Correct Composition in the Presence of Behavioural Conflicts and Dephasing[☆]

Juliana K. F. Bowles[a,*], Marco B. Caminati[a]

[a]*School of Computer Science, University of St Andrews, Jack Cole Building St Andrews KY16 9SX, United Kingdom*

Abstract

Scenarios of execution are commonly used to specify partial behaviour and interactions between different objects and components in a system. To avoid overall inconsistency in specifications, various automated methods have emerged in the literature to compose scenario-based models. In recent work, we have shown how the theorem prover Isabelle/HOL can be combined with an SMT solver to detect inconsistencies between sequence diagrams and, only in their absence, generate the behavioural composition. In this paper, we exploit this combination further and present an efficient approach that generates all valid composed traces giving us an equivalent representation of the conflict-free valid composed model. In addition, we show a novel way to prove the correctness of the computed results, and compare this method with the implementation and verification done within Isabelle alone. To reduce the complexity of our technique, we consider priority constraints and a notion of dephased models, i.e., models which start execution at different times. This work has been inspired by a problem from a medical domain where different clinical guidelines for chronic conditions may be applied to the same patient at different points in time. We illustrate the approach with a realistic example from this domain.

Keywords: Formal methods, Verification, SMT solver, Theorem prover, Isabelle/HOL, Event structures, Model Composition, Optimisation

1. Introduction

To cope with the complexity of modern systems, design approaches combine a variety of languages and notations to capture different aspects of a system, and separate structural from behavioural models. In itself behavioural modelling is

[☆]This research is supported by grants: EPSRC EP/M014290/1 and MRC MR/S003819/1.

[*]Corresponding author

Email addresses: jkfb@st-andrews.ac.uk (Juliana K. F. Bowles), mbc8@st-andrews.ac.uk (Marco B. Caminati)

URL: http://orcid.org/0000-0002-5918-9114 (Juliana K. F. Bowles), http://orcid.org/0000-0002-4529-5442 (Marco B. Caminati)

challenging, and rather than attempt to model the complete behaviour of a (sub)system [44], it is easier to focus on several possible scenarios of execution separately. Scenarios give a partial understanding of a component and include interactions with other system components. In industry, individual scenarios are often captured using UML's sequence diagrams [37]. Given a set of scenarios, we want to obtain their combined behaviour, and incrementally continue to enrich the overall behavioural model as more scenarios are identified. This requires a mechanism to compose scenario-based models, and when this cannot be done because the scenarios contain inconsistencies, we want to detect and remove them to keep the overall composition model valid. Our approach allows us to gain further understanding on the system behaviour, even when the complete and overall behaviour is unknown.

It is widely recognised that composing systems manually can only be done for small systems. As a result, in recent years, various methods for automated model composition have been introduced [1, 14, 41, 30, 33, 43, 45, 46, 50, 13, 15, 16]. Most of these methods involve introducing algorithms to produce a composite model from simpler models originating from partial specifications and assume a formal underlying semantics [30]. In our recent work [13, 15, 16], we have used constraint solvers for automatically constructing the composed model. This involves generating all constraints associated to the models, and using an automated solver to find a solution (the composed model) for the conjunction of all constraints. We used the SAT solver underlying Alloy [28] in [13, 15] and the SMT solver Z3 [34] in [16]. SAT-solvers have been used to merge different versions of a sequence diagram in accordance to the system behaviour given by a state machine in [46]. More generally, conflict detection algorithms and model merging strategies have been actively researched (cf. [21] for a review) in order to address problems inherent in model versioning systems. Even though some of the existing approaches reduce the problem to a constraint satisfaction (usually SAT) problem, the context and assumptions taken are very different from ours.

We have conducted several experiments showing that Z3 performs much better than the Alloy analyzer for large systems [16]. Using Alloy and its underlying SAT solver, for model composition, mostly in the context of structural models, is very common (e.g., [43, 50]), but the use of an SMT solver Z3 in the same context is a novelty of [16]. Even though we used Z3 in [16], we did not explore Z3's arithmetic capabilities, nor did we deal with incompatible constraints. We have addressed both points recently in [17] as well as in the present paper.

As in our earlier work, our approach in [17] used event structures [49] as an underlying semantics for sequence diagrams in accordance to [32, 12], and explored how the theorem prover Isabelle/HOL [36] and constraint solver Z3 [34] could be combined to detect and solve partial specifications and inconsistencies over event structures. In this paper, we go one step further in improving the process of automatically generating correct composition models for behavioural models that may contain inconsistencies. To do so, we take the view that a (composed) behavioural model corresponds to the set of valid traces of execution for that model. To compose two or more behavioural models, we can generate all possible traces of execution that respect any constraints for the composition

or individual models. If we can guarantee that the generated traces are valid and the overall set of traces is complete, then we have obtained an equivalent representation of the valid behaviour of the original composed model. Any inconsistencies in the composed model are excluded (i.e., no inconsistent traces are considered).

Generating *all valid* traces for a composition of behavioural models can quickly become very complex, even when considering further constraints that restrict the number of possible traces. Our approach allows the generation of possible combined traces by prioritising the generation of preferred traces first. A further novelty of our approach, which helps reduce the complexity of the composed model, is the introduction of a notion of *dephased* models prior to composition. This makes it possible to combine models which do not start execution simultaneously, where events are executed at a different *pace*, and where alternative events are given a different *priority*. The effect is a reduction of detected inconsistencies (if any), and the automated generation of what are valid context-specific traces of execution. This work has been inspired by a problem from a medical domain where different clinical pathways for chronic conditions are applied to the same patient with different starting points (diagnosis).

Our emphasis in this paper is the composition of two or more scenario-based models and the behaviour that can be obtained from that composition even in the presence of inconsistencies. If we disregard the traces from the scenario-based models that lead to overall inconsistencies, we can obtain a composed model. Approaches developed for the integration of multiple versions of a scenario such as [46, 29] work under the assumption of an existing state-based system model (a state machine) which can be used to detect inconsistencies of different versions of a sequence diagram. This is different in our case, as we do not have a reference system model and the inconsistencies in our approach are between (elements within) the scenarios themselves. We argue that this can be useful because a complete state-based behavioural model may not exist.

This paper extends our work presented in [18] by showing more generally how to generate automatically all valid traces for composed models (with or without dephasing) using SMT solvers. In the presence of inconsistencies between different behavioural models, we are interested in the composed model that can be obtained after removing such inconsistencies. By generating the set of all possible combined valid traces, we obtain an equivalent representation of the behaviour of the composed model as intended. We use Isabelle both to prove the validity of a trace and the completeness of the set of generated traces. In addition to these proofs under a given bound, we also present an original, general method to provide a formal correctness proof for SMT code.

This paper is structured as follows. The motivation and contributions of the work presented here are discussed in Section 2, while in Section 3 we recall our formal model (labelled event structures) used to provide a semantics to scenario-based models given by sequence diagrams. Section 4 translates the formal model into computable Isabelle functions and provides formal correctness theorems about them, while Section 5 introduces SMT code to compute the underlying traces of execution. Section 6 describes how Isabelle and the SMT solver Z3 are

combined to formally prove correctness theorems applying to the code. Section 7 uses the verified SMT code to obtain a complete set of execution traces and hence a description of the composition of the given models. In Section 8, we discuss some performance optimisations for the SMT code, and discuss how they impact on the validity of the formal correctness proofs obtained and add further proofs. We introduce an example from the clinical domain to illustrate the applicability and power of our approach in Section 9. We conclude the paper with a description of related work in Section 10, and a discussion of future work in Section 11.

2. Context and Contribution

Continuing the work started in [17], we exploit the interface between Isabelle and Z3 to obtain a versatile tool for the specification, analysis and computation of the behaviour of complex systems. Representing labelled event structures, our underlying semantic model for UML sequence diagrams (cf. [32, 12]), directly in Isabelle means that we can check automatically the correctness of the diagrams, obtain their composition (if it exists) with Z3 and fill any gaps, while being able to prove at any point that the diagrams are valid [17]. If our diagrams contain inconsistent behaviour, we use Z3 to locate the reason for this. However, we argue in this paper, that not generating a composition model because of the presence of conflicts is too restrictive. Instead, we are interested in obtaining a valid behavioural model for the composition after discarding all inconsistencies. Further, we may be able to avoid (some) inconsistencies further if we allow models to be *dephased*, that is, we allow scenarios to start execution at different times and continue execution at a different pace. We also consider a notion of priority in a model. We develop a technique to automatically generate all valid traces by defining exactly how the different scenarios come together (i.e., how they are dephased) and which traces are closer to satisfying assumed model priorities. This allows us to generate all preferred solutions first, successively generate all valid traces, and obtain the overall valid composed behaviour given through its set of traces of execution. To keep the approach relevant in practice we have chosen a healthcare inspired problem.

Consider the problem of caring for patients with *multimorbidities*, i.e., patients with two or more chronic conditions. Clinical guidelines describe how to care for a patient with one concrete chronic condition, but usually ignore the presence of several ongoing conditions. It is, however, increasingly common for people to develop two or more chronic conditions over time. In Scotland, over half of all people with chronic conditions have two or more conditions simultaneously [39]. When different clinical guidelines for chronic conditions are being applied to the same patient:

- different steps may be executed at a different *pace*. For instance, for one condition we may need observations to be carried out every month, whereas for others every three months is sufficient.

- one of the conditions may be prevalent and for this reason has higher *priority*.
- some of the possible medications prescribed at a given *step* in the guidelines may have higher *priority* due to better treatment effectiveness. For instance, the use of metformin in the treatment of type2 diabetes.
- the diagnosis of different conditions for a patient are likely to have occurred at different times. For instance, the diagnosis of chronic kidney disease often follows (and may be a consequence of) an earlier diagnosis of type 2 diabetes. This leads to the corresponding care guidelines starting execution at different times, in other words, their execution is *dephased*.

In particular, having an automated technique that allows us to find valid combined traces taking into account priorities is useful as it gives us a flexible mechanism to identify *different solutions* in similar but different cases. For instance, patients with the same conditions overall but with different orders of diagnosis, priorities or prevalent condition. To keep the presentation of this paper more focused, we omit the medical details and instead show how the approach works for an abstract example. A clinical example is introduced later in Section 9. Consider the following example of UML sequence diagrams [37].

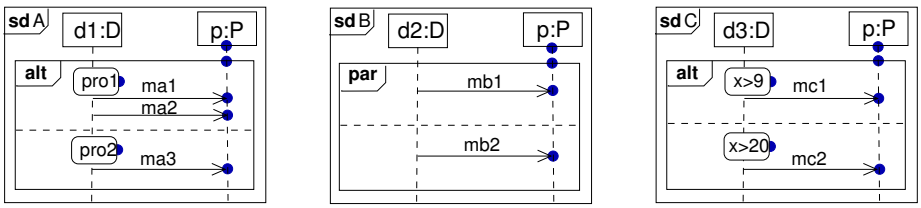

Figure 1: Three scenarios involving the same object instances.

Figure 1 shows three scenarios involving the same instance `p` and different instances of the same class `D`, that is, `d1`, `d2`, `d3`. The scenarios use interaction fragments for *alternative* behaviour (indicated by an `alt` on the top-left corner) and *parallel* behaviour (indicated by a `par` on the top-left corner). Other fragment operators exist but are not used in this paper (cf. [37] for details).

Interaction fragments, such as `alt` and `par` amongst others, contain one or more operands, which in the case of an alternative may be preceded by a constraint or guard. All diagrams in Figure 1 have two operands each separated by a dashed line. What happens to the behaviour contained within an operand depends on the interaction fragment used. The alternative fragment in `sdA` uses two constraints for the operands, namely `pro1` and `pro2`, and we note that they are not necessarily mutually exclusive. We may want to associate a priority to `pro1`, to indicate for instance that if it holds we will want the corresponding operand to execute (instead of the second operand and regardless of whether `pro2` holds or not). UML does not have direct notation to indicate this, but we can assume the existence of a priority tag (not shown) and add a priority notion

to our formal model. For the messages shown (for instance, `ma1`, `mb1`, `mc1`, and so on), we assume that when they are received, they imply an occurrence for instance `p`. The marked points along the lifeline of instance `p` and next to the conditions are what we call *locations*, borrowing terminology from Live Sequence Charts (LSCs) [26]. They do not serve a purpose at the design level but make it easier to understand the formal semantics (cf. [32] for details).

Assume that we know that the occurrence of `ma1` conflicts with `mc1`, and `ma2` conflicts with `mb2`. This is not encoded directly in the scenarios above, but is domain knowledge contained elsewhere. For instance, in a medical context it is known that certain combinations of drugs when given together cause adverse reactions and should hence not be given to a patient at the same time. Furthermore, we may wish to distinguish the severity of such conflicts either by categorising it (mild, moderate, severe) or giving it an integer value.

