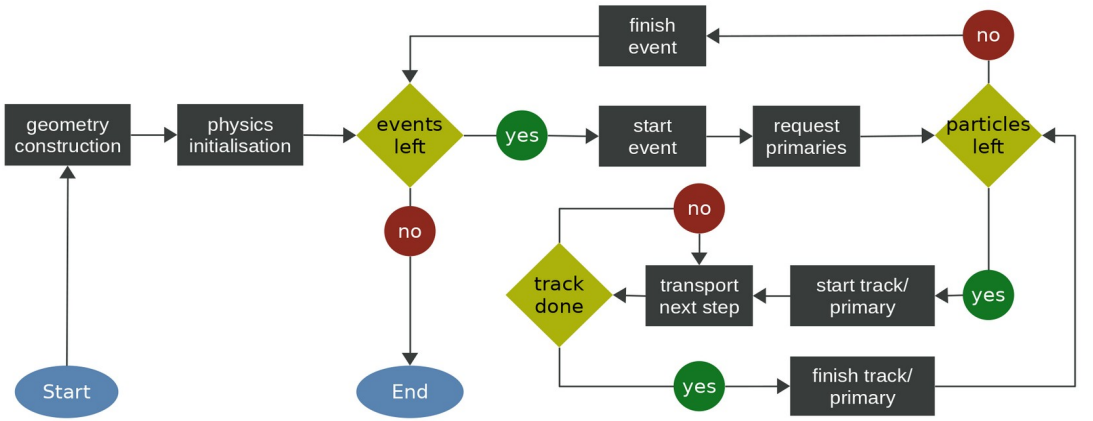


Design considerations

- VMC has been used in production for many years already and is used by the Fair and ALICE collaboration → hence many users
  - Ensure the developments are **fully backward-compatible** and do not break current implementations in any software stack
- Minimise potential run time overhead when
  - tracks are paused/resumed and particle stacks are updated → **no copying of track objects**
  - geometry states have to be re-initiated fast when tracks are picked up again → **cache**
- Hide implementations details as much as possible and automatise all necessary workflows → to a user it looks **as if one transport engine was running**
- **Mapping the requirements to the code took a significant amount of time; previous VMC and interface implementations have several 10,000 lines of code and consist of many C++ classes and interactions among those**

The generic VMC workflow

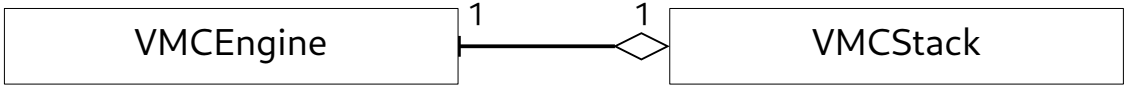
- Detector geometry is initiated
- Physics models are loaded
- For each event primary particles are requested to be transported
  [originate for instance from an MC event generator]
- Each primary is transported through the detector geometry



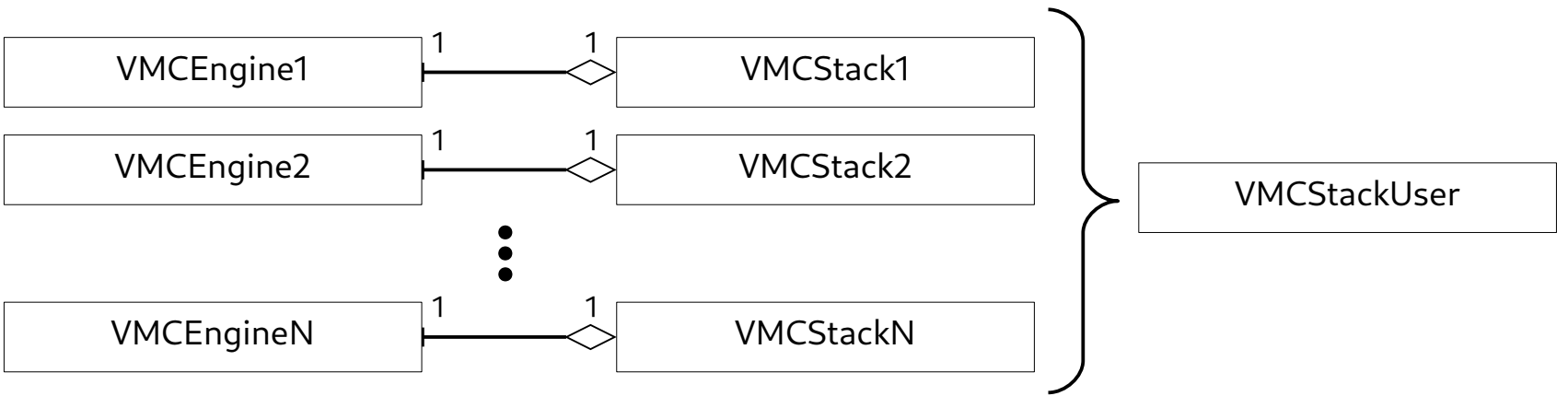
- At each stage indicated by a dark-blue box the user can inject additional routines
  [e.g. to manage the primary particle generation, to discard tracks during the simulation etc]
- Transport is done when all primaries (and produced secondaries) have been processed

