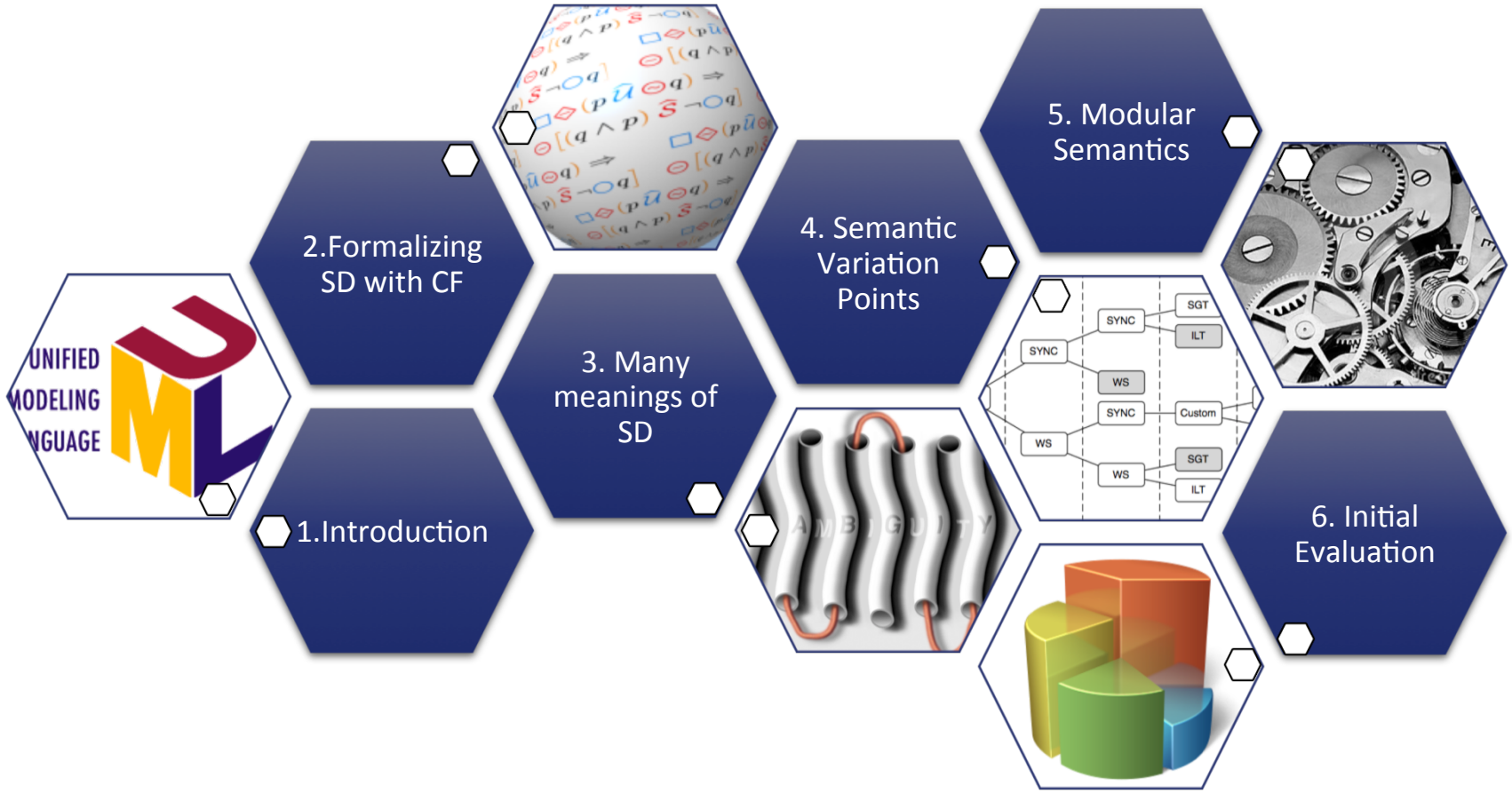
Flexible Modular Formalization of UML Sequence Diagrams

Luciano Baresi, Mohammad Mehdi Pourhashem K., Matteo Rossi

Politecnico di Milano (DEIB)

2nd FME Workshop on Formal Methods in Software Engineering (FormaliSE), 2014.

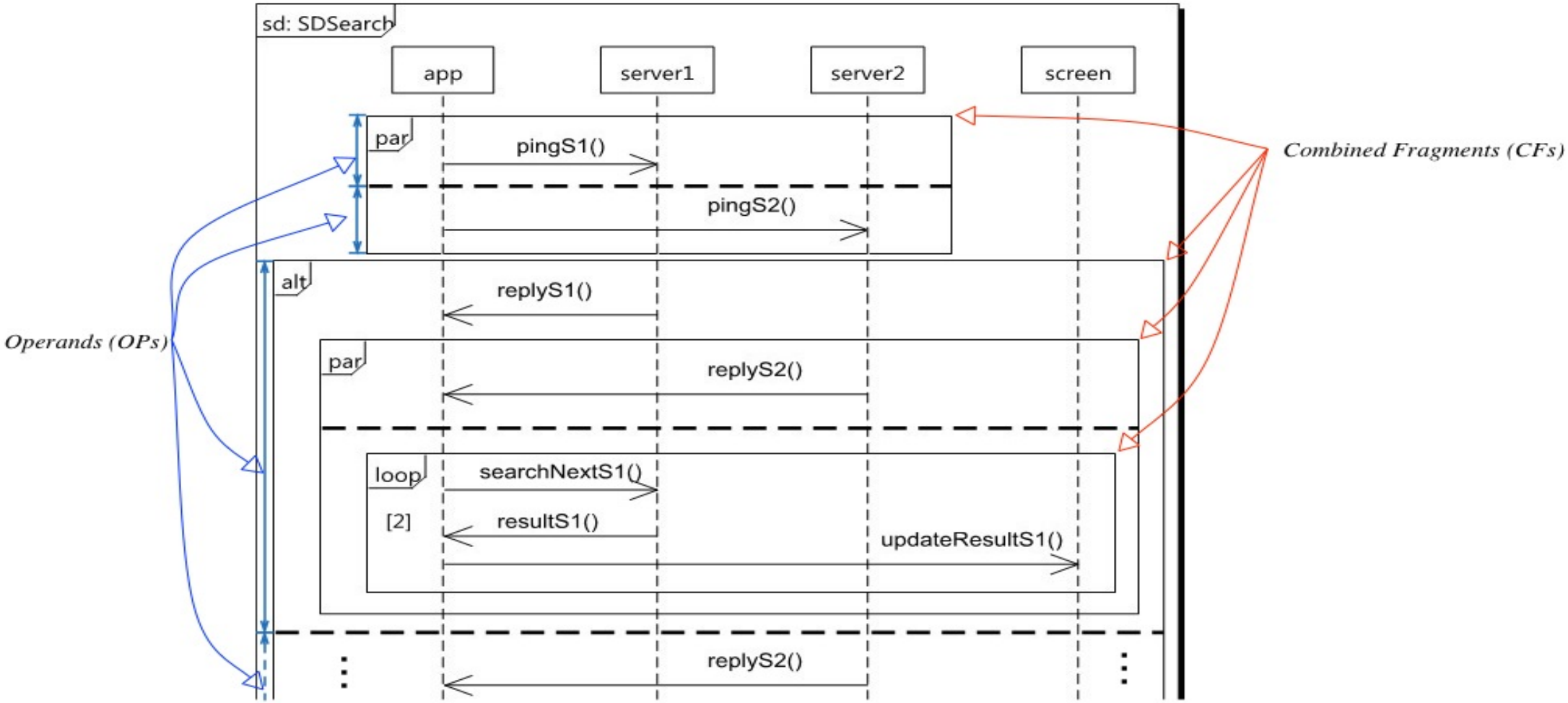
Outline


UNIFIED
MODELING
LANGUAGE
1.Introduction
2.Formalizing SD with CF
3. Many meanings of SD
4. Semantic Variation Points
5. Modular Semantics
6. Initial Evaluation
SYNC
SGT
ILT
SYNC
WS
SYNC
Custom
WS
SGT
WS
ILT
AMBIGUITY