We now want to obtain the composition of these three diagrams in such a way that the known underlying conflicts between some of the labels are taken into account. It seems clear that, focusing only on instance `p`, to avoid these conflicts the easiest thing to do is to take the second alternative in `sdA` assuming that `pro2` holds. No conflict is present in that case. However, it may be the case that `pro1` holds as well and it has an associated higher priority leading to the execution of `ma1` followed by `ma2`. The question is whether we can still obtain a valid trace that includes this preference and avoids the known conflicts. Our approach developed here gives an answer to this question under the assumption that simultaneous occurrence of conflicting messages is avoided. Notions of current state, pace and occurrence priority are introduced and used as parameters to find valid traces in a composed model. We describe how these are treated formally in the next sections.

In this paper, we focus on the formal semantics, the composition and valid traces defined at that level, and the formal methods used to detect them. We do not come back to a design level, but we assume the underlying formal models used here have been generated from scenarios or process descriptions. See our earlier work for an idea of the transformation defined at the metamodel level [13, 15, 16]. See [31] for a description of the medical problem of treating patients with multimorbidities. We will return to a medical example in Section 9.

3. Formal Model

The model we use to capture the semantics of a sequence diagram is a labelled (prime) event structure [49], or *event structure* for short. Event structures have been widely used and studied in the literature, and have been used to give a true-concurrent semantics to process calculi such as CCS, CSP, SCCS and ACP (e.g., [48]). The advantages of prime event structures include their underlying simplicity and how they naturally describe fundamental notions present in behavioural models including sequential, parallel and iterative behaviour (or the unfoldings thereof) as well as nondeterminism (cf. [32, 12]), and are hence our model of choice. Event structures have well-defined composition operators (cf. e.g., [35]). However, these composition mechanisms ignore labels and are

hence inadequate for our use here. In [17], we developed an automated approach in that can detect the conflicts in the scenarios as described in Section 2, formalised as event structures and given additional constraints on label conflicts. We now extend our approach to find all valid paths that avoid these label-induced conflicts. We describe the formal model first.

In an event structure, we have a set of event occurrences together with binary relations for expressing causal dependency (called *causality*) and nondeterminism (called *conflict*). The causality relation implies a (partial) order among event occurrences, while the conflict relation expresses how the occurrence of certain events excludes the occurrence of others. From the two relations defined over the set of events, a further relation is derived, namely the *concurrency* relation *co*. Two events are concurrent if and only if they are completely unrelated, i.e., neither related by causality nor by conflict.

To see how these relations can be associated to notions within a sequence diagram, recall the diagram `sdA` in Figure 1. The locations marked along the lifeline of an instance will typically correspond to events. For example, the location marking the receipt of message `ma2` can correspond to an event e_4. According to the diagram, the occurrence of e_4 has to be preceded by event e_2 associated to the location marking the receipt of message `ma1`. This means that these events are related by causality. Furthermore, the occurrence of event e_3 associated to the location marking the receipt of message `ma3` is in conflict with both e_2 and e_4, since these events correspond to different (mutually exclusive) operands in the `alt` interaction fragment. In the same figure, diagram `sdB` shows a `par` interaction fragment instead which means that events associated to locations in different operands should be in concurrency. This is the case for event g_2 associated to the receipt of message `mb1` and event g_3 associated to the parallel receipt of message `mb2`.

The formal definition of an event structure, as provided for instance in [32], is as follows.

Definition 1. *An* event structure *is a triple* $E = (Ev, \rightarrow^*, \#)$ *where* Ev *is a set of events and* $\rightarrow^*, \# \subseteq Ev \times Ev$ *are binary relations called* causality *and* conflict*, respectively. Causality* $\rightarrow^*$ *is a partial order. Conflict* $\#$ *is symmetric and irreflexive, and propagates over causality, i.e.,* $e\#e' \wedge e' \rightarrow^* e'' \Rightarrow e\#e''$ *for all* $e, e', e'' \in Ev$*. Two events* $e, e' \in Ev$ *are* concurrent*,* e *co* e' *iff* $\neg(e \rightarrow^* e' \vee e' \rightarrow^* e \vee e\#e')$*.* $C \subseteq Ev$ *is a* configuration *iff (1)* C *is* conflict-free*:* $\forall e, e' \in C \neg(e\#e')$ *and (2)* downward-closed*:* $e \in C$ *and* $e' \rightarrow^* e$ *implies* $e' \in C$*.*

We assume *discrete* event structures. Discreteness imposes a finiteness constraint on the model, i.e., there are always only a finite number of causally related predecessors to an event, known as the *local configuration* of the event (written $\downarrow e$). A further motivation for this constraint is given by the fact that every execution has a starting point or configuration. A *trace of execution* in an event structure is a maximal configuration. An event e may have an immediate successor e' according to the order $\rightarrow^*$: in this case, we will usually write $e \rightarrow e'$. The relation given by $\rightarrow$ is called *immediate causality*. An event

e within a configuration C is *maximal* iff there are no other events in C that are successors of e, i.e., for all $e^{'} \in C$ if $e \rightarrow^{*} e^{'}$ then $e = e^{'}$.

To make a connection between the semantic model (here an event structure) and the syntactic model (e.g., sequence diagram) it is describing, we need to associate some additional information to individual events. Let L be a given set of labels.

Definition 2. *A* labelled event structure *over* L *is a triple* $M = (Ev, \mu, \nu)$ *where* μ *and* ν *are partial labelling functions* $\mu : Ev \rightarrow 2^{L}$ *and* $\nu : Ev \rightarrow \mathbb{N} \times \mathbb{N}$.

Labelled event structures are event structures enriched with two labelling functions μ and ν. The function μ maps events onto a subset of elements of L. The labels in the set L either denote formulas (constraints over integer variables, e.g., $x > 9$ or $y = 5$), logical propositions (e.g., `pro1`) or actions (e.g., $ma1$). If for an event $e \in Ev$, $\mu(e)$ contains an action $\alpha \in L$, then e denotes an occurrence of that action α. If $\mu(e)$ contains a formula or logical proposition $\varphi \in L$, then φ must hold when e occurs.

The labelling function ν associates to each event its *priority* and *duration*, for instance, $\nu(e) = (p, d)$ indicates that p is the priority and d is the duration associated with e. The higher the value of p, the higher the priority associated to the event. The duration d indicates the time units spent at event e. Giving different priority values to events is meaningful in the presence of alternatives (conflicting events), where the highest value can be used to determine the ideal configuration in a model. We sometimes use $\nu^{(1)}$ to indicate the priority function and $\nu^{(2)}$ the duration function, that is, $\nu^{(i)} = \text{proj}_i \circ \nu$, where proj_i is the i-th Cartesian projector map (i.e., the map extracting the i-th component of a tuple). Further labels may be added to the framework as partial functions if required. We call a labelled event structure a model in what follows.

In what follows assume a finite number of models $M_1, \ldots, M_n$ where $n \in \mathbb{N}$, in accordance with Definition 2. We define a map Γ specifying the level of conflict between event labels across models as follows.

Definition 3. *Label conflicts are given by* $\Gamma \subseteq L_i \times L_j \times \mathbb{Z}$ *where* $i, j \in [1..n]$.

Here, we assume binary conflicts of a certain value. For instance, (l_1, l_2, v) indicates that l_1 and l_2 are in conflict with an *interaction score* of value v. We consider that the lower the value of v the higher the severity of the label conflict.

Let us return to the use of labelled event structures as a formal model for behavioural models such as sequence diagrams. We do not show here how to generate an event structure from a sequence diagram, but give the general idea. As described briefly earlier, the locations along the lifelines of sequence diagrams are associated to one or more events. Locations within different operands of an alternative fragment correspond to events in conflict, whereas locations within operands of a parallel fragment correspond to concurrent events. The events associated to the locations along a lifeline are related by causality (partial order). For more details, c.f. [32].

Recall the example of Figure 1. The locations along the lifeline of instance `p` have been marked. The locations associated to the conditions/guards of the alternative fragments belong to the instances of class `D`, but that distinction is irrelevant for our purposes. Assume the label conflicts given by $\Gamma = \{(ma1, mc1, -200), (ma2, mb2, -100)\}$. The behaviour of `p` in the individual diagrams of Figure 1 is shown in the three event structures M_A, M_B and M_C of Figure 2, where the events are associated to the marked locations of the corresponding sequence diagram as expected. The defined labels are as follows: $\mu_A(e_2) = \{pro1, ma1\}$, $\mu_A(e_3) = \{pro2, ma3\}$, and $\mu_A(e_4) = \{ma2\}$ for the event structure associated to `sdA`; $\mu_B(g_2) = \{mb1\}$ and $\mu_B(g_3) = \{mb2\}$ associated to `sdB`; and $\mu_C(f_2) = \{x > 9, mc1\}$ and $\mu_C(f_3) = \{x > 20, mc3\}$ associated to `sdC`.

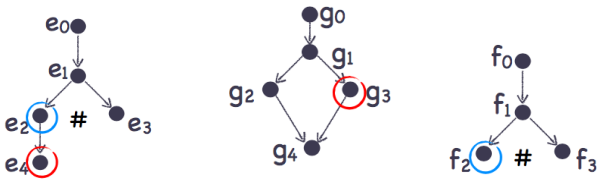

Figure 2: Corresponding event structures for instance `p`.

The labels of some of the events (marked) above are conflicting according to Γ, namely events e_2 and f_2, and events e_4 and g_3. When obtaining the composition of the models above we need to make sure label inconsistencies are detected and avoided. A composed model that avoids label conflicts could reduce the composition to the trace of execution $\tau_1 = \{e_0, e_1, e_3, g_0, g_1, g_2, g_3, g_4, f_0, f_1, f_3\}$ or τ_2 (identical to τ_1 except that it contains f_2 instead of f_3). Note that a trace of execution is a maximal configuration as introduced in Definition 1. However, these traces may not be the best with respect to the given priorities. The labels on events are only inconsistent if they occur simultaneously, and if we know where instance `p` is within each of the scenarios we may be able to avoid conflicts. Function ν gives us that information. Furthermore, if inconsistencies cannot be avoided we favour those with a higher interaction score (less severe).

Assume the following ν labels for some of the events in our example: $\nu_A(e_0) = \nu_B(g_0) = \nu_C(f_0) = (1, 1)$, $\nu_A(e_1) = \nu_B(g_1) = \nu_C(f_1) = (1, 1)$, $\nu_A(e_2) = (5, 3)$, $\nu_A(e_3) = (1, 3)$, $\nu_A(e_4) = (5, 2)$, $\nu_B(g_2) = (1, 2)$, $\nu_B(g_3) = (1, 1)$, $\nu_C(f_2) = (3, 3)$ and $\nu_C(f_3) = (1, 2)$. Consider the possible traces of execution shown in Figure 3 with time evolving from the left to the right, and considering the events in `sdA` and `sdC` with highest priority (here assumed to have value 5 and 3 respectively).

The traces illustrate how the event duration and the (dephased) order in which execution is done for the different scenarios may or may not contain inconsistencies. The first two example traces contain inconsistencies, because events with label conflicts occur at the same time. A resolution for $trace_1$ could replace the occurrence of f_2 with f_3 (compromising on the effectiveness of f_2 but guaranteeing the higher priority of e_2), and for $trace_2$ could change the order of

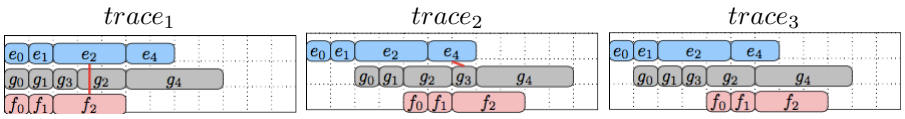

Figure 3: Possible traces of execution with and without inconsistencies.

occurrence of g_2 and g_3. Note that when having a conflict between two events with an assigned priority we always try to satisfy the event with the highest priority first. Here e_2 has priority 5 and f_2 has priority 3, so we favour e_2. If both events had the same priority the resolution would pick one of the events at random. Between traces $trace_1$ and $trace_2$, the conflict severity is lower in $trace_1$ (value -100) and hence this trace (if conflicts would be unavoidable) would be favoured. In this paper we avoid inconsistencies altogether and we do not explore this feature further. In $trace_3$ no inconsistencies are present and all events have the highest priority. In the next section we show how we can generate automatically the valid traces for a set of labelled event structures given a set of label conflicts and the degree that each structure is being dephased.

