Correct Composition of Dephased Behavioural Models[⋆]

Juliana Bowles and Marco B. Caminati

School of Computer Science, University of St Andrews
KY16 9SX St Andrews, United Kingdom
{jkfb|mbc8}@st-andrews.ac.uk

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 (behavioural) models. In recent work, we have shown how the theorem prover Isabelle can be combined with the constraint solver Z3 to efficiently detect inconsistencies in two or more behavioural models and, in their absence, generate the composition. Here, we extend our approach further and show how to generate the correct composition (as a set of valid traces) of dephased models. This work has been inspired by a problem from a medical domain where different care pathways (for chronic conditions) may be applied to the same patient with different starting points.

1 Introduction

To cope with the complexity of modern systems, design approaches combine a variety of languages and notation to capture different aspects of a system, and separate structural from behavioural models. In itself behavioural modelling is also difficult, and rather than attempt to model the complete behaviour of a (sub)system [22], 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 [19]. Given a set of scenarios, we then need to check whether these are correct and consistent, and to do so we first need to obtain the combined overall behaviour. The same ideas apply if we model (partial) business processes within an organisation, for instance using BPMN [18]. In either case, we need a means to compose models (scenarios or processes), and when this cannot be done, detect and resolve inconsistencies.

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 [1,6,20,12,15,21,23,24,26,3,4,7]. 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 [12]. In our

⋆ This research is supported by EPSRC grant EP/M014290/1.

recent work [3,4,7], 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 Alloy [11] in [3,4] and Z3 [16] in [7]. We conducted several experiments in [7], showing that Z3 performs much better than Alloy for large systems. Using Alloy for model composition, mostly in the context of structural models, is an active area of research [21,26], but the use of Z3 is a novelty of [7]. Even though we used Z3 in [7], we did not explore Z3's arithmetic capabilities, nor did we deal with incompatible constraints. We have addressed both points more recently in [8].

As in our earlier work, our approach in [8] used event structures [25] as an underlying semantics for sequence diagrams in accordance to [14,5], and explored how the theorem prover Isabelle [17] and constraint solver Z3 [16] 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 (through a set of valid traces) for behavioural models that may contain inconsistencies. We introduce a notion of *dephased* models prior to composition, to make it possible to combine models which do not start execution simultaneously, and where the *pace* of execution or a notion of *priority* in each model may be different as well. 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 care pathways (for chronic conditions) may be applied to the same patient with different starting points (diagnosis). As an additional contribution, we present in Section 4 an original, general method to provide formal correctness proof for SMT code.

This paper is structured as follows. The motivation and contribution of the work presented here are discussed in Section 2, while in Section 3 we recall our formal model (labelled event structures). Section 4 describes how Isabelle and Z3 are combined to compute valid traces of execution in specific settings. We describe the role that Isabelle plays in our work in Section 5. We conclude the paper with a description of related work in Section 6, and a discussion of future work in Section 7.

2 Context and Contribution

Continuing the work started in [8], we exploit the interface between Isabelle and Z3 to obtain a versatile tool for the specification, analysis and computation of the behaviour of complex distributed concurrent systems. By specifying our partial behavioural models in Isabelle we can check automatically their correctness, obtain their composition (if it exists) and fill any gaps, while being able to prove at any point that the models are valid [8]. If our model contains inconsistent behaviours, we are able to locate the conflicting events. However, we argue in this paper, that we may be overlooking valid behaviour in some cases, and we explore an approach to fine-tune the detection of inconsistent behaviour further.

In order to do so we allow models to be *dephased*, that is, different scenarios (or similarly for processes) may start execution at different times and continue execution at a different pace. We also consider a notion of priority of (locations in) a model. We develop a technique to find 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.

The problem we are addressing has been inspired by a problem from a medical domain where different care pathways (essentially processes or behavioural models) for chronic conditions are being applied to the same patient, such that:

- different pathway steps are 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, in other words, has higher *priority*.
- some of the possible medications prescribed at a given *step* in the pathway 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 pathways 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 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. Consider the following example using UML sequence diagrams [19].