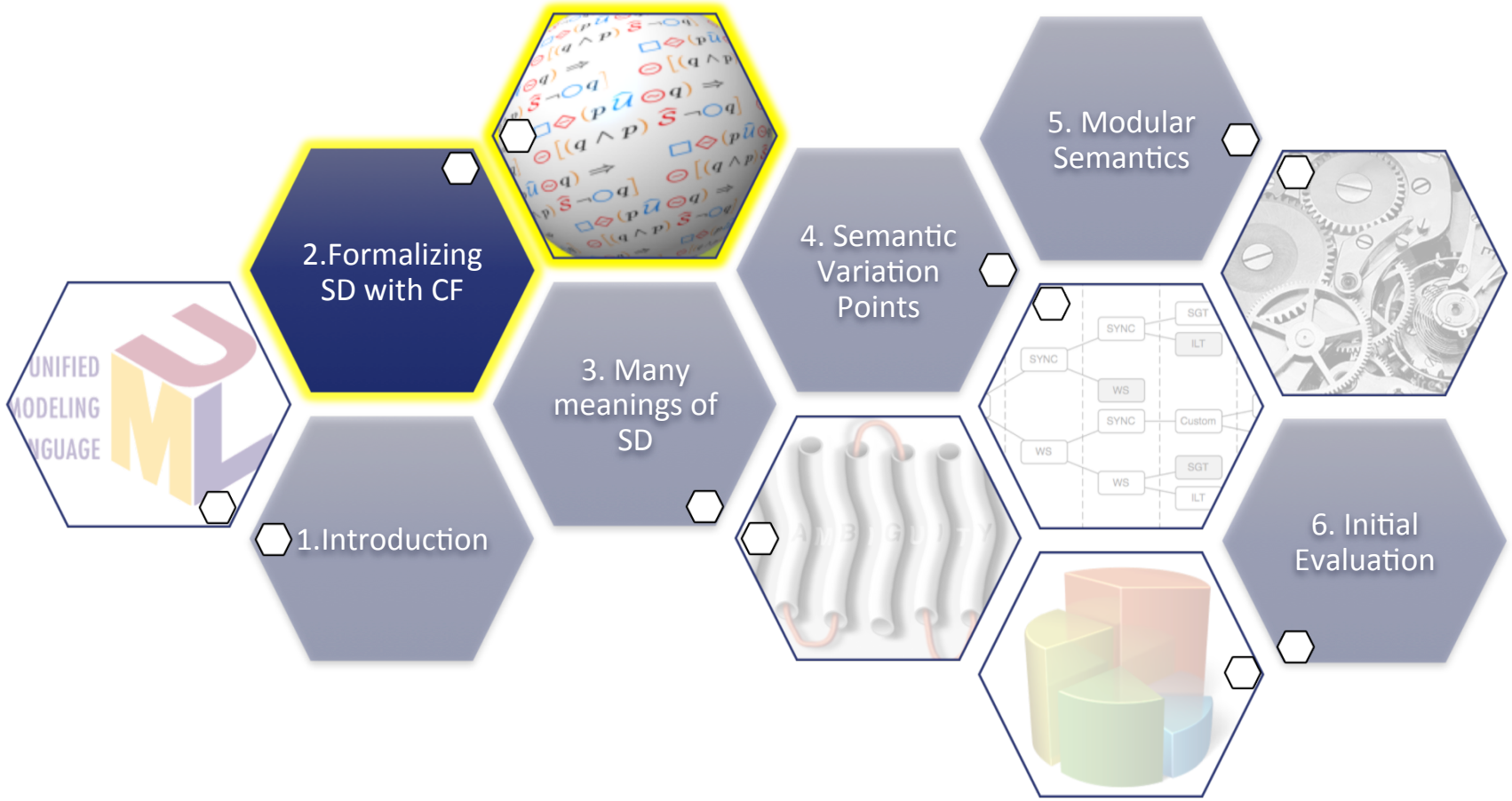
1.1 UML Formal Verification

- Model defects are a significant concern as model transformations and code generation may propagate errors to other notations where they are harder to detect and trace.
- UML formal verification comes into help by
  - Transforming UML models into mature and well-know formalism (Petri net, Temporal Logic, LTS, …)
  - Checking the correctness of user-defined properties with respect to the formal model.

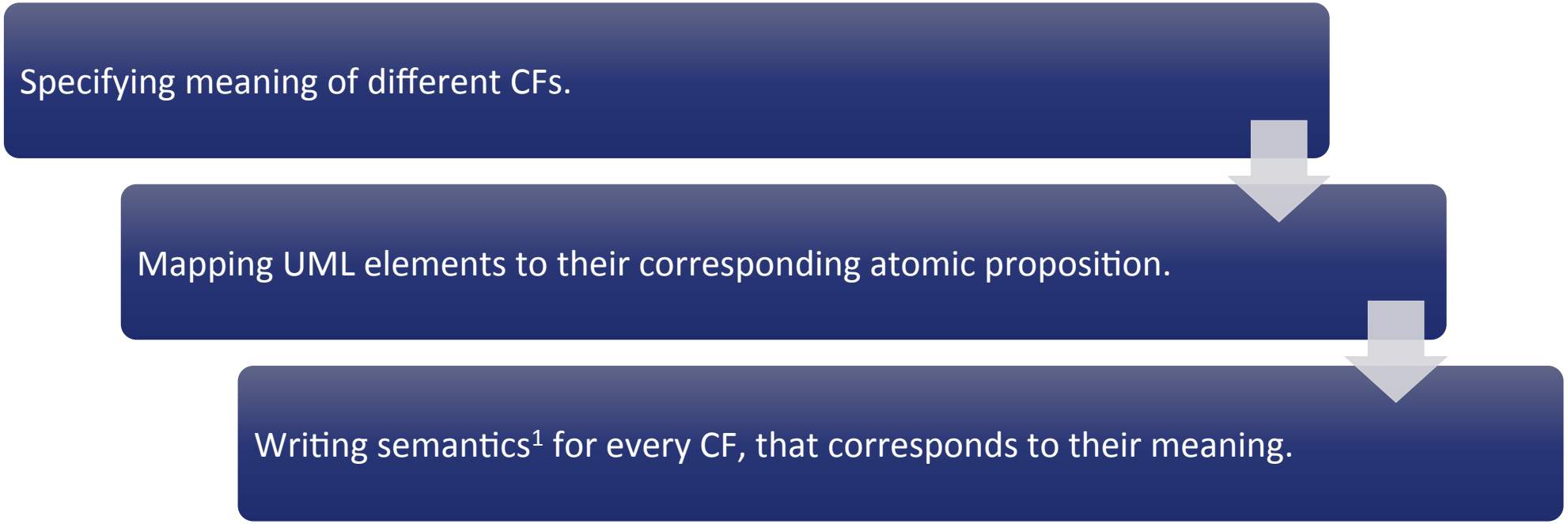
1.2 Sample Sequence Diagram (SD)