4. Isabelle Implementation and Verification

We combine two formal techniques to *calculate automatically the outcome* of the composition of two or more behavioural models as a set of allowed traces and to determine that the *result is correct*: the theorem prover Isabelle [36] and the SMT solver Z3 [34].

Isabelle is a theorem prover (proof assistant) providing a framework to accommodate logical systems (inference rules, axioms), and compute the validity of logical deductions according to the chosen logical system. In this paper, we use Isabelle's library based on *higher-order logic* (HOL): the resulting overall system is referred to as Isabelle/HOL, but here we will use Isabelle and Isabelle/HOL interchangeably. In Isabelle/HOL the basic notions are type specification, function application, lambda abstraction, and equality. Using these notions, mathematical definitions can be formulated; in turn, theorems about these definitions can be proved using the axioms describing the intuitive properties of the basic notions and the inference rules of HOL. An important point is that an Isabelle/HOL definition can be computed if suitably formulated: this allows to use Isabelle/HOL to both perform computations and formally prove their correctness by verifying theorems stating the wanted properties of the corresponding definitions.

This idea is general, and can be, in theory, used to verify any algorithm. In practice, however, the definition of the object we want to compute is often *non-constructive* and therefore, while we can still use Isabelle/HOL to prove theorems about it, we cannot directly compute it. One general approach (used in [23]) to overcome this limitation is to keep the given non-constructive definition specifying the given computational problem, to add a *computable* definition, and to prove in Isabelle that they are equivalent through a so-called *bridging*

theorem: in this way, any theorem we prove about one of the two will carry over to the other definition and, in particular, we can prove the correctness of our implementation (given by the constructive Isabelle definition) with the respect to the specification (given by the original, potentially non-constructive, Isabelle definition).

These general considerations can be applied to our problem as described in Section 3. As described earlier, a key step in obtaining a composed model is to compute the traces of the given event structures, which entails the computation of all possible configurations. We have defined event structures in Definition 1 and, according to that definition, a set of events `C` is a configuration if it is conflict-free and downward-closed. In Isabelle, a configuration is described as follows.

```
abbreviation "isDownwardClosed cau C ≡ (C ⊆ events cau &
     (∀ e f. e ∈ C & (f, e) ∈ cau → f ∈ C))"

abbreviation "isConflictFree cfl C ≡ (∀ e e'.
     e ∈ C & e' ∈ C → (e,e') ∉ cfl)"
```

Here we use `cau` to indicate the causality relation $\rightarrow^*$, given by the set of all ordered pairs related by $\rightarrow^*$, and `cfl` corresponds to the conflict relation #, given by the set of all pairs related by #. For instance, $f \rightarrow^* e$ corresponds to `(f,e)∈ cau`. Furthermore, `events cau` is a function returning the set of elements over which `cau` is defined which corresponds to the set of events Ev.

The above definitions are, however, not constructive since they describe the properties `C` must have, but not how to compute it. We mentioned that the general solution is to introduce a computable definition as follows.

```
abbreviation "extension cau C ≡ (C ∪ (cau^{−1}``C))"
abbreviation "restriction cfl C ≡ C − (cfl `` C)"

abbreviation "configurations cau cfl ≡
   {C. C ∈ Pow (events cau) & extension cau C ⊆ C &
                                        C ⊆ restriction cfl C}"
```

Above, `` applies a relation to a set, `^\{−1\}` takes the converse of a relation, and `Pow` takes the powerset. For example, let `C={e}` and `(f,e)∈cau`, then `(e,f)∈cau^\{−1\}` and `f∈(e,f)∈cau^\{−1\} ``C`, which then gives `C={e,f}`. In other words, the first expression `extension` guarantees that `C` contains all the predecessors of all events in `C`. The second expression guarantees that conflicting events of any event in `C` are not in `C`. The final expression combines both to define configurations, where `C` is a subset of the events in the considered structure. The advantage of `configurations` is that it is constructive and can therefore be used to actually compute all configurations. It is not immediate how `configurations` relates to the original definition, and we introduce the following bridging theorem to certify their equivalence.

```
theorem "(C ∈ configurations cau cfl) ↔
(isConflictFree cfl C & isDownwardClosed cau C)"
```

The next step to the solution of our problem outlined in Section 3 is finding traces (maximal configurations) in a model. A maximal set `X` in a family `F` of sets can be formulated as follows.

```
(X ∈ F & (∀ Y ∈ F. X ⊆ Y → X = Y ))
```

This states that `X` is a maximal set in a family of sets `F` if and only if any other arbitrary set `Y` in `F` larger than `X` is equivalent to `X`. This definition is also descriptive, rather than constructive, and we need to provide an equivalent, constructive definition instead. Below we use `f‘Z` to denote the image of set `Z` through function `f`.

```
abbreviation "isMaximal F X == ({} ∉ (λ Y. X−Y)‘(F−{X}))"
abbreviation "maximals F == {X ∈ F. isMaximal F X}"
```

along with a bridging theorem:

```
theorem  "X ∈ maximals F ↔ (X ∈ F & (∀ Y ∈ F. X⊆Y → X=Y))"
```

We are now able to compute all traces for a given event structure as follows.

```
abbreviation "traces cau cfl ==
    maximals (configurations cau cfl)"
```

In the above, we define `traces` over `cau` and `cfl` as maximal configurations over the same relations.

We proceed by implementing all notions outlined in Section 3 and illustrated in the example of Figure 2. We need to be able to determine whether traces for different event structures (aka models) contain events with conflicting labels (according to Γ, Definition 3), and what are the preferred traces with respect to defined priorities (according to labelling function $\nu^{(1)}$, Definition 2). In addition, we need to specify how events in a trace of one model overlap timewise with events in a trace of another model (using the duration values given by labelling function $\nu^{(2)}$, Definition 2). This corresponds to representing the traces in a linear fashion as shown in Figure 3 in accordance to `cau` and `cfl`.

Given a list `tr`, whose entries are the events in a trace, and two distinct elements `f` and `s` of such a list which are related by causality `(f,s)∈cau`, then the index of `f` must be smaller than the index of `s`. This is defined next.

```
(∀ f s. ((f,s) ∈ set cau & f ∈ set tr & s ∈ set tr) →
  the (findFirstIndex (λx. x=f) tr) <=
  the (findFirstIndex (λx. x=s) tr))
```

where `findFirstIndex` (λx. `x=e`) `l` returns the index of the first entry of the list `tr` equal to `e`. Since in general such an entry may not exist (prevented in the clause above by the conditions on `f` and `s`), this function actually returns a value of type *optional* with a special value `None` for these cases. Note that the function `the` above converts back this optional type to a natural number as required.

However, since this condition does not allow us to compute all the traces (lists) as needed, we introduce a corresponding constructive definition.

```
abbreviation "isOrderPreserving cau tr ==
  (None = (List.find (λx.x=True)
   [let m=findFirstIndex (λx.x=f) tr in
    let n=findFirstIndex (λx.x=s) tr in
      (m ≠ None & n ≠None & the m > the n ).(f,s) <- cau]))"
```

which in the last line makes use of the list comprehension notation: `.(f,s)<-cau`. This is used to parse all the pairs `(f,s)` in `cau` and applies to each one of them the function specified on the left of the dot, producing a list of the results thus obtained. The following bridging theorem ensures that `isOrderPreserving` is correct.

```
theorem "(∀ f s.
        ((f,s) ∈ set cau & f ∈ set tr & s ∈ set tr) →
         the (findFirstIndex (λ x. x=f) tr) <=
         the (findFirstIndex (λ x. x=s) tr))
                              ↔ (isOrderPreserving cau tr)"
```

Once we have all the lists representing the traces and respecting the underlying partial order given by causality, it is easy to calculate the temporal configurations of all the events occurring in one such list. This corresponds to calculating the abscissa of each event appearing in a diagram representing traces, as for instance shown in Figure 3, and can be implemented in Isabelle as follows.

```
abbreviation "clocks dephasing durations tr ==
  map (op + dephasing)
  [ listsum (map durations (take i tr)).i<-[0..<size tr]]"
```

where the function `clocks` takes a list `tr` representing a sorted trace and returns the list of the abscissas (i.e., the time at which they start) of the corresponding events. This is calculated by summing for each event the durations of the preceding events (given by function $\nu^{(2)}$, Definition 2) and by adding the dephased value of the corresponding model.

Finally, having computed all the sorted traces and the temporal scope of all events in the traces, pruning the combination of traces where events with label conflicts overlap is straightforward. From the remaining traces, the overall priority is computed through the standard Isabelle function `listsum`, then allowing us to use the function Isabelle `argmax` to pick the best combination of traces. We omit the details here, and focus, in the remainder, on the issue of the performance of the obtained implementation.

Up to now, the stress was on correctness achieved through bridging theorems between specification and implementation definitions in the functional language Isabelle/HOL. Like any implementation, however, the one we have introduced is liable to optimisations: for example, the definition of `configurations` on page 11 starts from the powerset of all events and is then refined in accordance with the properties `extension` and `restrictions`. Since the computation of the powerset is extremely expensive, one could try to find an equivalent definition for `configurations` which does not require it, and then proceed to prove a further

bridging theorem stating the correctness of the new definition. This extends the bridging theorem approach by introducing chains of equivalent definitions, each more and more efficient, linked by several bridging theorems, leading to a general approach to writing algorithms which are both efficient and formally proven correct [19].

Here, rather than following that approach, we proceed by introducing an alternative technique to improve performance by producing non-Isabelle code which is, however, still amenable to Isabelle proofs. In this case this corresponds to satisfiability modulo theories code. A satisfiability modulo theories (SMT) solver is a computer program designed to check the satisfiability of a set of formulas (known as *assertions*) expressed in first-order logic, where for instance arithmetic operations and comparison are understood, and additional relations and functions can be given a semantic meaning in order to make the problem satisfiable. The next section reformulates our problem in SMT terms, while Section 6 introduces a general technique to apply Isabelle correctness proofs to the SMT code.

5. SMT Implementation

Contrary to what we did in Section 4 with Isabelle code, we will not show here the very SMT code for our implementation. SMT code is essentially unreadable because of the limited number of native notions and constructs available, and any computation involving sets, for instance, is cumbersome. Furthermore, to increase efficiency, usually a number of transformations are applied to the code making it even less readable: e.g., a universally quantified assertion over a finite type is often replaced by multiple non-quantified assertions, each for one element of the type (quantifier elimination). In Section 6, we will see a way to ensure that all these modifications do not affect the correctness of the final SMT code. Finally, SMT-LIB [6], the standard specifying a common language for SMT solvers, consistently employs reverse polish notation, aggravating the problem of (human) readability.

Our solution to this expository problem is to write formulas close to the first-order logic language used by SMT solvers, but, for the sake of readability, adopt some simplifications. In particular, we adopt infix notation instead of prefix notation, use set-theoretical styling instead of predicates (e.g., write $(j, k) \in G_i$ instead of $G_i\ j\ k = \text{True}$), use set-theoretical operations (e.g., union, intersection, cartesian product, domain, range, etc.) instead of the corresponding first-order logic renditions, omit type specifications, and use the universal quantifier $\forall$ even when in the actual code it has been eliminated.

Recall Definitions 1, 2 and 3 (cf. Section 3). Our problem consists of an n-tuple of models

$$M_1 = (E_1, \mu_1, \nu_1), \ldots, M_n = (E_n, \mu_n, \nu_n),$$

where each model consists of an event structure and labelling functions,

$$E_1 = (Ev_1, \rightarrow_1^*, \#_1), \ldots, E_n = (Ev_n, \rightarrow_n^*, \#_n)$$

and an additional set of label conflicts Γ is provided

$$\Gamma \subseteq \bigcup_{i=1}^{n-1} \bigcup_{j=i+1}^{n} L_i \times L_j \times \mathbb{Z}$$

All notions above have been introduced, and we assume, as usual, that the sets of events $Ev_1, \ldots, Ev_n$ are pairwise disjoint. In what follows we denote the immediate causality $\rightarrow_i$ by G_i, and set

$$G := \bigcup_{i=1,\ldots,n} G_i \qquad \# := \bigcup_{i=1,\ldots,n} \#_i$$

Given a relation R over a set Y and a set $X \subseteq Y$, we introduce the notation $R^{\rightarrow}(X)$ to denote the image of X through R.

We proceed in steps: first, we show how to compute traces, then how to use ν to obtain the preferred one, depending on the priority ($\nu_i^{(1)}$) and the duration ($\nu_i^{(2)}$) assigned to arbitrary events of Ev_i.

5.1. Trace Calculation

To represent a trace of execution, we need to express which events are part of the trace and in which order. The first piece of information is given by a boolean function over all the events, namely, isSelected.