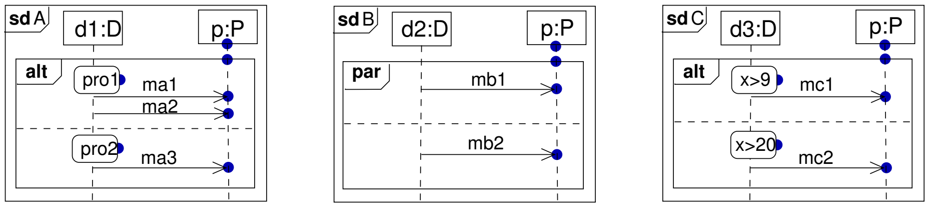

Fig. 1. Three scenarios involving the same object instances.

Fig. 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. [19] for details). Interaction fragments contain one or more operands, which in the case of an alternative may be preceded by a constraint or guard. 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 we will 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 lifelines and next to the conditions are what we call *locations*, borrowing terminology from Live Sequence Charts (LSCs) [10]. They do not serve a purpose at the design level but make it easier to understand the formal semantics (cf. [14] 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.

We now want to obtain the composition of these three diagrams in such a way that the known underlying conflicts are taken into account. To the best of our knowledge the only automated approach that can detect the conflicts in the scenarios above given such additional constraints is our work in [8]. We now extend our approach to find valid paths in a composed model that avoids these conflicts.

Clearly, to avoid the 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 (preference) 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 message 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 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 [3,4,7]. See [13] for a description of the medical problem of treating patients with multimorbidities.

3 Formal Model

The model we use to capture the semantics of a sequence diagram is a labelled (prime) event structure [25], or event structure for short. The advantages of an event structure are the underlying simplicity of the model and how it naturally describes fundamental notions present in behavioural models including sequential, parallel and iterative behaviour (or the unfoldings thereof) as well as nondeterminism (cf. [14,5]).

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 (e.g., an event occurring in one operand of an alternative fragment excludes events in another operand). From the two relations defined on 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. As a derived notion we thus obtain a way to model events associated to locations from different operands in a parallel fragment. The formal definition, as provided for instance in [14], 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*.

Event structures are enriched with a labelling function $\mu : Ev \rightarrow 2^L$ that maps each event onto a subset of elements of L. This labelling function is necessary to establish a connection between the semantic model (event structure) and the syntactic model it is describing. The set L of labels in our case 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 α, then e denotes the occurrence of that action α. If $\mu(e)$ contains a formula or logical proposition φ then φ must hold when e occurs.

We consider an additional labelling function $\nu : Ev \rightarrow \mathbb{N} \times \mathbb{N}$ to associate to each event its *priority* and *duration*. For an event e with $\nu(e) = (p, \_)$, the highest the value of p the higher the priority associated to the event. The second component of $\nu(e) = (\_, d)$ gives d, the time units spent at event e. The labelling function ν is used later when fine-tuning the composition with respect to label conflicts.

A labelled event structure over a set of labels L is a triple $M = (Ev, \mu, \nu)$. Let $M_1, \ldots, M_n$ with $M_i = (E_i, \mu_i, \nu_i)$ a finite set of labelled event structures over sets of labels L_i with $1 \leq i \leq n$. Let $\mathcal{L} = \bigcup_{i=1}^{n} L_i$. A finite set of *label constraints* defined over $\mathcal{L}$ is given by $\Gamma \subseteq L_i \times L_j$ where $i \neq j$ characterising label conflicts.

We do not show how to generate an event structure from a sequence diagram, just the general idea. 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 see for instance [14].

Recall the example of Fig. 1 introduced in the previous section. The locations along the lifeline of instance `p` have been marked in Fig. 1. 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. The label conflicts are given by $\Gamma = \{(ma1, mc1), (ma2, mb2)\}$. The behaviour of `p` in the individual diagrams of Fig. 1 is shown in the three event structures M_A, M_B and M_C of Fig. 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`.

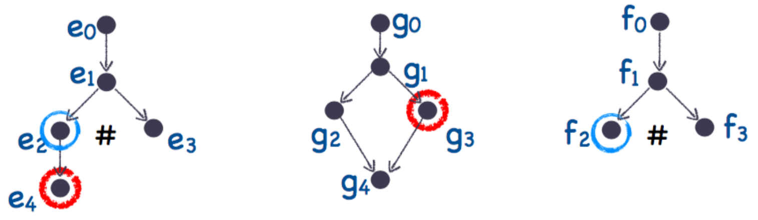

Fig. 2. Corresponding event structures for instance `p`.