2. Formalizing SD with CF

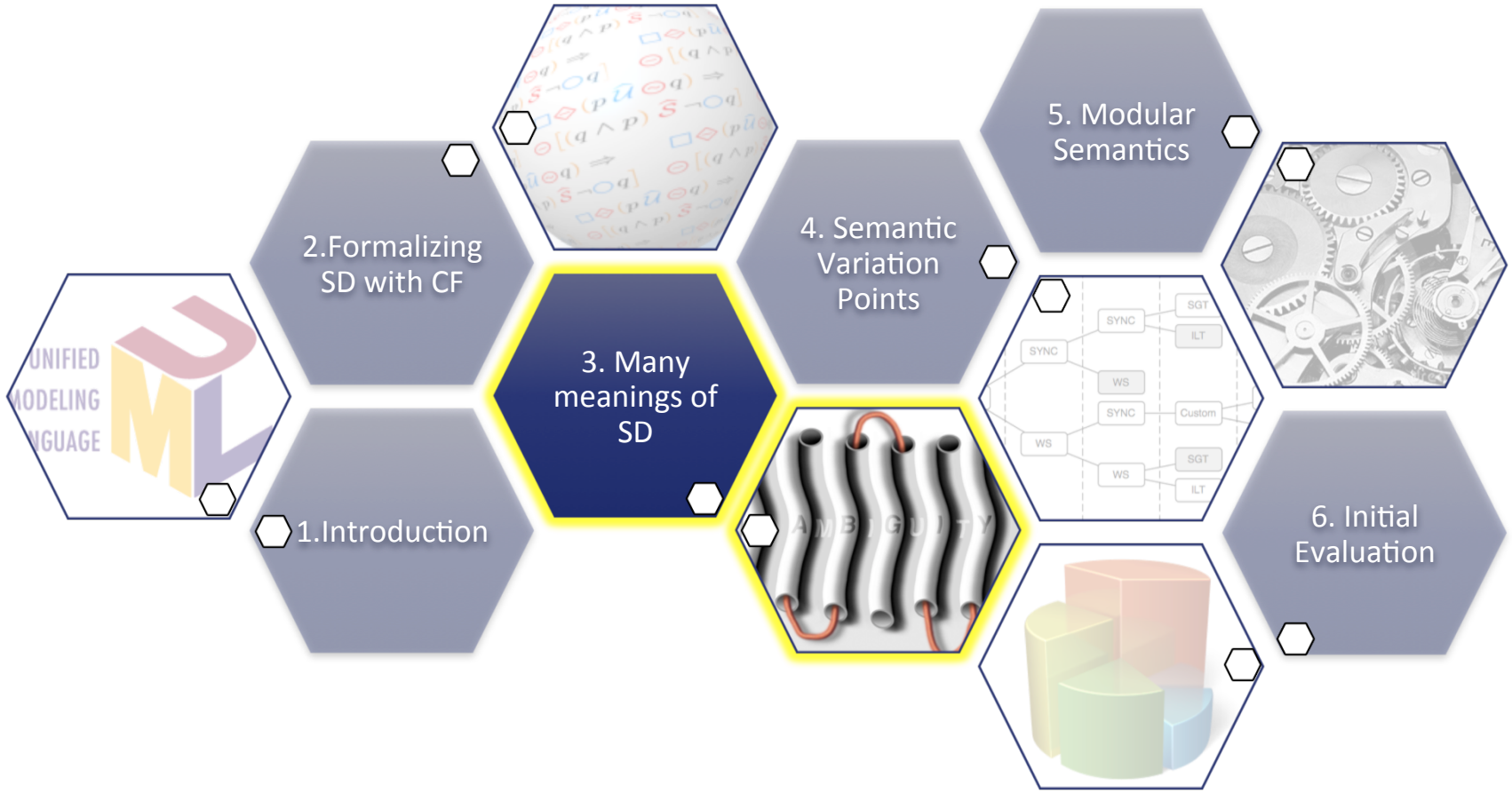


2.1 Formalizing SD with Combined Fragment (CF)?



1. Formal semantics are written in TRIO logic (on which formal methods researchers at Politecnico di Milano have been working more than 20 years).

3. Many meanings of SD



3.1 Many Meanings of SD

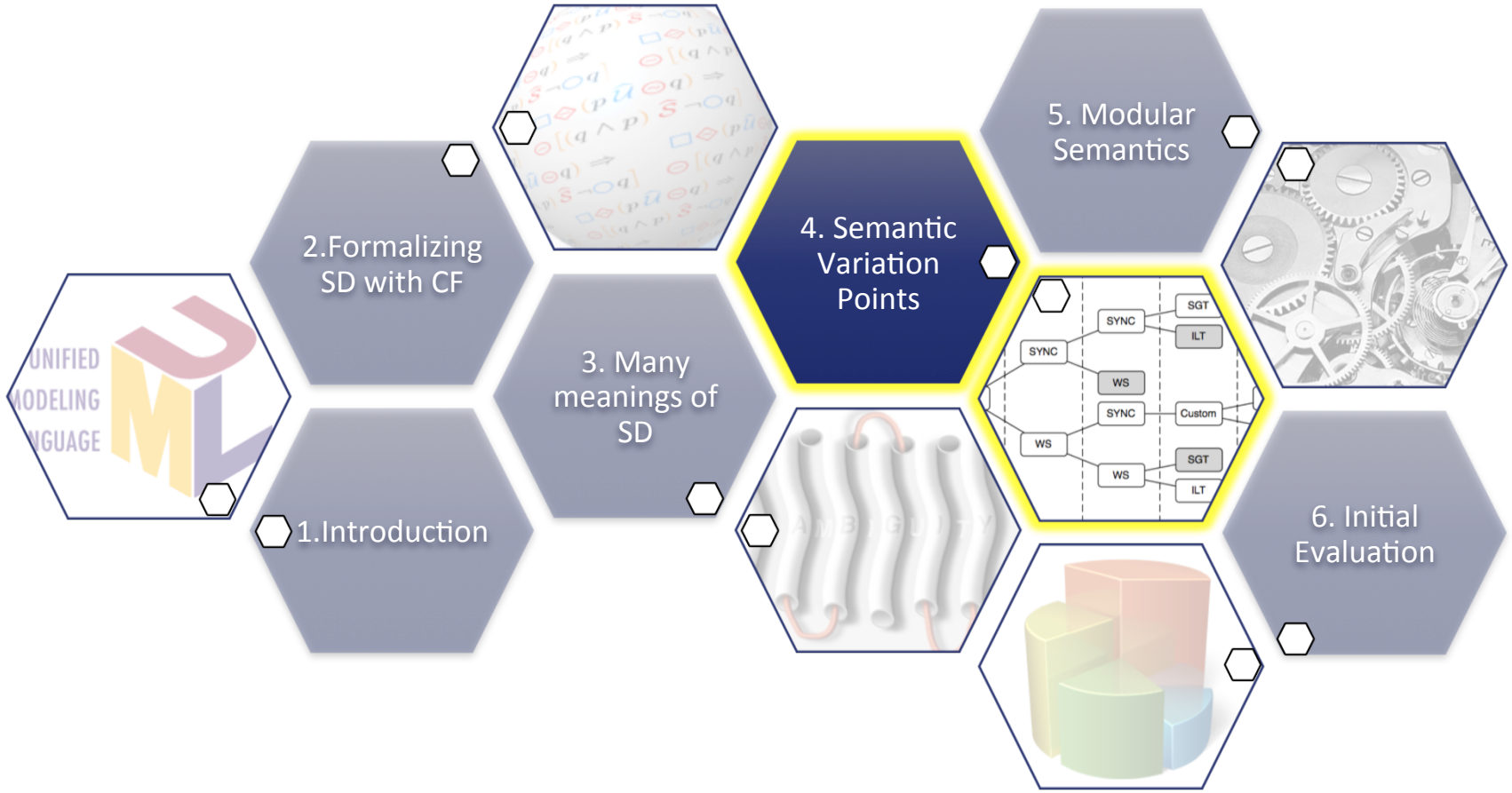
Why are there many meanings?

- OMG's semantics is not specified in detail to allow using UML in many domains. When UML is used in a concrete domain, the modeler has to choose from the different possible variants.
- Besides, some authors do not completely comply with OMG's semantics, which is very permissive and counter-intuitive in some cases.