The implementation I

- Particles to be transported are pushed to the engine's stack. In the previous implementation, one particle stack was associated to an engine and visible to the user to interact with it



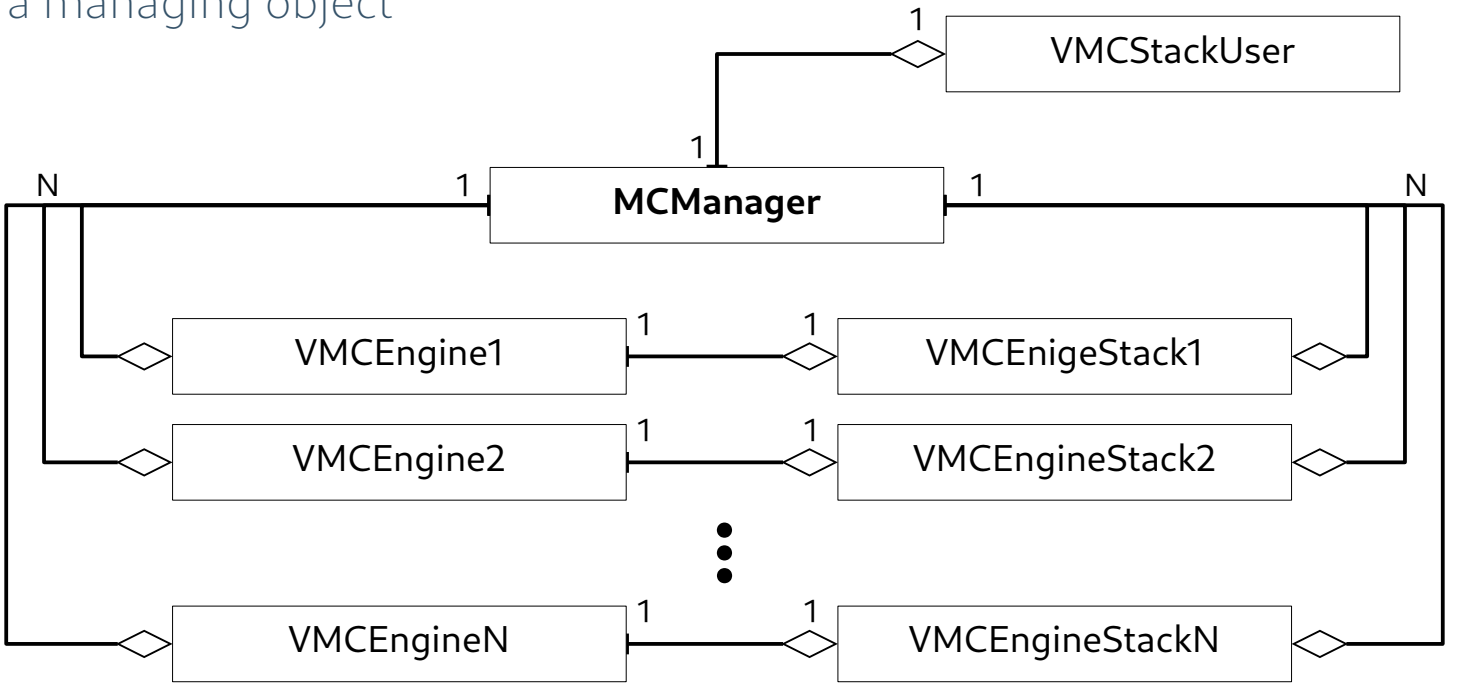
- But what is needed is rather



- Need control over multiple instances of engines and stacks, synchronise with what the user sees on the VMCStackUser