We can compute isSelected using an SMT solver as follows. Let us illustrate the procedure for a fixed event structure M_i with events given by Ev_i. A trace is a configuration and hence conflict-free and downward-closed (see Definition 1), and this can be expressed as follows:

$$\forall j, k \in Ev_i.\ \text{isSelected}(j) \wedge \text{isSelected}(k) \rightarrow \neg(j \# k)$$

$$\forall j \in \text{Range}(G_i).\ \text{isSelected}(j) \rightarrow \bigwedge_{k \in (G_i^{-1})^{\rightarrow}\{j\}} \text{isSelected}(k)$$

where Range(G_i) consists of all the events which are immediate successors of other events in M_i (in other words, j cannot be an initial event). The formulas above capture the notion of configuration, where the first represents that a configuration must be conflict-free and the second that a configuration must be closed for all predecessors of an event (known as downward-closed, cf. Definition 1) in a way amenable to SMT solvers. To compute traces of execution, we have to further capture the notion of a *maximal* configuration. This notion implies quantifying over configurations, which is not allowed in the first-order logic universe of SMT solvers. However, the notion of maximality can be reformulated in the case of configurations of finite event structures as follows.

$$\forall z \in Ev_i.\ \neg\,\mathrm{isSelected}\,(z) \rightarrow$$
$$(\exists y \in Ev_i.((y\#z \wedge \mathrm{isSelected}\,(y)) \vee ((y,z) \in G_i \wedge \neg\,\mathrm{isSelected}\,(y)))) \quad (1)$$

The formulas above can be used to compute traces via an SMT solver. More precisely, the events for which isSelected is true correspond to all the events in the trace, and the events of any legal trace must satisfy the assertions above. We will formally prove the correctness of this statement in Section 6.

To add an order to this set, we define on it a natural-valued map s_i which is an order morphism between the partial order $\rightarrow_i^*$ and the canonical order relation on $\mathbb{N}$. Therefore, we first obtain the partial order relation (let us denote it as P_i) from the immediate causality G_i.

The following assertions impose that P_i is a transitive and reflexive extension of G_i:

$$\begin{aligned} \forall j,\ k.\,(j,k) \in G_i \rightarrow (j,k) \in P_i \\ \forall j,\ k,\ l.\,(j,k) \in P_i \wedge (k,l) \in P_i \rightarrow (j,l) \in P_i \\ \forall j \in Ev_i.\,(j,j) \in P_i \\ \forall j,\ k.\,(j,k) \in P_i \wedge (k,j) \in P_i \rightarrow j = k \end{aligned} \quad (2)$$

P_i is the transitive-reflexive closure of G_i, i.e., the minimal of all the transitive and reflective extensions of G_i; therefore, on top of those assertions, conditions imposing the minimality of P_i are needed: these are discussed separately in Section 8.

Now that we have the partial order P_i, it is possible to require that s_i is an order morphism, thereby sorting all the selected events of Ev_i. This can be done by imposing that s_i is order-preserving (between the partial order P_i and the canonical order relation for natural numbers), surjective over the integer interval $[1,\ldots,|Ev_i|]$, and such that $s_i\,(j) < s_i\,(k)$ whenever j is selected and k is not:

$$\begin{aligned} \forall j,\ k.\,(j,k) \in P_i \rightarrow s_i\,(j) \leq s_i\,(k) \\ \forall j,\ k \in Ev_i.j \neq k \rightarrow s_i\,(j) \neq s_i\,(k) \\ \forall j \in Ev_i.s_i\,(j) \geq 1 \\ \forall j \in Ev_i.s_i\,(j) \leq |Ev_i| \\ \forall j,\ k \in Ev_i.\,\mathrm{isSelected}\,(j) \wedge \neg\,\mathrm{isSelected}\,(k) \rightarrow s_i\,(j) < s_i\,(k) \end{aligned} \quad (3)$$

We will see in Section 8 how to prove that one can use G_i in lieu of P_i for the job of sorting traces; we also anticipate that in Section 6 the notion of partial order will emerge again in the code verification process.

5.2. Using ν for Trace Selection

As done in the example of Figure 3, we want to be able to determine whether events from distinct event structures overlap, in order to decide whether the

conflict they might have is triggered or not. We associate a clock function to each event, expressing the time when the event starts. To calculate it, we use the sorting functions s_i obtained in the previous section, together with the duration of each event provided by ν. This can be done by requiring that an event following another (according to s_i) starts exactly when the latter ends:

$$\forall j,\ k \in Ev_i.$$
$$(\text{isSelected}\, j \wedge \text{isSelected}\, k \wedge s_i(j) \leq |Ev_i| \wedge s_i(k) \leq |Ev_i| \wedge s_i(k) - s_i(j) = 1)$$
$$\rightarrow \text{clock}(k) = \text{clock}(j) + \nu^{(2)}(j)$$

where $\nu^{(2)}(j)$ corresponds to the duration of event j. We note that some of the formulas above are logically redundant (for example, $s_i(k) \leq |Ev_i|$ follows from (3)). However, they serve the purpose of reducing the search space for the SMT solver: we recall that at the beginning of this section we anticipated the occurrence of performance-oriented modifications to the code, and this is just one of many such occurrences. The general verification technique discussed in Section 6 will make sure correctness will not be affected by these modifications.

The formula above leaves the clocks of the source nodes (i.e., the nodes with no incoming edges) undetermined, hence we need to set them separately. This allows us to introduce dephasing between different models, by specifying different clocks for the source nodes of different models, which means starting each model at dephased times. In other words, the set of values clock(j) obtained when j ranges over all the source nodes completely describes the dephasing, and can be chosen freely. Finally, the concept of clock allows us to avoid inconsistencies due to events mutually in conflict, but whose occurrence is not simultaneous.

To attain this goal, we assign a score to each event and to each pair of events from distinct models, through the function priority and Score, respectively, both yielding integer values. $\text{Score}(j,k)$ will take into account both the absolute conflict between events j and k, and their clock, in order to decide whether they are in conflict given a trace (recall, from the definition above and the definition of s_i in previous section, that each trace determines clock values for each event). Formally, this is obtained by the following requirement, repeated for all $m \neq n$, $m, n \in \{1, \ldots, n\}$:

$$\forall j \in Ev_m,\ k \in Ev_n.\, \text{isSelected}(j) \wedge \text{isSelected}(k) \rightarrow \text{Score}(j,k) =$$
$$f\left(\text{clock}(j), \text{clock}(k), \nu^{(2)}(j), D(\mu(j), \mu(k))\right)$$

where D calculates the absolute conflict (a negative number) between events based on their label, and is passed to f. For example, referring to the example depicted in Figure 2, one would have $D(\mu_A(e_2), \mu_C(f_2)) = -200$.

Further, f combines that with the distance of the event occurrences to obtain the effective result, as follows:

$$f(x_1, x_2, y, z) := \begin{cases} z, & \text{if } x_2 - x_1 \in [0, y] \\ 0, & \text{otherwise} \end{cases}$$

Besides label conflicts between events in distinct models, the other criterion when picking a trace is the absolute priority of each event. Therefore, we also require

$$\forall j.\, \text{isSelected}\,(j) \rightarrow \text{priority}\,(j) = \nu^{(1)}\,(j)$$
$$\forall j.\, \neg\, \text{isSelected}\,(j) \rightarrow \text{priority}\,(j) = 0$$

where $\nu^{(1)}(j)$ corresponds to the priority assigned to j.

To obtain the final trace, we define an integer variable `totScore` summing over all the $\text{Score}\,(j,k)$ and over all the values $\text{priority}\,(j)$, and pick the trace maximising `totScore`. This very last step corresponds to the addition of a single assertion using the Z3 keyword `maximize`, which is not part of the SMT-LIB standard, but is provided by the optimizing part of the SMT solver Z3, νZ [7]. νZ is now part of the default distribution of the recent versions of Z3.

5.3. Example

We test the output of our approach with respect to the simple example of Figure 3. In the first case ($trace_1$ of Figure 3), all the models start executing together, and the SMT solver yields the optimal trace in Table 1. The incompatibility between g_3 and e_4 does not pose problems, since those two events cannot overlap. However, the solver has been forced to choose between the branch starting at e_2 and f_2. Given that the e_2 branch has the highest priority overall, it has been picked. But event f_2 also has a high priority, which leads to his choice over f_3, as soon as dephasing allows that. We now test that this is indeed the case. Table 2 displays the output resulting from running the same experiment, but with f_0 happening at time 4 and g_0 happening at time 1 (corresponding to $trace_3$ in Figure 3).

clock	event	priority	duration
0	e_0	1	1
0	f_0	1	1
0	g_0	1	1
1	e_1	1	1
1	f_1	1	1
1	g_1	1	1
2	e_2	5	3
2	f_3	1	2
2	g_2	1	2
4	g_3	1	1
5	e_4	5	2
5	g_4	1	4

Table 1: Outputs corresponding to $trace_1$

clock	event	priority	duration
0	e_0	1	1
1	e_1	1	1
1	g_0	1	1
2	e_2	5	3
2	g_1	1	1
3	g_3	1	1
4	f_0	1	1
4	g_2	1	2
5	e_4	5	2
5	f_1	1	1
6	f_2	3	3
6	g_4	1	4

Table 2: Outputs corresponding to $trace_3$

Now, the incompatibility between e_2 and f_2 can be avoided by dephasing, and indeed both events are part of the new trace. We also note that the incompatibility between e_4 and g_3 has also been avoided by swapping the execution of g_2 and g_3, as expected.

6. Verification

The first-order language used in SMT solvers often requires laborious and error-prone translation from higher-level mathematical abstractions. Let us take the notion of event structure as an example: the concepts of partial order, and relation in general are expressed typically through sets of ordered pairs. However, the notion of set is not directly available in SMT-LIB, and one is forced to choose an alternative lower-level representation. One possibility is to represent binary relations as boolean predicates taking two arguments, but a consequence of this choice is that it will make higher-level operations (such as composition, image, taking the domain, injectivity, etc.) more complicated.

A way of tackling the complexity arising from this translation, and to make sure that it correctly represents the involved objects, is to write the wanted original definitions in a higher-order language (e.g., higher-order logic, HOL) which allows us to express them easily. In the same language, we can of course also write definitions closer to the ones required for SMT solvers. The crucial point is that Isabelle provides an SMT-LIB generator which can generate, from the latter definitions, SMT assertions directly executable by SMT solvers. This in addition means that we can prove, inside Isabelle, the equivalence between the standard definitions and those closer to the SMT language. Since we use the latter directly to generate the SMT code, the formal equivalence proof is also a proof of correctness for our generated SMT code.

To illustrate this, consider the following Isabelle definition of event structure close to the formal one given in Definition 1:

```
abbreviation "isLes causality conflict ==
  propagation conflict causality & sym conflict &
  irrefl conflict & trans causality &
  antisym causality & reflex causality"
```

Above `isLes causality conflict` returns `true` exactly when the binary relations `causality` and `conflict` satisfy the necessary conditions and hence constitute a valid event structure. In the definition above, the relations of causality and conflict are given as sets of pairs, which permits us to use the standard property of symmetry (`sym`), transitivity (`trans`) already present in the Isabelle libraries. We only needed to introduce `propagation` as a direct translation of the propagation condition occurring in Definition 1, which we omit here.

On the other hand, an equivalent definition is also introduced in Isabelle:

```
abbreviation "IsLes Causality Conflict ==
  Propagation Conflict Causality & Sym Conflict &
  Irrefl Conflict & Trans Causality &
  Antisym Causality & Reflex Causality"
```

which is similar to the previous one, but where `Causality` and `Conflict` are no longer sets, but predicates.

This allows us to use the definition of `IsLes` for producing SMT code directly through Isabelle's SMT generator. Since this generator is originally provided for theorem proving, and not for direct SMT computations as we are interested here, we have to trick Isabelle into proving a lemma:

```
lemma assumes "IsLes Causality Conflict" shows False
 sledgehammer run [provers=z3, minimize=false,
                   overlord=true, timeout=1] (assms)
```

The lemma above makes some assumptions (hypotheses) written after the keyword `assumes`. The assumptions include that the two relations described constitute a valid event structure. The keyword `shows` introduces the thesis (here `False`) and `sledgehammer` is Isabelle's command for referencing outside tools (ATPs, SMT solvers), used here to run Z3. We note that the argument `assms` is used to instruct `sledgehammer` to ignore any other theorems in the Isabelle library and consider only the stated assumptions.