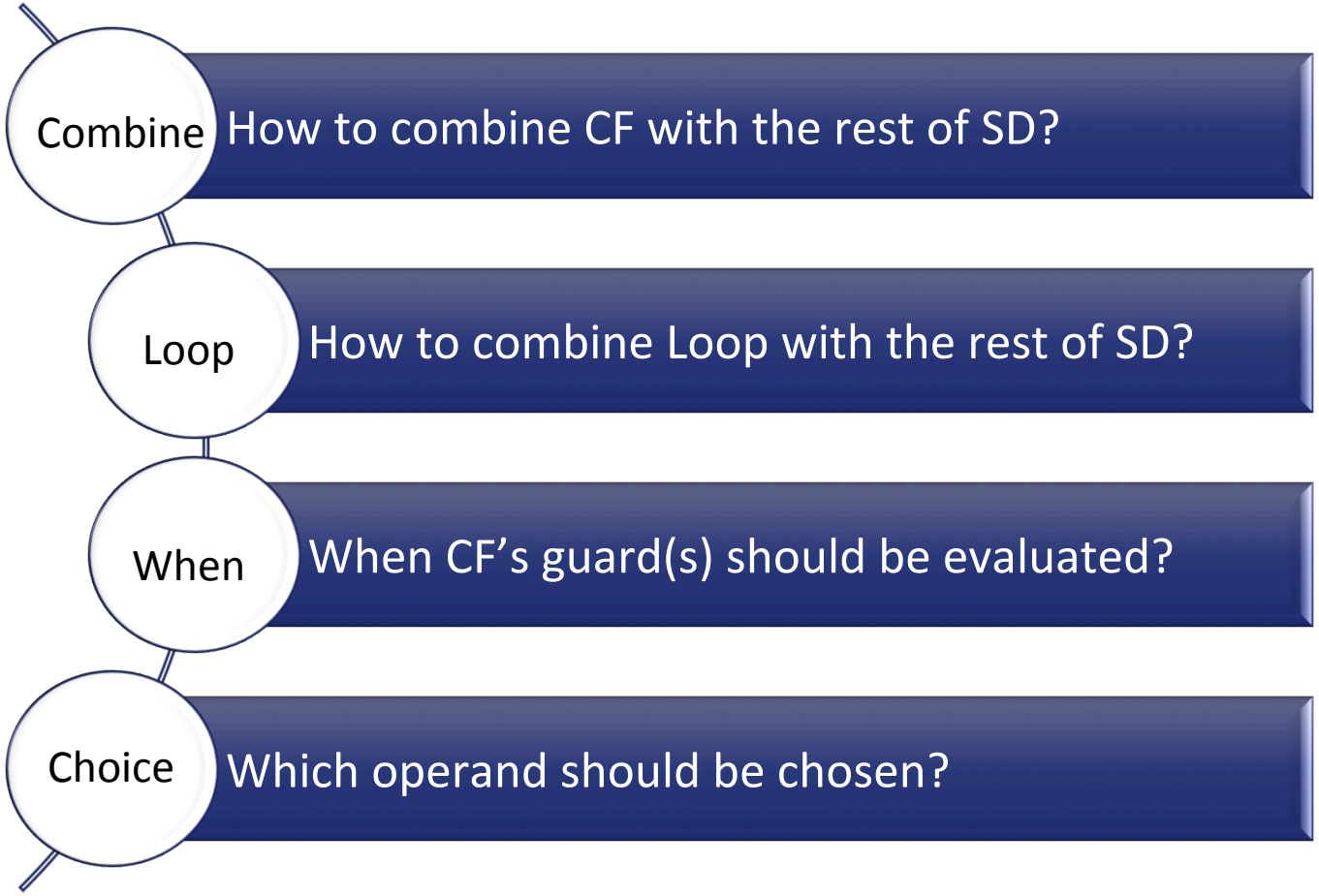
3.2 Many Meanings of SD

- More than 10 groups of authors proposed their semantics based on their interpretation and preference.
- **Configurability** is missing in their semantics, which led them to fail in realizing OMG's ambition of keeping UML useful for many domains.
- We identified those variation points that are important for verification purpose analysis.

4. Semantic Variation Points



4.1 Semantic Variation Points

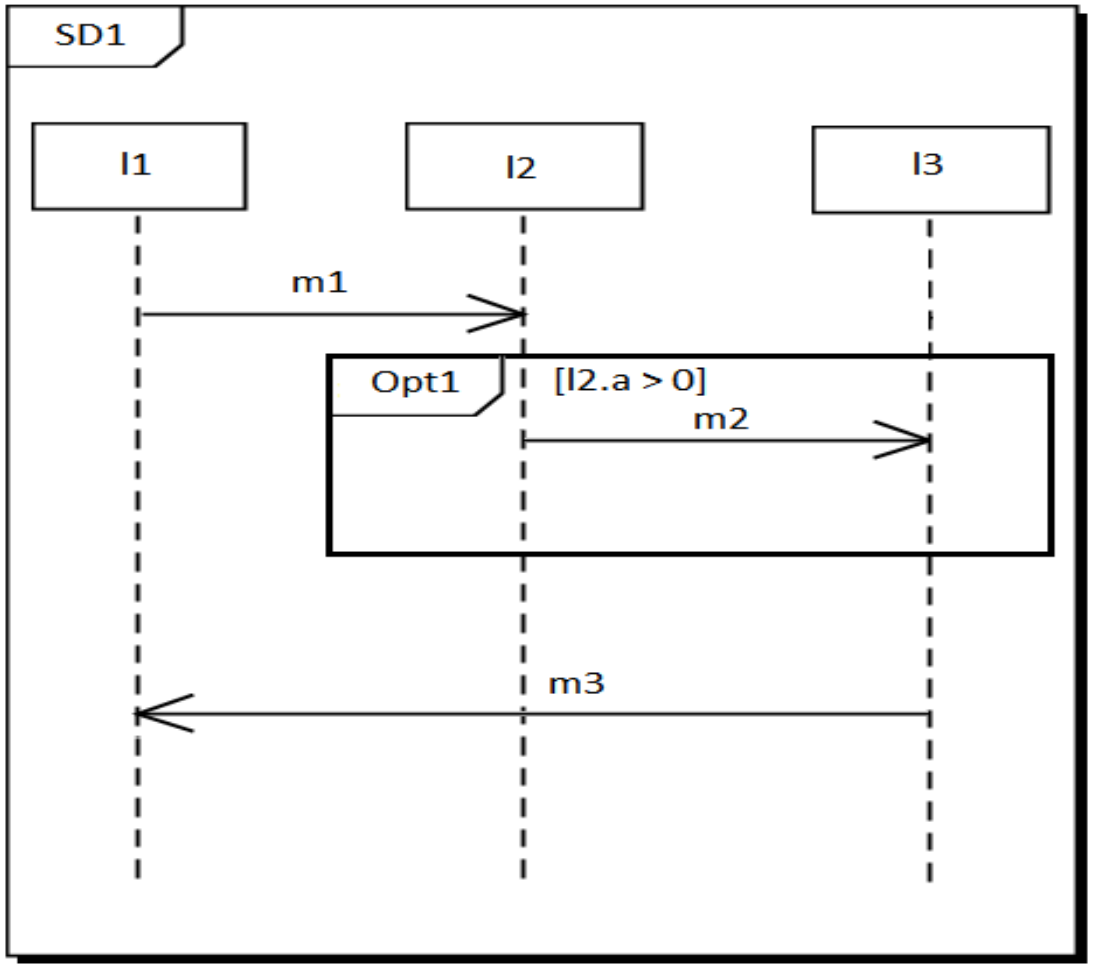


4.2 Semantic Variation Points [combine]

[Combine]: How to combine CF with the rest of SD?

1. Synchronized (**SYNC**):
   All lifelines must enter and leave CF in a synchronized way.
2. Weak Sequencing (**WS**):
   Every lifeline is allowed to enter or leave CF as soon as it is done with its preceding events.

4.3 [combine].SYNC vs. [combine].WS



When l2.a = 0:

Only possible sequence of events for **SYNC** is
!m1, ?m1, !m3, ?m3

Whereas in **WS**, there are more sequences of events including
!m3, !m1, ?m3, ?m1

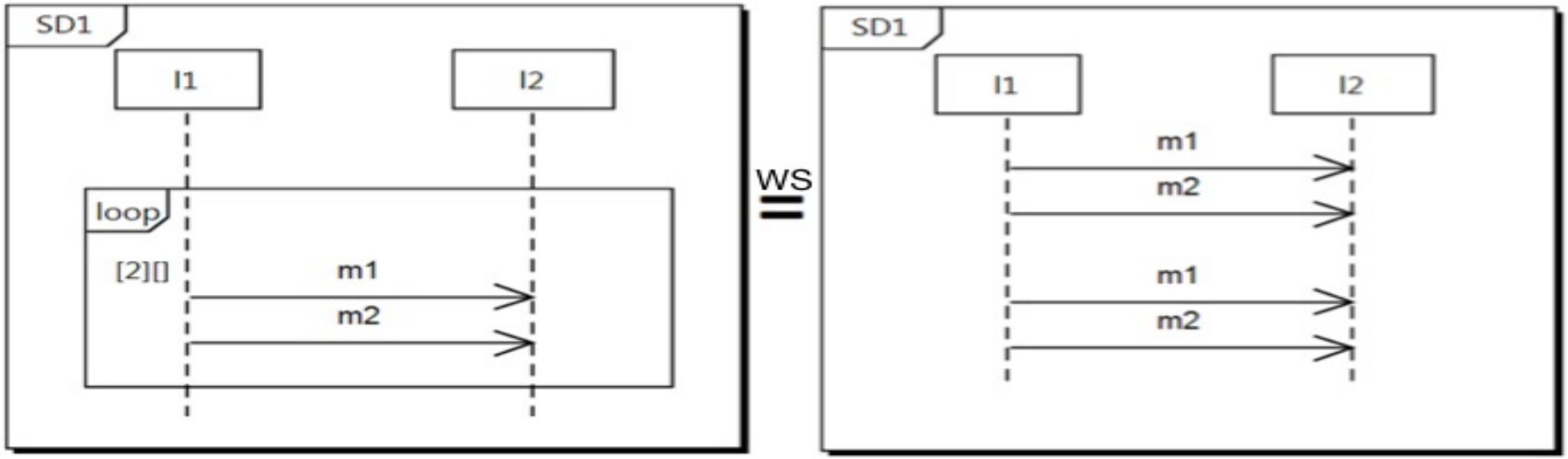
4.4 Semantic Variation Points [Loop]

[Loop]: How to combine *Loop* with the rest of SD?