The implementation II

- Introduce a managing object

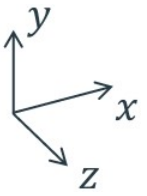


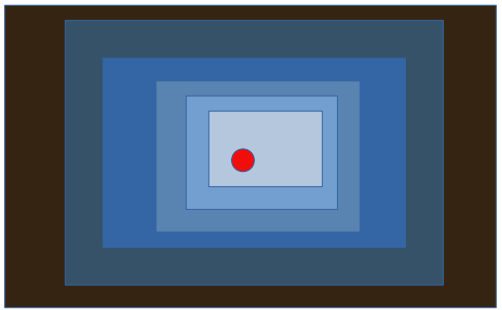
- The **MCManager** is now the central object
  - Always knows the currently running engines
  - Automatically synchronises the engines' stacks with the user stack
  - Fully contains the control of the event loop

Indeed, the whole picture collapses to the previous one in case only one engine is running

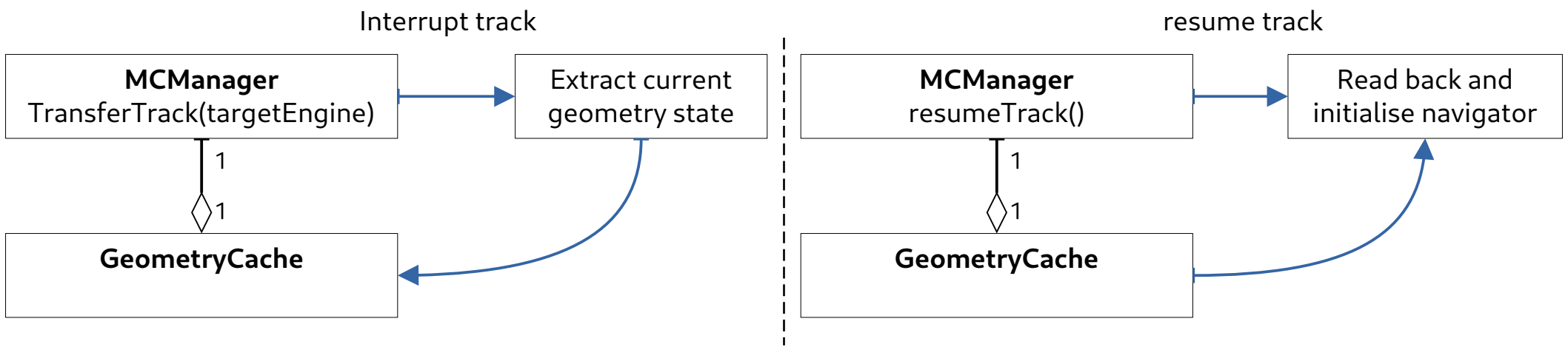
The implementation III

- When a track is transported, a so-called navigator is used to find volumes in a geometry tree based on spatial coordinates
- Very expensive task in a complex and deep structure
  → avoid that when a paused track is resumed in another engines