In the lines above, Isabelle will pass to Z3 a file which contains one declaration for each of the relations `Causality` and `Conflict`, and assertions for each of the stated hypotheses. In the present case, we only have one hypothesis, which will result in an SMT definition of event structure, directly usable for our computations.

The last step to certify the correctness of this SMT generated code is to prove the equivalence of `isLes` and `IsLes`, which is attained through the following theorem:

```
theorem "IsLes causality conflict ↔
        (isLes (pred2set Causality) (pred2set Conflict))"
```

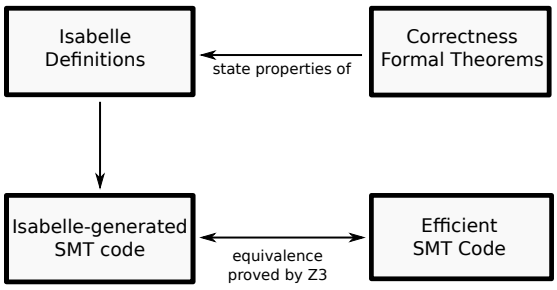

Figure 4: Overview of the formal verification of the SMT code.

where `pred2set` converts from relations represented as predicates into relations represented as sets.

The idea of using Isabelle as an interface to SMT code becomes even more fruitful in cases where the SMT code used for computing a given object departs substantially from the original or standard mathematical definition of that object. This usually happens, e.g., because the original definition is not directly expressible as a finite number of formulae in first-order logic (the language of SMT solvers), or because, even if it is, it is inefficient. In such cases, we can express both the original definition and the definition used for SMT computing in Isabelle, which we can then use both to generate the SMT code for the latter and to formally prove the equivalence of the two definitions, as from the diagram in Figure 4. As an example, let us take the trace computation seen at the beginning of Section 5.1: there we had to resort to an alternative, less intelligible definition of maximal configuration (1), because the original definition implied quantifying over all configurations.

In Isabelle, we can easily render the pen-and-paper definitions of an event structure (which we have seen earlier). A configuration is defined as follows.

```
abbreviation "isConfiguration Ca Cf C =
  isConflictFree Cf C & isDownwardClosed Ca C"
```

In addition, a trace (maximal configuration) is defined next.

```
abbreviation "isTrace Ca Cf C =
  isConfiguration Ca Cf C &
  (∀ Y. Y ⊃ C → ¬ (isConfiguration Ca Cf Y))"
```

where the last line expresses that the configuration `C` is maximal. We write the same line in the way seen in Section 5.1:

```
abbreviation "isMaximalConfSmt Ca Cf C ==
  (∀ z ∈ events Ca − C.
     z ∈ Cf``C ∨ (immediatePredecessors Ca {z})−C ≠ {})"
```

where `immediatePredecessors Ca {z}` returns all the events e satisfying $e \to z$ (we recall that $\to$ is the immediate causality obtained from $\to^*$). Finally, the following Isabelle theorem states that (1) is equivalent, for a configuration of a finite event structure, to the original trace definition:

```
theorem correctness: assumes "finite Ca" "isLes Ca Cf"
 "isConfiguration Ca Cf C" shows
 "(isTrace Ca Cf C) ↔ isMaximalConfSmt Ca Cf C"
```

We note that the theorem assumes that `C` is a configuration: this is not a problem because, as seen in Section 5.1, the notion of configuration admits a straightforward formulation in SMT, while the problematic one is that of a *maximal* configuration. We also note that `isMaximalConfSmt` builds on `immediatePredecessors Ca`, rather than directly on `Ca`. This is also not a problem, since the SMT computations we introduced in Section 5.1 take as input the immediate causality relations $G_i, i = 1, \ldots, n$, and use them to calculate via SMT the causality relations $\rightarrow_i^*$.

The Isabelle definition `isMaximalConfSmt` can be used to automatically generate SMT code through `sledgehammer`, as we did with `IsLes`. This corresponds to the vertical arrow on the left in Figure 4. In this case, however, the obtained SMT code is not as efficient as the one we wrote in Section 5.1. In general, the efficiency of SMT code can depend dramatically on formal details, such as eliminating universal quantifiers by explicit enumeration, rewriting assertions in normal form, etc. We want to keep both the efficiency of the non-Isabelle-generated SMT code and the correctness of the Isabelle-generated SMT code. Our solution is to take both, and prove their equivalence using the SMT solver itself. This corresponds to the horizontal arrow at the bottom of Figure 4, and can be implemented as follows. We introduce an SMT boolean function `maximality` which is true exactly when (1), repeated for each $i = 1, \ldots, n$, is true. We also introduce another boolean function `maximalityIsabelle`, defined by using the SMT code generated by Isabelle using `isMaximalConfSmt`. If `maximality` and `maximalityIsabelle` were not equivalent, there would be some isSelected satisfying one but not the other. Therefore, we challenged the SMT solver as follows:

```
(assert (or (and maximality (not maximalityIsabelle))
            (and (not maximality) maximalityIsabelle)))
```

obtaining the answer (`unsat`), which guarantees that the SMT code we use for maximal configuration (trace) calculation is correct.

Correctness, as usual, means that if we trust the SMT solver, Isabelle, and the environment in which they run, then we can trust that the result of our computation is indeed a trace. Furthermore, we know that any trace will satisfy the SMT formula (i.e., Formula 1) passed to the solver for the computation. To increase our confidence in the results, we could also prove the correctness of the remaining computations, for instance trace selection (Section 5.1). The general mechanism represented in Figure 4 could again be applied: we would need to write an Isabelle formal specification of the desired property guiding the trace selection, write an Isabelle definition to generate SMT code, and an Isabelle theorem proving that the latter obeys the former. Finally, we would use the SMT solver to prove that the Isabelle-generated SMT code and the non-Isabelle-generated SMT code are equivalent. Again, this would imply correctness as soon as we trust the solver and Isabelle, though in this case we would also

need to trust νZ (see end of Section 5.2), which we use for trace selection. This is because the νZ commands `minimize` and `maximize` cannot be expressed in the first-order logic fragment of SMT-LIB, and hence cannot undergo the transformation represented by the vertical arrow in the diagram of Figure 4.

One of the motivations we gave above for the approach represented in Figure 4 is the ability to rewrite SMT formulas into logically equivalent ones for performance reasons. Some of such rewritings, however, depend on the arguments. For example, if we replace a universal quantifier ranging over all the events of an event structure with the explicit enumeration of each single event, we obtain a logical formula which varies upon which event structure we are considering. An important consequence of this is the need, in general, of repeating the last SMT assertion given above for each case.

7. Completeness

The optimal trace obtained and verified previously is usually not the only possible one. The SMT solver will not return sub-optimal traces, and will arbitrarily break any possible tie among optimal solutions when there are several. However, in our case we want to obtain all possible traces (respecting certain constraints) to gain a complete understanding of the valid composed model.

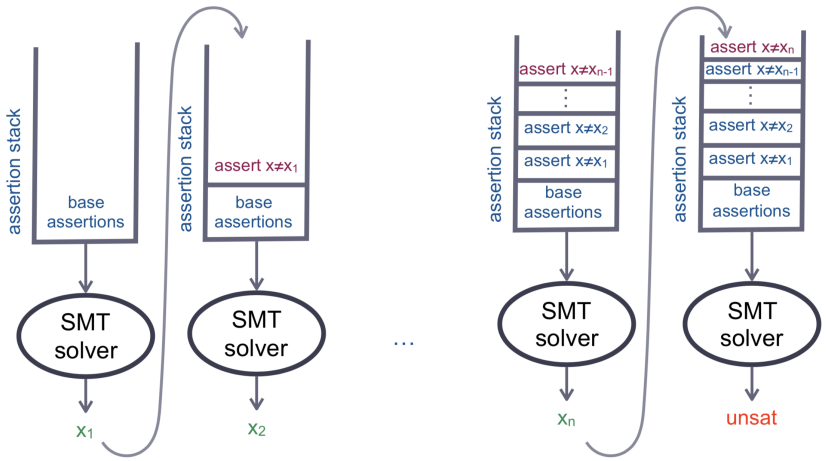

Figure 5: Incremental approach

To this end, the idea is to first compute the optimal trace (cf. Section 5), called *base assertions* in the sequel, then successively add assertions to the SMT solver's stack imposing that the next solution differs from the previous one (along with all the previous assertions characterising the previous solution), and iterate until no further solutions can be found and the solver returns `unsat`. This approach is schematically represented in Figure 5.

We make two remarks on the above approach: first, this scheme reuses the same base assertions for each computed solutions, thereby carrying over the correctness results introduced in Section 6 to each computed solution. Second, since the solutions are determined incrementally, without restarting the SMT solver between each solution, the solver will typically be able to improve its performance after finding the first solution: this is because a part of its internal computations are common over the determination of all the solutions, and hence can be saved after finding the first.

However, implementing the scheme just proposed presents technical challenges: the main problem is that, after finding the first solution (e.g., x_1 in Figure 5), there is no way, in the SMT-LIB standard used by SMT solvers, of referring to it in the subsequent assertions (as in, e.g., the assertion "assert $x \neq x_1$" appearing in the second step of Figure 5).[1] There are provisions to attain this in some APIs provided by some SMT solvers [7]; however, giving up SMT-LIB in favour of a particular API would mean no longer being able to apply Isabelle correctness proofs to SMT code as outlined in Section 6. More precisely, the vertical arrow on the left of Figure 4 relies on Isabelle's SMT-LIB generator, which in turn exploits the close adherence of SMT-LIB with the first-order logic fragment contained in higher-order logic. Additionally, these APIs are dependent on the particular SMT solver used, while SMT-LIB has the advantage of preserving some freedom in the choice of the solver to run our SMT code on.

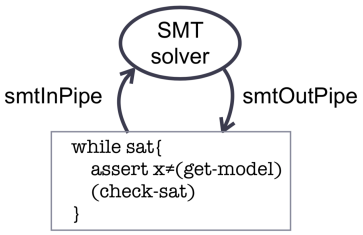

Figure 6: Schematic diagram of the incremental invocation of the SMT solver

Our solution to hurdle this is to keep the SMT solver running as a background process, waiting for commands to be supplied through a UNIX named pipe (let us call it `smtInPipe`), and outputting its results to another named pipe (let us call it `smtOutPipe`). In parallel with this background process, a simple UNIX Bourne shell script loops analysing the output received on `smtOutPipe`: when the script sees `sat`, it requests the SMT solver the current solution through the SMT-LIB command (`get-model`), saves the solution on a cumulative file, formulates assertions for the next solution to be distinct from the current solutions and

[1] See the comment at https://stackoverflow.com/questions/11867611/z3py-checking-all-solutions-for-equation#comment15793591_11869410 from one of the lead developers of the Z3 SMT solver.

clock	event	order	priority	duration
0	e_0	1	1	1
1	e_1	2	1	1
1	g_0	1	1	1
2	e_2	3	5	3
2	g_1	2	1	1
3	g_3	3	1	1
4	f_0	1	1	1
4	g_2	4	1	2
5	e_4	4	5	2
5	f_1	2	1	1
6	f_3	3	1	2
6	g_4	5	1	4

Table 3: A trace of score 20

sends them to the solver through `smtInPipe`. This is iterated until the script sees `unsat` on `smtOutPipe`. This `unsat` message is important not only because it allows the mechanism to terminate, but also because, through it, the SMT solver provides a partial correctness proof stating the completeness of the results we obtained. Figure 6 represents this solution.

A secondary technical issue is that the illustrative pseudocode `assert x`$\neq$ `x_1` appearing in Figure 5 cannot usually expressed directly as appears. Indeed, `x` represents a model, which is typically expressed through several objects not being first-class in the first-order logic of SMT. In particular, those objects cannot be directly the subject of an inequality assertion: for example, in case the model `x` is represented by a function over some finite type, we will instead assert that there is at least one argument over which it yields a different value than the previously found model `x_1`. Therefore, the particular additional assertions behind the pseudocode `assert x`$\neq$`x_1` will depend on the problem at hand.

We applied this approach to the example from Section 3 (with the dephasings corresponding to the rightmost part of Figure 3), obtaining all the possible execution traces. Referring to this particular application of the technique described above, note that the dephasings are not variables to be optimised by the solver, but arbitrary constants fixed before running our solver. The optimal score found is 22, but there were other execution traces scoring at 20, 13 and 11, with no other score values possible. A solution with score 22 had already been reported in Table 2. We represent, in the same form, one solution for each of the sub-optimal score values 20, 13 and 11 in Table 3, 4 and 5 respectively.