1. Synchronized (**SYNC**):
   All iterations are strictly separated.

2. Weak Sequencing (**WS**):
   Events in different iterations can be interleaved.

4.5 [Loop].SYNC vs. [Loop].WS



SYNC: {<!m1,!m2,?m1,?m2,!m1,!m2,?m1,?m2>, …}

WS:
{<!$m1_{lt1}$,!$m2_{lt1}$,!$m1_{lt2}$,!$m2_{lt2}$,?$m1_{lt1}$,?$m2_{lt1}$,?$m1_{lt2}$,?$m2_{lt2}$>, …}

4.6 Semantic Variation Points [When]

[When]: When CF's guard(s) should be evaluated?

1. Single global time (**SGT**):
   When all lifelines are ready to enter CFx they evaluate the guard(s) in a synchronized manner. (consistent iff CFx.[combine] = SYNC)

2. Independent local times (**ILT**):
   The first lifeline that reaches to CFx evaluates the guard(s), and forces other lifelines to follow its footstep. (consistent iff CFx.[combine] = WS)

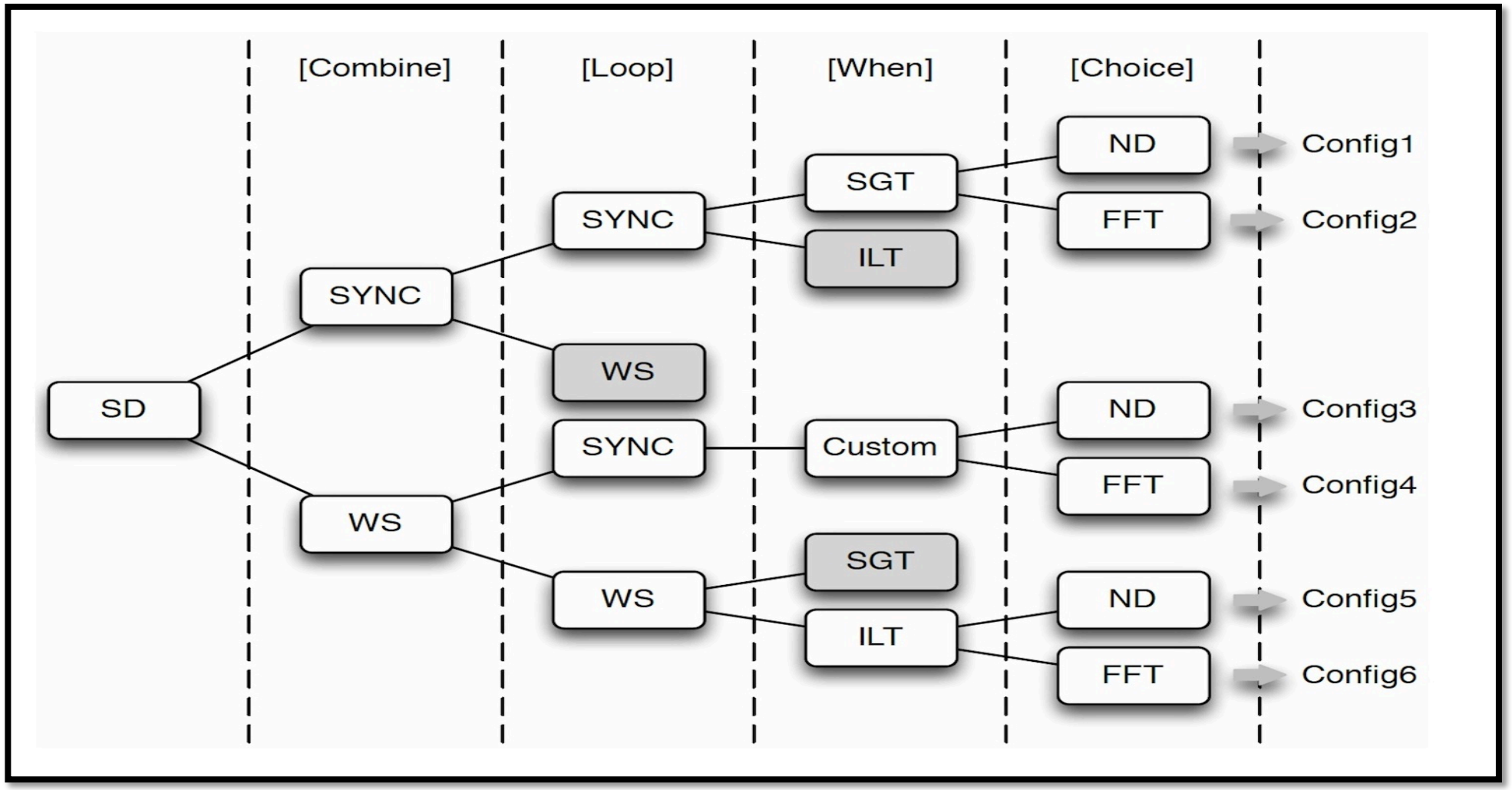
4.7 Semantic Variation Points [choice]

[Choice]: Which operand should be chosen?

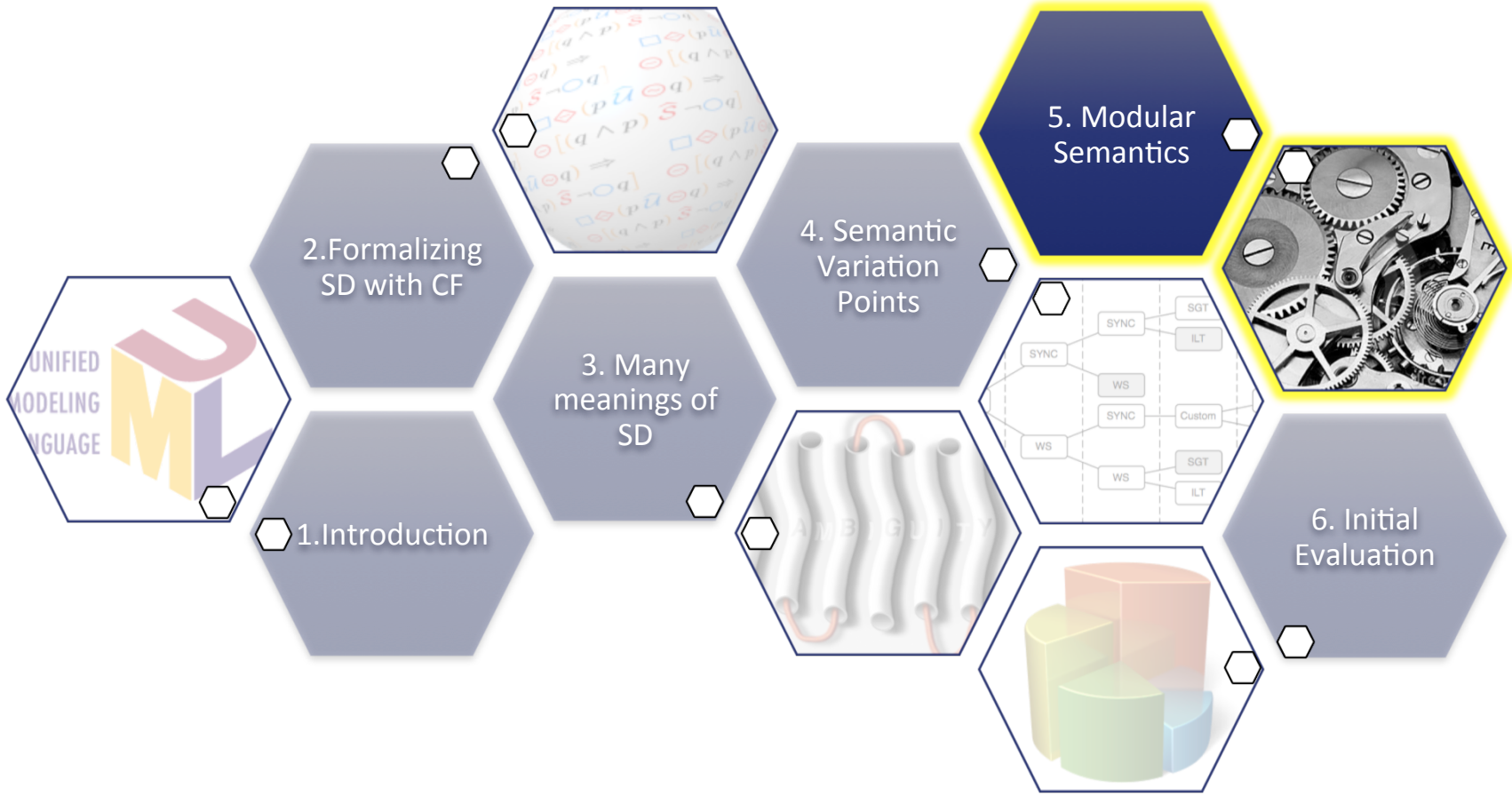
In **Alternative** CF if there are several true guarded operands, there are two options to choose one operand among them.