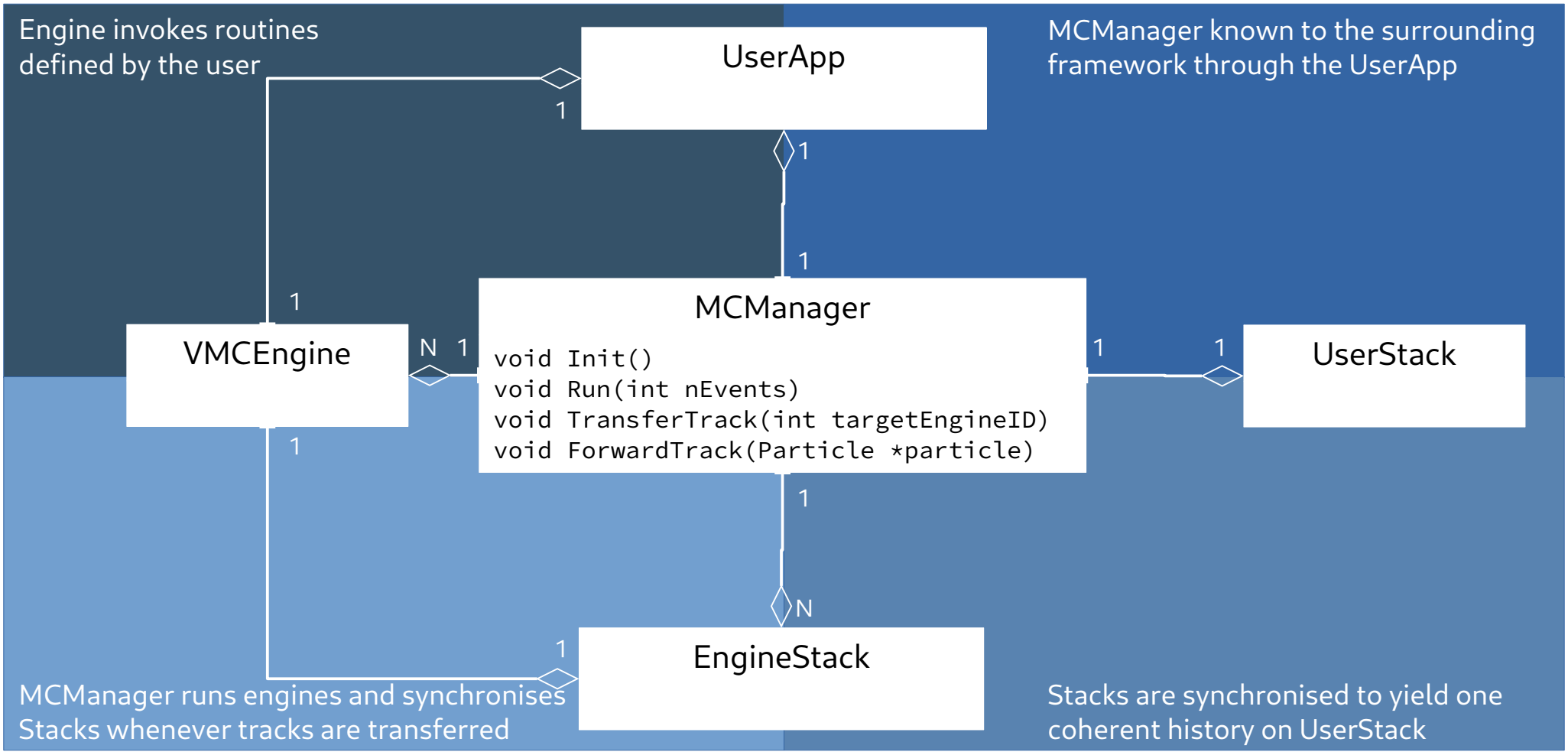


Find deepest volume containing position



- No explicit search in geometry tree necessary for paused tracks

The implementation IV



Overall design consideration

- For an interface/framework like the VMC there are 2 principal design choices
  - **Run-time polymorphism** (“inheritance from virtual/abstract base class”)
  - **Compile-time** polymorphism (basically templating in C++)

Another overall design decision

- For an interface/framework like the VMC there are 2 principal design choices
  - Run-time polymorphism (“inheritance from virtual/abstract base class”)
  - Compile-time polymorphism (basically templating in C++)

```
VMCEngine *engine = new TGeant4();
engine->run(42);
```

```
class VMCEngine
{
  // purely virtual in base class
  virtual bool run(int nEvents) =
0;
}

Class TGeant4 : public VMCEngine
{
  virtual bool run(int nEvents)
  {
    // implementation
  }
}
```

- A virtual table is available providing to find correct implementation of TGEant4::run() although it is called on type VMCEngine
- This lookup is fast, however, has to be done for each such call
- On the other hand, since working with VMCEngine pointers, all can be treated the same
  [remember that C++ is strictly type safe at compile time]

```
std::vector<VMCEngine> = {TGeant3(), TGeant4()}; // won’t work
std::vector<VMCEngine*> = {new TGeant3(), new TGeant4()}; // works
```

Another overall design decision

- For an interface/framework like the VMC there are 2 principal design choices
  - Run-time polymorphism (“inheritance from virtual/abstract base class)
  - Compile-time polymorphism (basically templating in C++)