8. Relations between Optimisations and Verification

In Section 4, we used the Isabelle function `isOrderPreserving` to construct a sorting of traces, and then proved its correctness through the formal theorem on page 13, which states that the sorting preserves the partial order given by the

clock	event	order	priority	duration
0	e_0	1	1	1
1	e_1	2	1	1
1	g_0	1	1	1
2	e_3	3	1	3
2	g_1	2	1	1
3	g_2	3	1	2
4	f_0	1	1	1
5	f_1	2	1	1
5	g_3	4	1	1
6	f_2	3	3	3
6	g_4	5	1	4

Table 4: A trace of score 13

clock	event	order	priority	duration
0	e_0	1	1	1
1	e_1	2	1	1
1	g_0	1	1	1
2	e_3	3	1	3
2	g_1	2	1	1
3	g_2	3	1	2
4	f_0	1	1	1
5	f_1	2	1	1
5	g_3	4	1	1
6	f_3	3	1	2
6	g_4	5	1	4

Table 5: A trace of score 11

causality relation. The corresponding computation using SMT solving is that of the formulas in (3). In this section, we discuss how the general verification idea put in place in Section 6 cannot only be used to prove the correctness of these formulas, but also to improve their performance while preserving the validity of the proof.

We start by defining an SMT boolean variable `isMonotonicSMT` defined as the conjunction of some of the formulas appearing in (3) for all values of i ranging over the given event structures; more precisely, we take the formulas

$$\begin{aligned} \forall j,\ k.\,(j,k) \in P_i \rightarrow s_i\,(j) \leq s_i\,(k) \\ \forall j,\ k \in Ev_i.j \neq k \rightarrow s_i\,(j) \neq s_i\,(k) \\ \forall j \in Ev_i.s_i\,(j) \geq 1 \\ \forall j \in Ev_i.s_i\,(j) \leq |Ev_i| \end{aligned}$$

let i range over all the given event structures, and take the conjunction to determine the value of `isMonotonicSMT`. Note that the last formula of (3) was

dropped, being it an additional condition not related to monotonicity. We also define an integer-yielding SMT function s over all the nodes by taking the union of all the functions s_i, so that `isMonotonicSMT` is actually a condition imposed on s.

Wanting to apply again the idea of Section 6, we will also need an Isabelle counterpart of `isMonotonicSMT` stating in Isabelle that s is monotonic. This is expressed by the following statement.

```
abbreviation "isMonotonicIsabelle P s ==
(∀ e1 e2. P e1 e2 & e1 ≠ e2 → s e2 > s e1) &
(∀ e1 e2.
    (e1 ≠ e2 & (∃ e0 . (P e0 e1 & P e0 e2))) → s e1 ≠ s e2) &
(∀ e. ((s e)::int) ≥ 1) &
(∀ e2. s e2 > 1 →
    (∃ e0 e1. s e1=s e2 − 1 & P e0 e1 & P e0 e2 & s e0=1))"
```

This Isabelle definition can be used to automatically generate a corresponding SMT definition by forcing Isabelle into proving a dummy theorem, as seen in Section 6. In this case a possible dummy theorem is:

```
lemma assumes "isMonotonicIsabelle' p s"
 shows False sledgehammer
 [provers=z3,minimize=false,timeout=1,overlord=true](assms)
```

the generated SMT code corresponds to the bottom left node in Figure 4, while `isMonotonicIsabelle` corresponds to the top left node and `isMonotonicSMT` corresponds to the bottom-right node. Although the definition of `isMonotonicIsabelle` could be already be seen itself as stating the monotonicity of s, we also proved a more explicit Isabelle theorem granting this, and corresponding to the top-right node in Figure 4:

```
lemma assumes "isMonotonicIsabelle' (set2pred P) s"
  "s e0 < s e1" "(e0, e1) ∈ P ∨ (e1,e0) ∈ P"
  shows "(e0, e1) ∈ P & e0 ≠ e1"
```

This theorem states that if `e0` and `e1` are comparable through the partial order `P`, then `s` must respect their `P`-order. Corresponding to the bottom horizontal arrow in the diagram (Figure 4), we finally prove the equivalence of `isMonotonicSMT` and the SMT code automatically generated from `isMonotonicIsabelle` by making sure to obtain `unsat` following the assertion:

```
(assert (or (and isMonotonicIsabelle (not isMonotonicSMT))
            (and (not isMonotonicIsabelle) isMonotonicSMT)))
```

Having formally established the monotonocity of the s function as specified by the assertions (3), we can wonder whether we can optimise those assertions, for example by reducing their number. It is intuitive that, by virtue of the transitivity of the partial orders P_i's, we can reduce the assertions corresponding to the first line of formulas (3) by substituting it with:

$$\forall j,\ k.\, (j,k) \in G_i \rightarrow s_i\,(j) \leq s_i\,(k)$$

Recall that G_i only contains the pairs being bound by *immediate* causality, thereby reducing the number of checks the SMT solver has to perform. If we modify the definition of `isMonotonicSMT` accordingly, the last assertion we wrote still returns `unsat`, thereby providing correctness proof of the new implementation with very little effort.

Similar modifications can be implemented and cheaply verified in other parts of the SMT code: for example, the assertions represented by last lines in formulas (2), and imposing antisymmetry of the transitive closure P_i of G_i can intuitively be dropped due to the fact that the antisymmetry of G_i is preserved through the operation of transitive–reflexive extension performed by the lines above it. This is again confirmed by re-running the assertion above, thereby carrying over the whole proof we already produced to the new implementation. We can crudely compare the original algorithm against the one improved with the optmisations we introduced in this section by applying them to the simple example given in Table 2: the former computes that solution in 5.13 seconds, while the latter takes 2.09 seconds to produce the same solution. These are run times on the same machine, averaged over three runs after discarding the first.

We conclude this section discussing one point we left open in Section 5, where we mentioned that formulas (2) only produce a transitive–reflexive extension of the immediate causality G_i. On top of those assertions, additional ones have to be imposed specifying that the extension we pick is minimal. We implemented two versions of this step. One is written purely in first-order logic and is based on recursion on the graph G:

$$\forall l,\ n.\ (l,n) \in P \rightarrow l = n \vee (l,n) \in G \vee \exists m.\, ((l,m) \in G \wedge (m,n) \in P)$$

where P is the union of all the partial orders $P_1, \ldots, P_n$. The second version works by reducing the minimisation of the extension to the minimisation of its cardinality, which, being a numerical minimisation problem, can be dealt with by the numerical optimisation capabilities of νZ. Concretely, this means defining an SMT integer variable `numberOfEdges` counting the number of pairs contained in P and then adding the assertion

```
(minimize numberOfEdges)
```

Two observations about this minimisation are in order: first, such minimisation must appear before the maximisation of `totScore` (see end of Section 5). This is because the solver will give priority to the optimisation constraint in the order in which they appear, and we want to maximise `totScore` over all the possible transitive closures of G; therefore, the computation of the transitive closure, which involves the `minimize` command above, must happen before the maximisation of `totScore`. The second observation is that the two versions of the transitive closure computation above present one important difference: the second implementation using `minimize` breaks the verification idea represented in Figure 4. This is because the `minimize` statement cannot be expressed in first-order logic, and therefore, will not be included in the definition of `maximality` (see Section 6). As a consequence `maximality` will no longer actually represent

maximality correctly, and the assertion appearing on page 22 will obviously return `sat`, which means we cannot conclude that our code is correct. This is why we also provide the first implementation not using `minimize`: however, it is slightly slower than the one using it.

9. Clinical example

The approach introduced in this paper can be used to obtain a composed model in a variety of different applications. Here, we show a concrete example from the medical domain (taken from [47]) to illustrate what our technique can do when we do not have a complete behavioural model, and instead only a partial understanding of how individual diseases are managed (clinical guidelines) and the consequences of applying them simultaneously on a patient. We consider atrial fibrillation and chronic kidney disease as conditions to be managed.

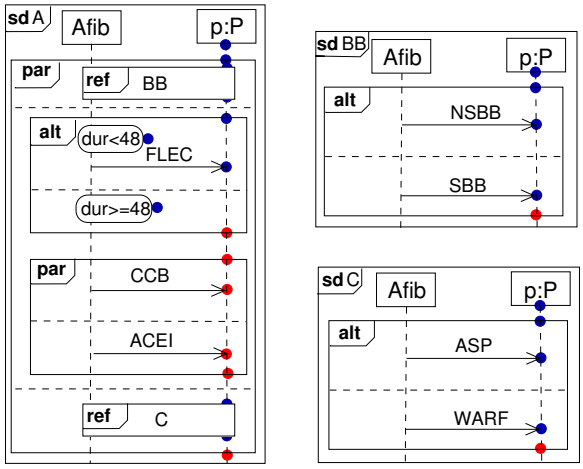

Figure 7: Modelling Atrial fibrillation (Afib)

Figure 7 shows a clinical guideline for atrial fibrillation (`Afib`) modelled as a sequence diagram where the main lifeline of interest is the patient lifeline `p`, and the messages received imply actions or checks to manage the disease. To keep the models simpler, some of the details are referred to (using the interaction fragment `ref`) other models (here `BB` and `C`). As earlier in the paper, we mark the locations along the lifeline of `p` which will correspond to events in the formal model. We colour-coded the locations to indicate locations associated to duplicated events caused by alternative behaviour (red) or single events (blue).

The sequence diagram `Afib` shows within the main parallel fragment, three operands (separated by dashed lines) executing in parallel: (1) the patient receives a beta blocker (diagram `BB`, and the priority is to give a non-selective beta blocker (`NSBB`) though an alternative of a selective beta blocker (`SBB`) exists); (2) if the duration of the Afib episode was less than 48 hours then the patient

receives flecainide (FLEC), and thereafter both a calcium channel blocker (CCB) and an ACE inhibitor (ACEI) in an arbitrary order; and (3) the behaviour given in sequence diagram C is executed. This corresponds to a choice between two medications: aspirin (ASP) and warfarin (WARF).

The composition of fairly simple models quickly becomes very complex, as can be seen with the Afib labelled event structure of Figure 8.

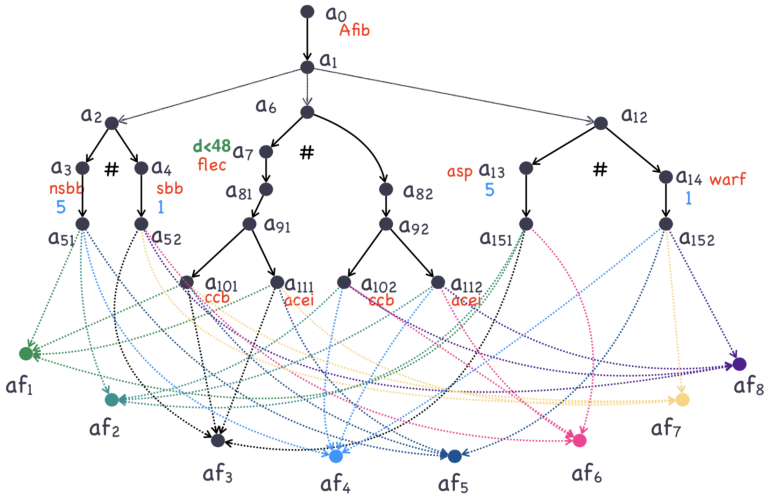

Figure 8: Labelled event structure for Atrial fibrillation (Afib)

The model contains eight possible traces. The initial event $\mu_A(a_0) = \{afib\}$ has a label to indicate the diagnosis of atrial fibrillation. The model shows the medication given at different events (e.g., $\mu_A(a_{13}) = \{asp\}$), conditions that may be required to hold (e.g., $\mu_A(a_7) = \{d < 48, flec\}$ indicating that $flec$ is prescribed if the episode has lasted less than 48 hours). Some events have an associated priority value because they are preferred choices in the clinical guideline (e.g., $\nu_A(a_3) = (5, n)$ where n is the duration the medication will be given for). There are two traces with the highest priority: τ_1 with maximal event af_1 and τ_2 with maximal event af_2. Our solver would obtain either of these as the best treatment path for a patient with atrial fibrillation.