1. Nondeterministic (**ND**):
   One of true guarded operand is chosen nondeterministically.
2. First from top (**FFT**):
   First true guarded operand from top of Alternative CF is chosen.

4.8 Six Semantic Variants



5. Modular Semantics



5.1 Modular Semantics

- Ordering semantics:
  - **OrderMonoD(ev1, ev2, guard, exception, isConcur)**
  - **OrderMonoDRev(ev1, ev2, guard, exception, isConcur)**
  - **Order(ev1, ev2, guard, exception, isConcur)**
- Borders(Module, exception)
- Auxiliary Operators
  - $SomFIn_i(ev_1,$ enclosingModule)
  - $SomPIn_i(ev_1,$ enclosingModule)
- Combined Fragments
  - **AltCF(Alt, config)**
  - OptCF(Opt, config)
  - ParallelCF(Parallel, config)
  - LoopCF(Loop, config)
- Link_Pre_Post(CF, config)
- Combine(Module, config)

5.2 Mono-Directional Order

If the guard holds at the moment ev1 occurs, ev2 will happen. But if we have ev2 now, it does not necessarily mean that we had ev1 earlier.

It is used when ev2 has several triggers including ev1.

$$\mathbf{OrderMonoD}(ev_1, ev_2, \text{guard}, \text{exception}, \text{isConcurrent}) \stackrel{\text{def}}{=}$$
$$\begin{cases} (1) & \text{if isConcurrent} = true \\ (1) \wedge (2) & \text{otherwise} \end{cases}$$

$$ev_1 \wedge \text{guard} \Rightarrow \begin{pmatrix} \text{Until}_{\text{ei}}((\neg ev_1 \wedge \neg ev_2), \text{exception}) \ \vee \\ \text{Until}_{\text{ei}}((\neg ev_1 \wedge \neg \text{exception}), ev_2) \end{pmatrix}. \quad (1)$$

$$ev_1 \wedge \text{guard} \Rightarrow \neg ev_2. \quad (2)$$

5.3 Mono-Directional Order Reverse

ev2 is preceded by true guarded ev1. But having true guarded ev1 does not necessarily mean that ev2 will occur.
It is used when ev2 is one of possible consequences of true guarded ev1.

$$\mathbf{OrderMonoDRev}(ev_1, ev_2, \text{guard}, \text{exception}, \text{isConcurrent}) \stackrel{\text{def}}{=}$$

$$\begin{cases} (1) & \text{if isConcurrent} = true \\ (1) \wedge (2) & \text{otherwise} \end{cases}$$

$$ev_2 \Rightarrow \text{Since}_{\text{ei}}((\neg ev_2 \wedge \neg \text{exception}), (ev_1 \wedge \text{guard})) \,. \qquad (1)$$

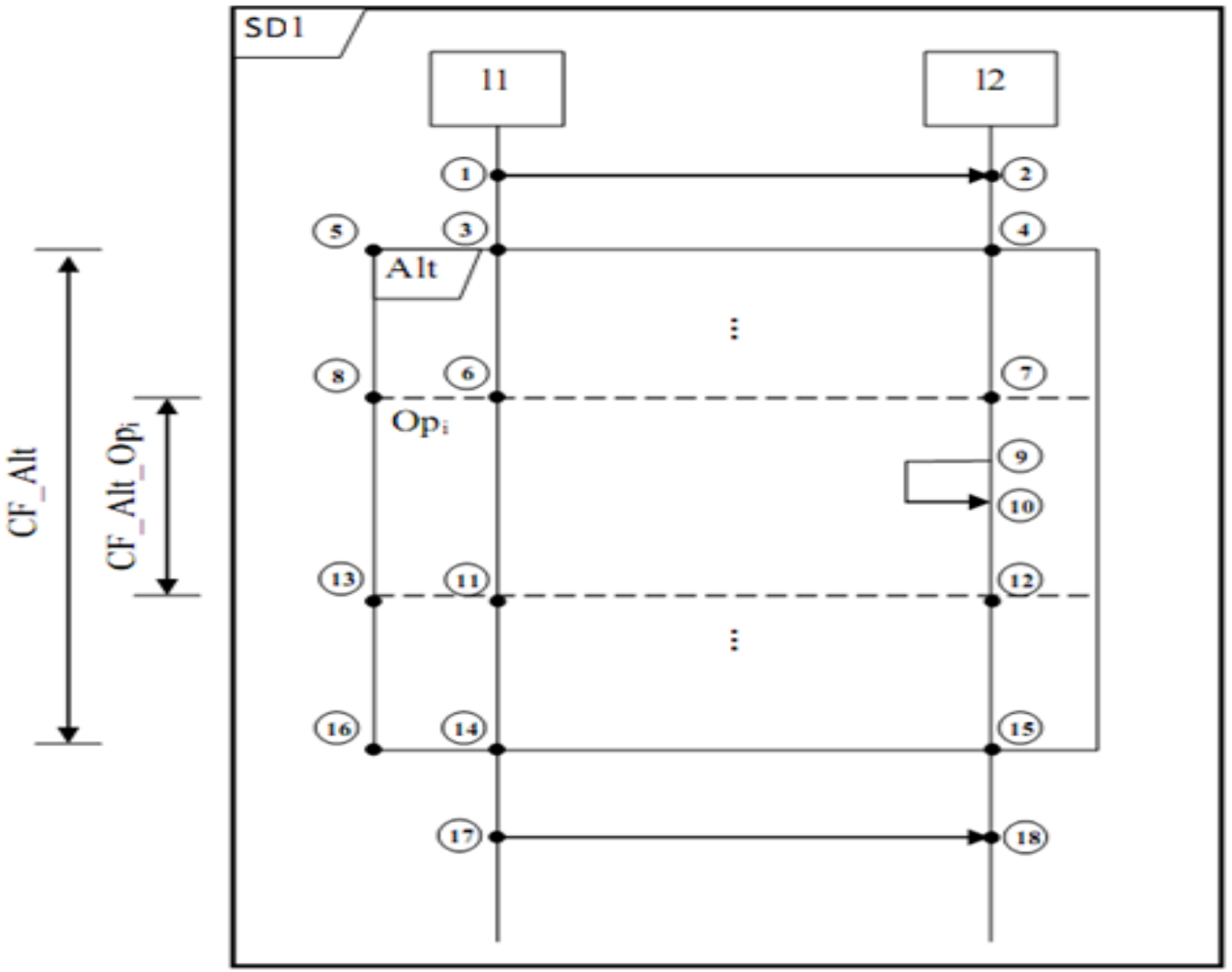
$$ev_1 \wedge \text{guard} \Rightarrow \neg ev_2. \qquad (2)$$

5.4 Bidirectional Order

If true guarded ev1 occurs, ev2 will happen and if ev2 occurs, it means that we had true guarded ev1.

$$\begin{aligned}&\mathbf{Order}(ev_1, ev_2, \text{guard}, \text{exception}, \text{isConcur}) \stackrel{\text{def}}{=} \\ &\quad \mathbf{OrderMonoD}(ev_1, ev_2, \text{guard}, \text{exception}, \text{isConcur}) \wedge \\ &\quad \mathbf{OrderMonoDRev}(ev_1, ev_2, \text{guard}, \text{exception}, \text{isConcur})\end{aligned}$$