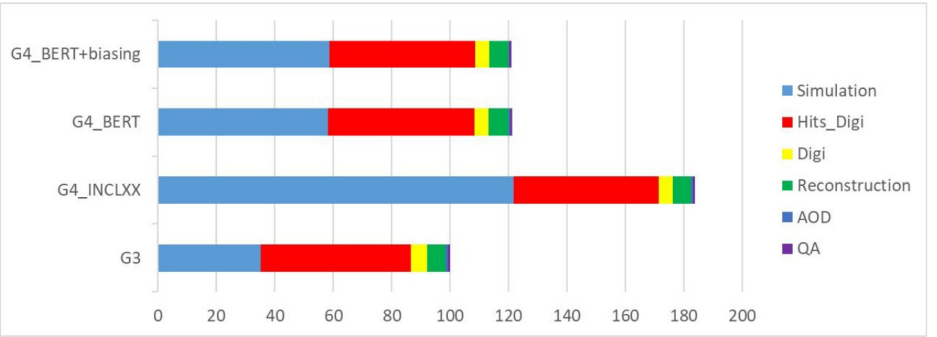
```
template <typename T>
class VMCEngine
{
  // purely virtual in base class
  virtual bool runUser() = 0;
  Virtual bool run()
  {
    static_cast<T*>(this)->runUser();
  }
}

Class TGeant4 : public VMCEngine<TGeant4>
{
  virtual bool runUser()
  {
    // implementation
  }
}
```

- Correct type can be determined at compile time
- Potentially faster
- However, stuck to run-time polymorphism; main reason: backward-compatibility

The status of the ALICE detector simulation framework

- Optimisations and QA of GEANT4 went/goes on in parallel
- Run time of GEANT4 depends on physics list and increases by ~2 using INCLXX
  (better and required accuracy for light nuclei)
  [physics list provides modelling of interactions and decays]



Overall time normalised to GEANT3 [%]

- However, now possible to only use INCLXX in in beam pipe and ITS and BERT everywhere else
- GEANT4 performance becomes compatible in terms of run time with GEANT3
- GEANT4 provides fast simulation interfaces to inject user-specific fast simulation providing very similar freedom compared to above presented VMC extensions
- However, exact implementation of potential fast simulation still to be decided, furthermore, the VMC implementation works also independent of GEANT4

Conclusion

- Extended the VMC package to partition the event simulation among multiple transport engines
- Principal commits
  - VMC: https://github.com/vmc-project/vmc/commit/ce3fa3004d32925ea3c6c7f7b0728453d25e6df9
  - GEANT3 interface: https://github.com/vmc-project/geant3/commit/74ab9dde34ebd906a7fa05ffe28932497299b4a6
  - GEANT4 interface: https://github.com/vmc-project/geant4_vmc/commit/fd51ec51a2b50616f44f1e96a7bd00e5b2298c94
- Overall almost 10,000 lines of code added/modified/deleted
- Fully backward compatible and current state of code used by Fair and ALICE
- VMC package ready to partition among full and fast simulation engines
  → can be used to inject fast simulation based on concepts like parametrisation up to ML

Conclusion

- Extended the VMC package to partition the event simulation among multiple transport engines
- Principal commits
  - VMC: https://github.com/vmc-project/vmc/commit/ce3fa3004d32925ea3c6c7f7b0728453d25e6df9
  - GEANT3 interface: https://github.com/vmc-project/geant3/commit/74ab9dde34ebd906a7fa05ffe28932497299b4a6
  - GEANT4 interface: https://github.com/vmc-project/geant4_vmc/commit/fd51ec51a2b50616f44f1e96a7bd00e5b2298c94
- Overall almost 10,000 lines of code added/modified/deleted
- Fully backward compatible and current state of code used by Fair and ALICE
- VMC package ready to partition among full and fast simulation engines
  → can be used to inject fast simulation based on concepts like parametrisation up to ML

Thank you for your attention

BACKUP

Implementation details of the TMCManager I

TMCManager

`void ForwardTrack(Int_t toBeDone, Int_t trackId,`
`  Int_t parentId,`
`  TParticle* particle)`
`void TransferTrack(Int_t targetEngineId)`
`...`

TVirtualMCApplication
`void RequestManager()`
`TMCManager* fMCManager`

TMCManagerStack
A concrete implementation of `TVirtualMCStack` *providing the interfaces accordingly for the usage and communication with the* `TMCManager`.