The clinical guideline associated to chronic kidney disease (CKD) is shown in Figure 9 on the left. The severity of CKD is determined by analysing the results of the estimated glomerular filtration rate (EGFR). If the value is below 60 (which is the case for more severe stages of the disease), the patient needs to be checked for anemia (shown separately in diagram Acheck). Thereafter, a patient is checked for metabolism abnormalities (mabn). If they are present the patient is given a phosphate binder (PB). Such a patient receives further recommendations given in the separate diagram MGMNT. This specifies two concurrent checks: life-style management for CKD (ls mgt ckd) and cardio-vascular risk management (cv risk mgt). If the value of EGFR is greater or equal to 60

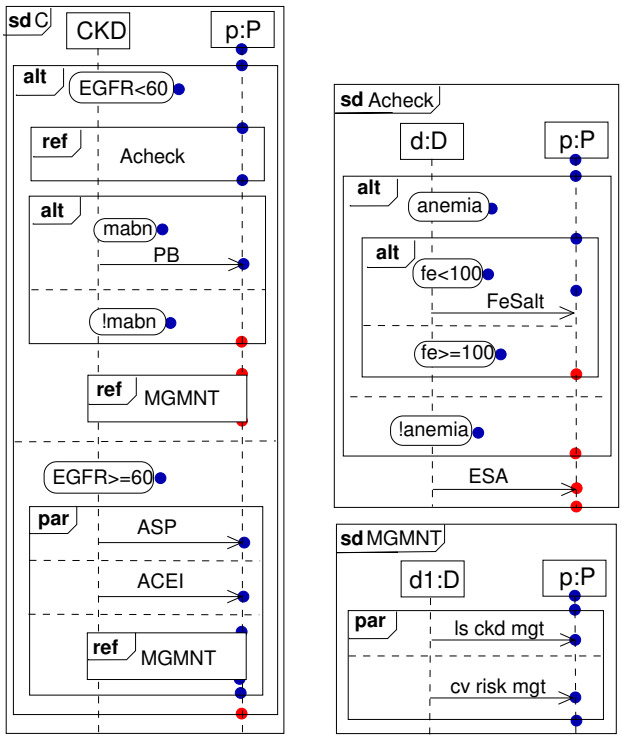

Figure 9: Modelling CKD, checking for anemia, and CKD and CV risk management

(first two stages of the disease), then the patient receives (in arbitrary order) aspirin (`ASP`), an angiotensin-converting enzyme (ACE) inhibitor (`ACEI`), and recommendation for management of conditions `MGMNT`. For checking anemia the following is done. If the ferritin level (`fe`) is below 100, the patient receives ferrous salt (`FeSalt`), otherwise nothing is prescribed. In both cases the patient is given an erythropoiesis-stimulating agent (`ESA`). We do not show the formal model for `CKD`. The initial event in the model, say c_0, would have a label to indicate the diagnosis, i.e., $\mu_C(c_0) = \{ckd\}$. Let c_5 denote the event with marking given by $\mu_C(c_5) = \{asp\}$, a medication prescribed if `EGFR`≥ 60.

Consider a patient with both `Afib` and `CKD`, and with `EGFR`≥ 60. Experts recommend that a patient with `CKD` should not be given a non-selective beta blocker `NSBB` and such a medication has to be replaced by a selective beta blocker `SBB`, and the same medication cannot be prescribed twice for the different purposes (e.g., aspirin). We hence consider the following conflict labels $\Gamma = \{(nsbb, ckd, -100), (asp, asp, -200)\}$. It is clear that we have to avoid both label conflicts mentioned: prescribing `nsbb` and overdosing on aspirin. Our solver indeed now discards traces within `Afib` (cf. Figure 8) that pass event a_3 reducing the valid traces to four: τ_3 (maximal event af_3), τ_6 (maximal event af_6), τ_7 (maximal event af_7) and τ_8 (maximal event af_8). Further avoiding a_{13}

(prescribing aspirin twice), we are reduced to two traces: τ_7 and τ_8 with the same score.

In general, our approach offers a flexible mechanism to compute all valid traces for complex scenarios where patients with several chronic conditions have to follow different treatment plans simultaneously.

From a computational point of view, the resulting SMT code is around 550KB, and its generation time is negligible when compared to its execution time. The SMT solver returns a solution in less than 10 minutes on an off-the-shelf machine.

10. Related Work

Systems are usually designed through a combination of several models, some to capture structural aspects and some to describe more complex aspects of behaviour. As argued in [26], modelling the complete behaviour of a component or subsystem is difficult and error prone. Instead, it is easier to formulate partial behaviour as scenarios in Live Sequence Charts (LSCs), UML sequence diagrams or similar. One of the problems that arises from partial modelling is potentially inconsistent or incomplete behaviour.

When looking at the integration of multiple model views or diagrams, Widl et al. [46] offer a solution in the context of model versioning, and how to integrate versions of a sequence diagram in accordance with the behaviour given in state machines. They make direct use of SAT-solvers for detecting inconsistencies in the (different versions of a) sequence diagram, with respect to the state machine. Liang et al. [33] present a method of integrating sequence diagrams based on the formalisation of sequence diagrams as typed graphs. Both these papers focus on less complex structures. For example, they do not deal with combined fragments, which can potentially cause substantial complexity. More fundamentally, however, the motivation of their work is considerably different from ours, as we are concerned with the inconsistency between different sequence diagrams and do not assume given a complete understanding of the system behaviour through a state machine. Bowles and Bordbar [14] present a method of mapping a design consisting of class diagrams, OCL constraints and sequence diagrams into a mathematical model for detecting and analysing inconsistencies. It uses the same underlying categorical construction as done in [12] but it has not been automated. On the other hand, Zhang et al. [50] and Rubin et al. [43] use Alloy for the composition of class diagrams. They transform UML class diagrams into Alloy and compose them automatically. They focus on composing static models and the composition code is produced manually.

We used Alloy to automatically compose sequence diagrams in [13, 15]. Our experience with Alloy has shown that it has limitations which have a direct impact on the scalability of the approach [16]. There is an exponential growth in time when trying to compose diagrams with an increasing number of elements, which becomes unusable in practice. The Alloy analyzer is SAT solver-based and SAT-solving time may increase enormously, depending on factors such as the number of variables and the average length of the clause [24]. Z3 [34] performs

much better and we have used it in more recent work [16, 31, 17]. In the context of coordination, Z3 has been used to find solutions for constraints capturing glue code in coordination models in [40].

We are addressing inconsistent combination of behavioural models in this paper. A SAT-based approach, such as Alloy, would allow us to detect inconsistencies and highlight them, as a result of not being able to generate a solution for the composition. When two or more scenarios combined have inconsistencies, a designer benefits not only from knowing which inconsistencies there are, but what traces of execution can bypass the inconsistencies. In practice, it is unlikely that inconsistencies can be removed altogether, and instead we want to find the traces that are valid, avoid the inconsistencies, and may satisfy additional criteria such as priorities. SAT solvers cannot be used in this case whereas we have shown that SMT solvers can in another context [31]. The present paper makes a novel contribution by showing how SMT solvers such as Z3 can be used to find the best solution to a generally unsolvable problem of composing models with known (label) inconsistencies. In other words, if there is no solution completely avoiding inconsistencies, our score approach provides a metric allowing the user to express preferences among inconsistencies, and hence compute a solution respecting such preferences. Finally, the typical combination of SMT solvers and proof assistants is done to help finding proofs, and we bring this combination into a completely different setting for detecting and resolving problems in complex behaviour. Our approach provides the well known advantages of theorem proving approach making it a valuable complement to other formal methods [38].

The underlying semantic model used in our composition approach is the labelled prime event structure [49]. As mentioned before, the mathematical simplicity of the model is well suited to our combined theorem prover/SMT solver approach for searching for valid traces of execution. However, we could have explored a variety of alternative models, and indeed techniques, to tackle similar problems including for instance (weighted timed) automata [11] or other variants of labelled transition systems. Such models are compositional, have been studied for many years and adequately capture the semantics of design languages and component-based approaches.

To avoid problems associated to very complex and large models, which in model checking leads to state explosion problems, compositional verification (cf. for instance [25] for a detailed description of assume-guarantee reasoning) addresses the complexity in the context of component-based systems by replacing individual components we can abstract from by assumptions. A compositional proof for the overall system will need to guarantee the consistency of the assumptions associated to the individual components, but it does simplify the verification problem considerably. Treating our individual models as components and describing them as assumptions, for then checking their consistency would be an alternative approach to ours when wanting to detect inconsistency between some of our models. However, we expect that determining the assumptions for each component when we allow for some inconsistent labels is much harder to express in general.

Coordination models and languages provide high-level abstractions and a clean separation between individual software components, their interactions and dynamic composition within an overall software organisation. In particular, these models and languages have made component-based systems more amenable to verification and global analysis. Various formalisms help with verification in this setting, and are often variants of labelled transition systems or automata-based models. One such example includes constraint automata [4] which provide an operational semantics for the coordination language Reo [2]. An interesting aspect of using a compositional model such as constraint automata (mainly developed to suit Reo), and similarly for other automata, is that we can apply techniques available for labelled transition systems, and define powerful notions of bisimulation equivalence and simulation pre-order. These make it possible to compare automata, and check for instance, whether one system or component is observationally equivalent to another, and so on. Model comparison was not, however, the focus of our present work.

11. Conclusions

Inspired by a problem from the medical domain, we have explored a novel approach to compose behavioural models and their underlying, possibly dephased, traces of execution. Our approach detects and avoids inconsistencies of models prior to composition, and constructs the correct composition by generating the complete set of valid traces of execution underlying the composed model. The traces can be fine-tuned to take into account additional requirements on the degree of priority that one model or certain steps in a process (events in our approach) have over alternatives. Moreover, our approach is able to find the best trace of execution with respect to these constraints, and then pushes this further by forcing the SMT solver to yield all the possible traces sorted by optimality. To achieve our goal, we introduced a novel solution to the technical problem of referring to previous SMT-computed solutions within one run of the solver, and without giving up the SMT-LIB format, which is key to the proposed approach.

Even though several formal techniques could have been picked, we opted for the use of SMT solvers in our setting because we search for valid traces of execution within a composed model, and generate solutions incrementally and sorted by how optimal the traces are. As we are interested in all valid traces for combinations of finite models of behaviour, the implicit notion of optimal solutions given by the SMT solver is not explored, but it gives us a mechanism to consider further. For example, we can investigate how to obtain an optimal valid trace simultaneously under several different criteria (for instance, selected labels, mutually exclusive selection of labels, and so on) and keep the ability of formally proving properties of such a trace.

From a computational point of view, the problem we presented combines traits of distinct well-explored problems in combinatorial optimisation (CO). For example, one could try to apply existing, well-known algorithms for the trace-finding portion of our problem, and then hope to reduce the remaining portions of the problem to well-understood problems in CO, maybe by applying

integer programming (IP), one of the domains of CO richest in methods and algorithms, to the sub-problem of optimality as given by overlapping labels. However, it is not clear how to express such a problem in terms of integer variables and functions. More importantly, even if one finds an IP rendition of it, that rendition cannot, in general, be assumed to be linear. This implies that we do not have direct access to the powerful and well-developed methods of integer linear programming (ILP), and have to deal with the more difficult approaches available in the less developed field of non-linear programming, where typically one has to resort to techniques (reconnecting the problem to special cases) such as continuous relaxation, branch and bound algorithms, approximation algorithms, ad-hoc heuristics, or fractional programming [27, 22, 3]. All these approaches typically require significant amounts of work, add complexity to the solutions and require assumptions about the problem specification.

By contrast, our approach avoids these issues by adopting a holistic attitude, in that it expresses all the constraints and the optimality requirements for our problem as a single stack of SMT assertions, without splitting it into sub-problems. An additional, important advantage to this approach is that it enables us to employ a general technique we have used in the past to obtain correct-by-construction SMT code using the theorem prover Isabelle/HOL.

Before choosing the problem treatment presented in this paper, as explained above in the case of IP, we reviewed many other combinatorial optimisation sub-domains, without finding techniques able to capture all the aspects of the problem we consider. For example, resource scheduling [42, Section 22], an actively researched field, deals with executing a set of activities, each needing to employ some resources, while respecting the resource capacities, temporal constraints, and while optimising a given objective function. While this can accommodate the concepts of nodes (as activities) and labels featured in our problem, the focus of the constraints in resource scheduling is on the capacities of the resources and maybe (through the objective function) on temporal optimisations (e.g., minimising the number of late activities). By contrast, in our problem the optimisation focuses on the mutual interaction of resources.

We use a novel combination of the theorem prover Isabelle/HOL and SMT solver Z3, where the theorem prover guarantees the correctness of the approach and facilitates the interaction with the SMT solver through the SMT-LIB generator provided by Isabelle/HOL. This is important not only because writing SMT code directly is difficult, but because existing APIs of SMT solvers have not been formally verified. It should be noted that Isabelle's SMT generator is used internally as an automated deduction tool, and has not been conceived to be exposed to the user. We showed how this feature can act as a link between the two languages we use, and how to use the highly expressive higher-order logic to both ease the definitions and prove the correctness of our labelled event structure definitions in both higher-order logic and SMT code.