5.5 Mapping UML Elements to Atomic Propositions



3. Alt_l1_Start
4. Alt_l2_Start
5. Alt_Start
6. Alt_Op$_i$_l1_Start
7. Alt_Op$_i$_l2_Start
8. Alt_Op$_i$_Start
11. Alt_Op$_i$_l1_End
12. Alt_Op$_i$_l2_End
13. Alt_Op$_i$_End
14. Alt_l1_End
15. Alt_l2_End
16. Alt_End

5.6 AltCF (WS, ND) vs. AltCF (WS, FFT)

AltCF(WS, FFT):

$$\mathbf{Borders}(Alt, S_{Stop}) \tag{6}$$

$$(\bigvee_{i=1}^{n} Alt\_L_{Start}^{i} \Rightarrow Alt) \wedge (Alt_{Start} \Rightarrow \bigvee_{i=1}^{n} Alt\_L_{Start}^{i}) \tag{7}$$

$$\bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n} \mathbf{Order}(Alt\_L_{Start}^{j}, Alt\_OP^{i}\_L_{Start}^{j}, Alt\_OP^{i}, S_{Stop}, true) \tag{8}$$

$$\bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n} \mathbf{OrderMonoD}(Alt\_OP^{i}\_L_{End}^{j}, Alt\_L_{End}^{j}, true, S_{Stop}, true) \tag{9}$$

$$Alt\_OP_{Start}^{Else} \Leftrightarrow (\neg(\bigvee_{i=1}^{m} Alt\_Guard^{i}) \wedge Alt_{Start}) \tag{10}$$

$$\bigwedge_{i=1}^{m} Alt\_OP_{Start}^{i} \Leftrightarrow Alt_{Start} \wedge Alt\_Guard^{i} \wedge \neg \bigvee_{j=1}^{i-1} Alt\_Guard^{j} \tag{14}$$

$$\bigvee_{i \in [1,m] \cup Else} Alt\_OP_{End}^{i} \Leftrightarrow Alt_{End} \tag{12}$$

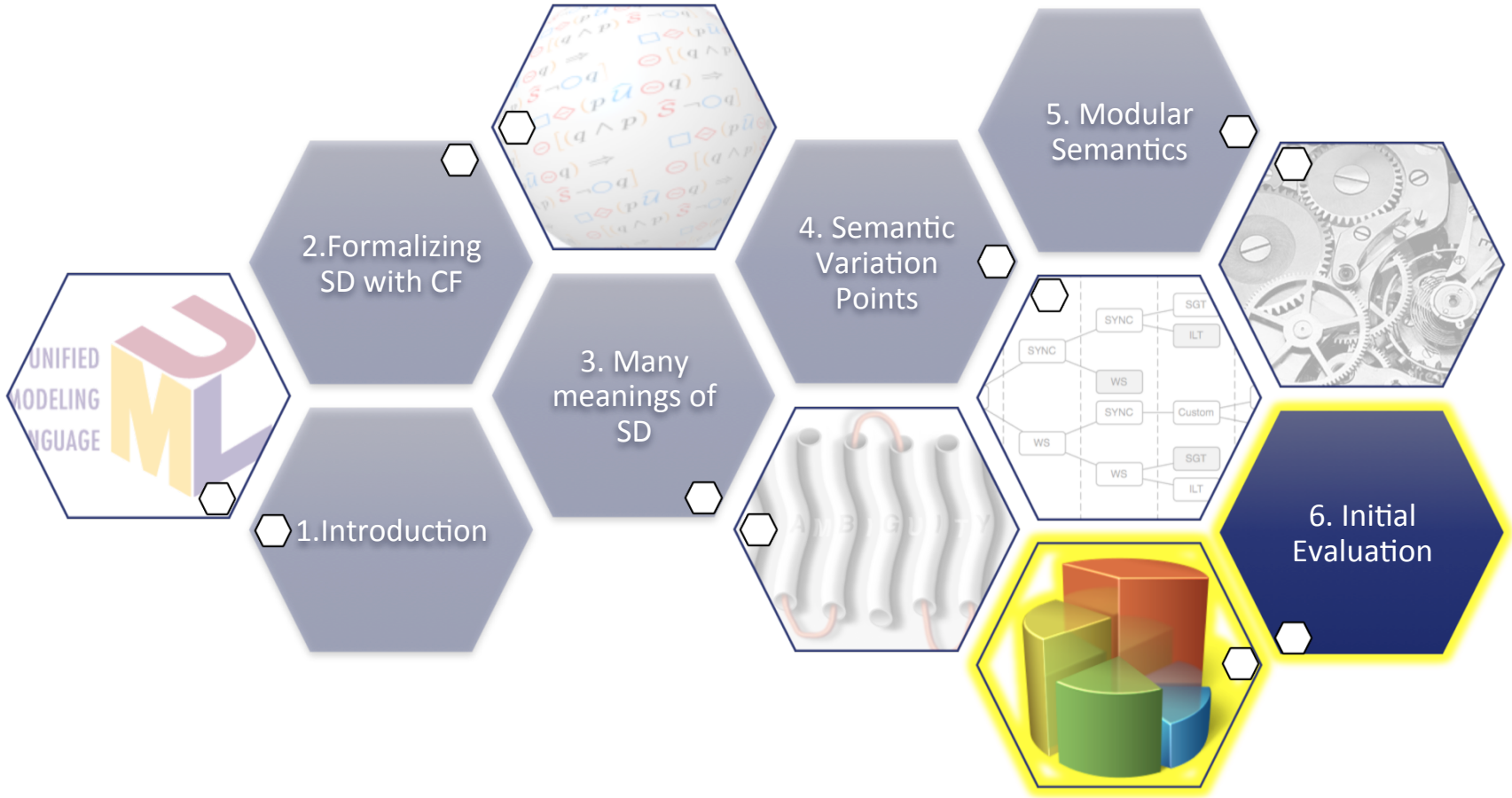
$$\bigwedge_{i=1}^{m} (Alt\_OP_{Start}^{i} \Rightarrow Alt_{Start} \wedge Alt\_Guard^{i}) \tag{13}$$

5.7 Alternative CF (SYNC, ND)

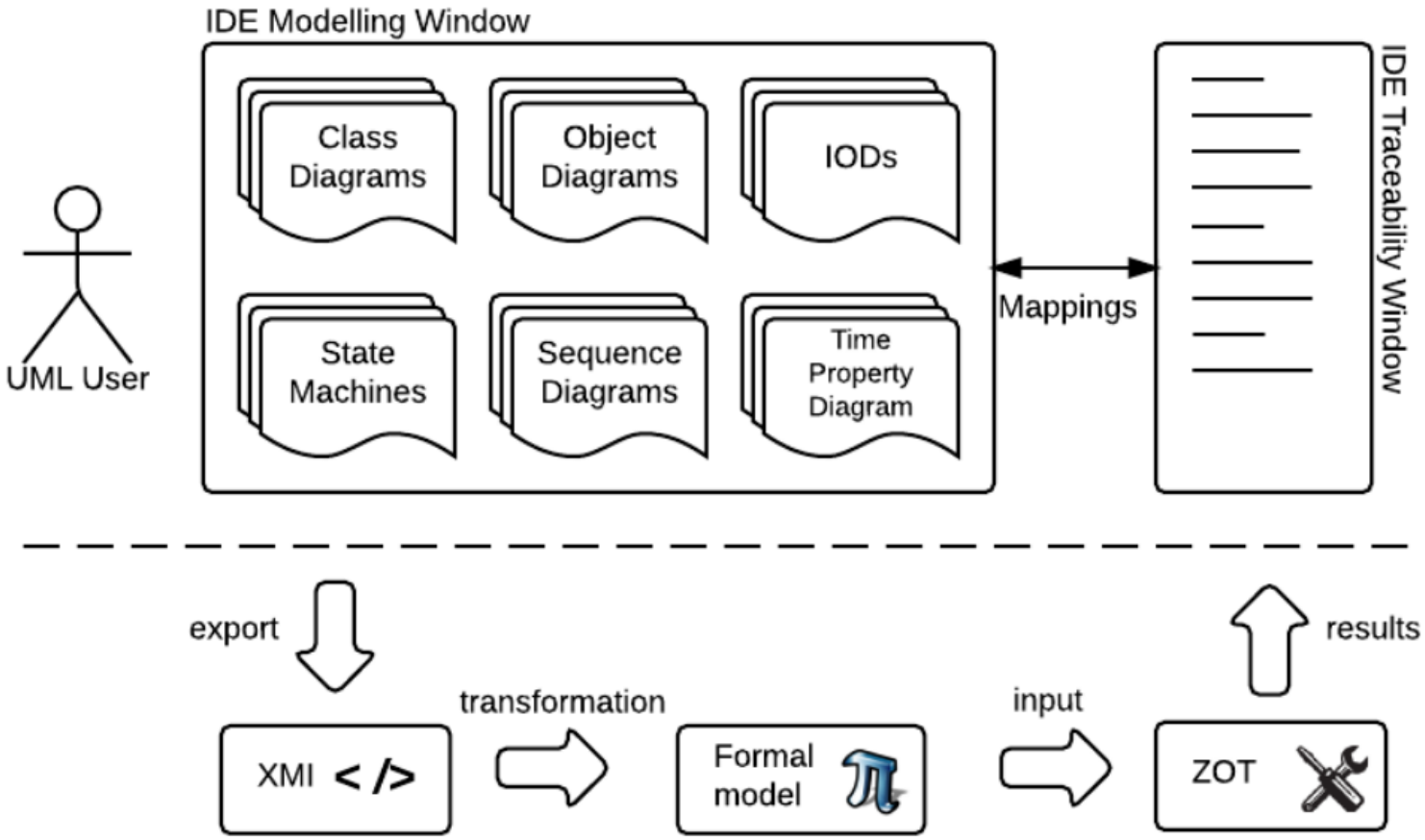
If the choice of **[Combine]** is **SYNC** instead of **WS**, the positive traces are a subset of those allowed by **WS**, since there are additional constraints on the start and end of the combined fragment. More precisely, in this case the following formula is added:

$$(Alt_{Start} \Leftrightarrow \bigwedge_{i=1}^{n} Alt\_L^{i}_{Start}) \wedge (Alt_{End} \Leftrightarrow \bigwedge_{i=1}^{n} Alt\_L^{i}_{End}) \quad (15)$$

6. Initial Evaluation



6.1 Our Verification Tool (Corretto)



https://github.com/mmpourhashem/CorrettoUML

6.2 SDSearch

We extended our verification tool, ***Corretto***, to support SD with CF, and performed some experiments regarding History Checking and Completion (HCC), property checking, and time constraints on this SD.

HCC:

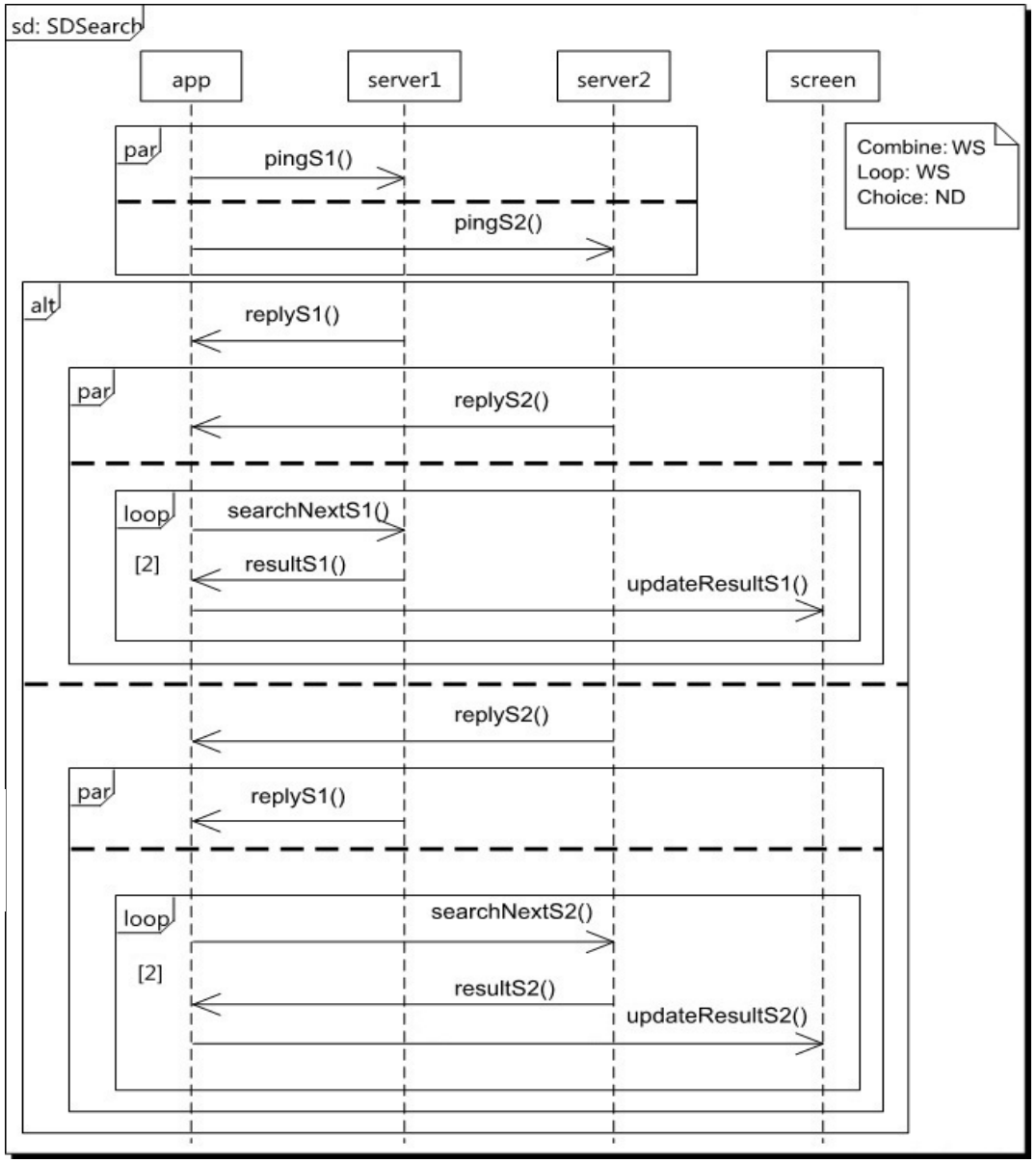
T1 = {⟨!?pingS1, !pingS2⟩, ⟨!replyS1⟩, ⟨?pingS2⟩}

Property:

$$P_1 \stackrel{\text{def}}{=} \text{Alw}\left(\begin{array}{l}\text{replyS1}_{Start} \wedge \text{Futr}(\text{replyS2}_{Start}, 1) \Rightarrow \\ \quad \neg\text{SomFIn}_{\text{i}}(\text{searchNextS2}_{Start}, \text{SDSearch})\end{array}\right)$$

Time Constraints:

$$(@updateResultS1_{End} - @updateResultS1_{Start}) \geq 1$$
$$(@updateResultS2_{End} - @updateResultS2_{Start}) \geq 1$$
$$(@SDSearch_{End} - @SDsearch_{Start}) \leq 8$$



6.3 Initial Evaluation

		S-S-ND	S-S-FFT	W-S-ND	W-S-FFT	W-W-ND	W-W-FFT
T1	Result	UNSAT	UNSAT	SAT	SAT	SAT	SAT
	Time -s	6	6	31	27	40	33
T2	Result	UNSAT	UNSAT	UNSAT	UNSAT	SAT	SAT
	Time -s	6	6	15	14	40	38
P1	Result	PASSED	PASSED	FAILED	PASSED	FAILED	PASSED
	Time -s	14	11	41	36	64	46
P2	Result	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED
	Time -s	13	11	57	37	53	41
TC	Result	UNSAT	UNSAT	UNSAT	UNSAT	SAT	SAT
	Time -s	12	9	30	24	42	34

7. Conclusion and Future Work

- We presented a modular semantics for UML SD that accommodates existing interpretations.
- First evaluation of the six possible configurations we have identified through Corretto is presented.
- The plan for the future is to cover the whole set of combined fragments.

- Coming soon …

encoding for Bounded Satisfiability Checking of LTL formula (and also MTL and CLTLB with bounded variables), which is both faster and more memory efficient, i.e. it is able to check bigger models (and **100 times faster** for some models).

Questions?

pourhashem.kallehbasti@elet.polimi.it