- TMCManager is a singleton object
- Requested by the UserApplication (otherwise fall back to single engine run), no TMCManager is constructed
- TVirtualMC objects are owned by the TMCManager and automatically registered when instantiated
- TMCManager handles
  - Communication between engines
  - Pausing and resuming engines as needed
  - Transferring tracks between stacks

Implementation details of the TMCManager I

TMCManager

```
void SetUserStack(TVirtualMCStack* userStack)
void ForwardTrack(Int_t toBeDone, Int_t trackId,
                  Int_t parentId,
                  TParticle* particle)
void TransferTrack(Int_t targetEngineId)
template <typename F> Apply(F f)
template <typename F> Init(F f)
void Run(Int_t nEvents)
void ConnectEnginePointer(TVirtualMC*& mc)
TVirtualMC* GetCurrentEngine()
```

```
void Ex03MCStack::PushTrack(Int_t toBeDone, Int_t parent, ..., Int_t& ntr, ...) {
  // TParticle construction yielding “particle”
  // define track ID
  ntr = GetNtrack() - 1;
  if(auto mgr = TMCManager::Instance()) {
    mgr->ForwardTrack(toBeDone, ntr, parent, particle);
  }
  // further implementation
}
```

- User is owner of created TParticle objects, hence of created particles and therefore also the numbering scheme
- ForwardTrack should be called in UserStack::PushTrack
- An optional last argument can specify the target engine ID

Implementation details of the TMCManager I

TMCManager

```
void SetUserStack(TVirtualMCStack* userStack)
void ForwardTrack(Int_t toBeDone, Int_t trackId,
                  Int_t parentId,
                  TParticle* particle)
void TransferTrack(Int_t targetEngineId)
template <typename F> Apply(F f)
template <typename F> Init(F f)
void Run(Int_t nEvents)
void ConnectEnginePointer(TVirtualMC*& mc)
TVirtualMC* GetCurrentEngine()
```

```
void Ex03MCApplication::Stepping() {
  // ...
  Int_t targetId = -1;
  if(fMC->GetId() == 0 && strcmp(fMC->GetCurrentVol(), “ABSO”) == 0) {
    targetId = 1;
  } else if(fMC->GetId() == 1 && strcmp(fMC->GetCurrentVol(), “GAPX”) == 0) {
    targetId = 0;
  }
  // ...
  fMCManager->TransferTrack(targetId);
}
```

- Called most likely in UserApp::Stepping
- Interrupts transport and transfers track to target engine stack
- Decision can be made based on geometry, particle type, phase space…

Implementation details of the TMCManager I

TMCManager

```
void SetUserStack(TVirtualMCStack* userStack)
void ForwardTrack(Int_t toBeDone, Int_t trackId,
                  Int_t parentId,
                  TParticle* particle)
void TransferTrack(Int_t targetEngineId)
template <typename F> Apply(F f)
template <typename F> Init(F f)
void Run(Int_t nEvents)
void ConnectEnginePointer(TVirtualMC*& mc)
TVirtualMC* GetCurrentEngine()
```

```
void Ex03MCApplication::InitMC(
 std::initializer_list<const char*> setupMacros) {
  // ...
  fMCManager->Init([this](TVirtualMC* mc) {
    mc->SetRootGeometry();
    mc->SetMagField(fMagField);
    mc->Init();
    mc->BuildPhysics();
  });
  // ...
}
```

```
Ex03DetectorConstruction::Ex03DetectorConstruction() {
  // ...
  if(auto mgr = TMCManager::Instance()) {
    mgr->ConnectEnginePointer(fMC);
  }
  // ...
}
```

- Type F is assumed to be callable taking TVirtualMC* as argument
- f is then applied to all registered engines
- User can register a pointer to point to the current engine

Stacks

TMCManager

```
std::vector<TParticle*> fParticles;
std::vector<std::unique_ptr<TMCParticleStatus>> fParticleStatus;
TVirtualMCStack *fUserStack;
```

TMCManagerStack

```
std::vector<TParticle*> *fParticles;
std::vector<std::unique_ptr<TMCParticleStatus>> *fParticleStatus;
Std::vector<int> fPrimaries;
std::vector<int> fSecondaries;
```

- Pointer to particles is kept in the TMCManager
- TMCManager also holds a pointer to the user's stack to be able to synchronise it
- TMCManagerStack holds pointer to the vector `std::vector<TParticle*> *fParticles` and indices of primaries and secondaries to be transported by the connected engine
  → no duplication of TParticle objects, especially, since they are owned by the user