Indeed, in this paper we showed a two-fold application of SMT solvers: Isabelle's SMT generator used for theorem proving (Section 6), and for generating SMT code used directly for computations (Section 5). The original theorem proving application of SMT solvers employs the capability of the latter of gen-

erating proofs of unsatisfiability (see e.g., [10]), but crucially imposes a final step in which the theorem prover attempts to reconstruct the proof found by the SMT solver: if that attempt is successful, only the theorem prover needs to be trusted in order to rely on the overall correctness of the result. This is to be contrasted with our approach, in which the infrastructure existing for that original approach is employed to generate SMT code for purposes different than (and in addition to) proof generation. For this reason, we cannot perform the final proof reconstruction step, and have to trust both the SMT solver and the theorem prover. In particular, we note that the Isabelle component generating SMT code also relies on proof reconstruction. However, potential unsoundness deriving from it is unlikely for two reasons: (1) work has specifically been done to improve the generator and to informally prove its soundness [9, 8]; (2) possible soundness problems are linked to the fact that higher-order logic is much richer than SMT-LIB first-order logic [8, Section 2]. In our approach, this is issue is greatly mitigated by the fact that we use the translator for Isabelle code which only uses a first-order logic fragment of HOL, while the type-richer definitions are confined to duplicate Isabelle definitions which are used for theorem proving and which are proved equivalent, *within* Isabelle, to the first-order definitions seen by the translator generating SMT code. This is the point, for example, for introducing the duplicate definitions `isLes` and `IsLes`, as discussed at the beginning of Section 6.

While this paper focused on the semantics of the underlying behavioural models, we are also developing mechanisms to visualise the solutions obtained back to the designer. We have used Graphviz in our earlier work in [13, 15] to show the composition solution obtained with Alloy. In future work we want to explore visualisations that work directly on the modelling approaches used by designers, and in particular in the case of inconsistencies, can show them more effectively; thus we also aim at achieving an increased adoption of our approach by designers, which in turn is needed to test and validate our techniques on realistic application problems. To this end, it would be ideal to preliminarily modularise the work we presented here in a way similar to what we did elsewhere [20], to facilitate the addition of components taking care of input, output, conversion to and from specification formats, etc. Work is in progress to generalize the time representation to allow the duration of an event to be a range, rather than a specific amount of time units. Future work will also tackle the issue of finding a way to accommodate infinite loops and non-terminating behaviours, possibly present in given models.

Finally, our verification mechanism for SMT code provides the freedom to experiment with SMT code to find provably equivalent forms resulting in better performance from the SMT solver: we discussed some concrete applications of this possibility with respect to formula (2) from Section 5.1 and in Section 6. Currently, however, such modifications to the SMT code are found empirically. Indeed, further research is needed to make this idea more systematic, and to complement alternative approaches aiming at improving solver performance [5].

References

[1] Araújo, J., Whittle, J., Kim, D., 2004. Modeling and composing scenario-based requirements with aspects. In: Requirements Engineering (RE 2004). IEEE Computer Society Press, pp. 58–67.

[2] Arbab, F., 2004. Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science 14, 1–38.

[3] Avriel, M., Diewert, W. E., Schaible, S., Zang, I., 2010. Generalized concavity. Vol. 63 of Classics for Applied Mathematics. Siam.

[4] Baier, C., Sirjani, M., Arbab, F., Rutten, J., 2006. Modeling component connectors in reo by constraint automata. Science of Computer Programming 61, 75–113.

[5] Balunovic, M., Bielik, P., Vechev, M., 2018. Learning to solve SMT formulas. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (Eds.), Advances in Neural Information Processing Systems 31 (NIPS 2018). Curran Associates, Inc., pp. 10317–10328.

[6] Barrett, C., Stump, A., Tinelli, C., 2010. The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (Eds.), Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK).

[7] Bjørner, N., Phan, A.-D., Fleckenstein, L., 2015. νz-an optimizing smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015). Vol. 9035 of LNCS. Springer, pp. 194–199.

[8] Blanchette, J. C., Böhme, S., Paulson, L. C., 2013. Extending Sledgehammer with SMT solvers. Journal of automated reasoning 51 (1), 109–128.

[9] Blanchette, J. C., Böhme, S., Popescu, A., Smallbone, N., 2013. Encoding monomorphic and polymorphic types. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, pp. 493–507.

[10] Böhme, S., Weber, T., 2010. Fast LCF-style proof reconstruction for Z3. In: International Conference on Interactive Theorem Proving. Vol. 6172 of LNCS. Springer, pp. 179–194.

[11] Bouyer, P., 2006. Weighted timed automata: Model-checking and games. Electronic Notes in Theoretical Computer Science 158, 3–17.

[12] Bowles, J., 2006. Decomposing Interactions. In: Johnson, M., Vene, V. (Eds.), Algebraic Methodology and Software Technology (AMAST 2006). Vol. 4019 of LNCS. Springer, pp. 189–203.

[13] Bowles, J., Alwanain, M., Bordbar, B., Chen, Y., 2015. Matching and merging scenarios automatically with Alloy. In: et al., S. H. (Ed.), Model-Driven Engineering and Software Development. Vol. 506 of CCIS. Springer, pp. 100–116.

[14] Bowles, J., Bordbar, B., 2007. A formal model for integrating multiple views. In: Application of Concurrency to System Design (ACSD 2007). IEEE Computer Society Press, pp. 71–79.

[15] Bowles, J., Bordbar, B., Alwanain, M., 2015. A logical approach for behavioural composition of scenario-based models. In: M. Butler, S. C., Zaïdi, F. (Eds.), Formal Methods and Software Engineering: 17th International Conference on Formal Engineering Methods. Vol. 9407 of LNCS. Springer, pp. 252–269.

[16] Bowles, J., Bordbar, B., Alwanain, M., June 2016. Weaving true-concurrent aspects using constraint solvers. In: Application of Concurrency to System Design (ACSD 2016). IEEE Computer Society Press, pp. 35–44.

[17] Bowles, J., Caminati, M., 2016. Mind the gap: addressing behavioural inconsistencies with formal methods. In: 23rd Asia-Pacific Software Engineering Conference (APSEC). IEEE Computer Society, pp. 313–320.

[18] Bowles, J., Caminati, M., 2017. Correct composition of dephased behavioural models. In: Proença, J., Lumpe, M. (Eds.), Formal Aspects of Component Software (FACS 2017). Vol. 10487 of LNCS. Springer, pp. 233–250.

[19] Bowles, J., Caminati, M., 2017. A verified algorithm enumerating event structures. In: Geuvers, H., England, M., Hasan, O., Rabe, F., Teschke, O. (Eds.), Intelligent Computer Mathematics (CICM 2017). Vol. 10383 of LNCS. Springer, pp. 239–254.

[20] Bowles, J., Caminati, M., Cha, S., Mendoza, J., 2019. A framework for automated conflict detection and resolution in medical guidelines. Science of Computer Programming.

[21] Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., Wimmer, M., 2011. The Past, Present, and Future of Model Versioning. IGI Global, Ch. 15, pp. 410–443.

[22] Burer, S., Letchford, A. N., 2012. Non-convex mixed-integer nonlinear programming: A survey. Surveys in Operations Research and Management Science 17 (2), 97–106.

[23] Caminati, M., Kerber, M., Lange, C., Rowat, C., 2015. Sound auction specification and implementation. In: 16th ACM Conference on Economics and Computation (EC 2015). ACM, pp. 547–564.

[24] D'Ippolito, N. N., Frias, M., Galeotti, J., Lanzarotti, E., Mera, S., 2010. Alloy+hotcore: A fast approximation to unsat core. In: Abstract State Machines, Alloy, B and Z. Vol. 5977 of LNCS. Springer, pp. 160–173.

[25] Giannakopoulou, D., Namjoshi, K. S., Păsăreanu, C. S., 2018. Compositional reasoning. In: Clarke, E. M., Henzinger, T. A., Veith, H., Bloem, R. (Eds.), Handbook of Model Checking. Springer, Cham, Ch. 12, pp. 345–383.

[26] Harel, D., Marelly, R., 2003. Come, Let's Play: Scenario-based Programming Using LSCs and the Play-Engine. Springer.

[27] Hemmecke, R., Köppe, M., Lee, J., Weismantel, R., 2010. Nonlinear integer programming. In: 50 Years of Integer Programming 1958-2008. Springer, pp. 561–618.

[28] Jackson, D., 2006. Software Abstractions: logic, language and analysis. MIT Press.

[29] Kaufmann, P., Kronegger, M., Pfandler, A., Seidl, M., Widl, M., 2014. A sat-based debugging tool for state machines and sequence diagrams. In: SLE 2014: Software Language Engineering. Vol. 8706 of LNCS. pp. 21–40.

[30] Klein, J., Hélouët, L., Jézéquel, J., 2006. Semantic-based weaving of scenarios. In: AOSD'06. ACM, pp. 27–38.

[31] Kovalov, A., Bowles, J., 2016. Avoiding medication conflicts for patients with multimorbidities. In: 12th International Conference on Integrated Formal Methods (iFM 2016). Vol. 9681 of LNCS. Springer, pp. 376–392.

[32] Küster-Filipe, J., 2006. Modelling concurrent interactions. Theoretical Computer Science 351, 203–220.

[33] Liang, H., Diskin, Z., Dingel, J., Posse, E., 2008. A general approach for scenario integration. In: MoDELS 2008. Vol. 5301 of LNCS. Springer, pp. 204–218.

[34] Moura, L. D., Bjørner, N., 2008. Z3: An efficient SMT solver. In: TACAS 2008. Vol. 4963 of LNCS. Springer, pp. 337–340.

[35] Nielsen, M., Plotkin, G., Winskel, G., 1981. Petri nets, event structures and domains, part i. TCS 13, 85–108.

[36] Nipkow, T., Paulson, L. C., Wenzel, M., 2002. Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Vol. 2283 of LNCS. Springer.

[37] OMG, 2011. UML: Superstructure. Version 2.4.1. OMG, http://www.omg.org, document id: formal/2011-08-06.

[38] Ouimet, M., Lundqvist, K., March 2007. Formal software verification: Model checking and theorem proving. Tech. rep., Mälardalen University, Sweden.

[39] Polypharmacy Model of Care Group, 2018. Polypharmacy Guidance: Realistic Prescribing, 3rd Edition. Scottish Government.

[40] Proença, J., Clarke, D., 2013. Data abstraction in coordination constraints. In: ESOCC 2013: Advances in Service-Oriented and Cloud Computing. Vol. 393 of CCIS. pp. 159–173.

[41] Reddy, R., Solberg, A., France, R., Ghosh, S., 2006. Composing sequence models using tags. In: Proc. of MoDELS Workshop on Aspect Oriented Modeling.

[42] Rossi, F., Van Beek, P., Walsh, T., 2006. Handbook of constraint programming. Elsevier.

[43] Rubin, J., Chechik, M., Easterbrook, S., 2008. Declarative approach for model composition. In: Proceedings of the 2008 international workshop on Models in Software Engineering. ACM, pp. 7–14.

[44] Uchitel, S., Brunet, G., Chechik, M., 2009. Synthesis of partial behavior models from properties and scenarios. IEEE Transactions on Software Engineering 35 (3), 384–406.

[45] Whittle, J., Araújo, J., Moreira, A., 2006. Composing aspect models with graph transformations. In: Proceedings of the 2006 international workshop on Early aspects at ICSE. ACM, pp. 59–65.

[46] Widl, M., Biere, A., Brosch, P., Egly, U., Heule, M., Kappel, G., Seidl, M., Tompits, H., 2014. Guided merging of sequence diagrams. In: Software Language Engineering (SLE 2012). Vol. 7745 of LNCS. pp. 164–183.

[47] Wilk, S., Michalowski, M., Michalowski, W., Rosu, D., Carrier, M., Kezadri-Hamiaz, M., 2017. Comprehensive mitigation framework for concurrent application of multiple clinical practice guidelines. Journal of Biomedical Informatics 66, 52–71.

[48] Winskel, G., 1982. Event structure semantics for CCS and related languages. In: Nielsen, M., Schmidt, E. (Eds.), Automata, Languages, and Programming. Vol. 140 of LNCS. Springer, pp. 561–576.

[49] Winskel, G., Nielsen, M., 1995. Models for Concurrency. In: Abramsky, S., Gabbay, D., Maibaum, T. (Eds.), Handbook of Logic in Computer Science, Vol. 4, Semantic Modelling. Oxford Science Publications, pp. 1–148.

[50] Zhang, D., Li, S., Liu, X., 2009. An approach for model composition and verification. In: Fifth International Joint Conference on INC, IMS and IDC, 2009. IEEE Computer Society Press, pp. 1102–1107.