The labels of some of the events above are inconsistent/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 the label inconsistencies are detected and avoided. A composed model that avoids the labels 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). However, these traces may not be the best traces of execution. 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 it. The labelling function ν gives us that information.

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

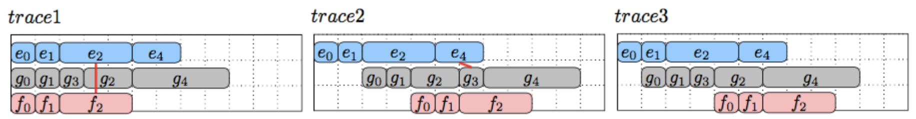

Fig. 3. Possible traces of execution with and without inconsistencies.

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 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. 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 and Z3 Combined

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 [17] and the SMT solver Z3 [16].

Isabelle is a theorem prover or proof assistant which provides a framework to accommodate logical systems (deductive rules, axioms), and compute the validity of logical deductions according to a given system. In this paper, we use Isabelle's library based on *higher-order logic* (HOL). In HOL, the basic notions are type specification, function application, lambda abstraction, and equality. In addition to be able to check logical inference over logical systems, theorem provers such as Isabelle also contain automated deduction tools, and interfaces to external tools such as SMT solvers and automated theorem provers. We use the theorem prover to guarantee the correctness of our models, the composition result and traces.

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. Within proof assistants, SMT solvers are used to find proofs by adding already proved theorems to the list of assertions, and by negating the statement to be proved to reach a contradiction. If a SMT solver returns `unsat`, then a proof can be reconstructed from the given assertions. The integration between Isabelle and SMT solvers such as Z3 provides users an additional powerful combination to be able to produce more proofs automatically. We use the SMT solver to identify label inconsistencies, which may require the use of arithmetic operations and comparison, and to find a solution which avoids the inconsistencies and considers the additional labelling information given by ν.

Let $M_1, \ldots, M_n$, with $M_i = ((Ev_i, \rightarrow_i^*, \#_i), \mu_i, \nu_i)$ and $1 \leq i \leq n$ over a set of labels L_i, be a list of finite event structures. Let $\varGamma \subseteq L_i \times L_j$ with $i \neq j$ denote the set of label conflicts. We assume that the corresponding sets of events are pairwise disjoint. In what follows we denote the immediate causality $\rightarrow_i$ by G_i, and set

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

$$\# := \bigcup_{i=1,\ldots,n} \#_i.$$

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 will now proceed in steps: first, we show how to compute traces, then we show how to use ν to obtain the preferred one, depending on the duration and priority assigned to single events. In doing so, we will write formulas close to the first-order logic language used by SMT solvers; for the sake of readability, however, we will employ some simplifications. In particular, we adopt infix notation instead of prefix notation, we use set-theoretical styling instead of predicates (e.g., writing $(j, k) \in G_i$ in lieu of $G_i\ j\ k$), and we omit type specifications.

4.1 Trace Calculation

To represent an execution trace, we need to express which events are part of it, and in which order. The first piece of information will be 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 Ev_i. The conditions of isSelected being conflict-free and downward-closed (see Definition 1) are straightforward to express:

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

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

The two formulas above capture the notion of configuration (see Definition 1) in a way amenable to SMT solvers. To compute traces of execution (Section 3), we have to 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: sets in general are not first-class objects. However, the notion of maximality can be reformulated in the case of configurations of finite event structures as follows:

$$\begin{aligned} \forall z \in Ev_i.\ \exists y \in Ev_i. \qquad & ((y \# z \wedge \text{isSelected}\,(y)) \vee \\ & ((y, z) \in G_i \wedge \neg\,\text{isSelected}\,(y))). \end{aligned} \tag{1}$$

The formulas above can be used to compute traces via an SMT solver; more precisely, the events for which isSelected is true represent the event set of a trace, and the event set of any legal trace satisfies the assertions above. We will formally prove the correctness of this statement in Section 5.

To add an order to this set, we proceed as follows. First, taken a single G_i, we need to obtain the corresponding partial order P_i (effectively obtaining the original causality relation $\rightarrow_i^*$), which can be derived from the following assertions:

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

We now use P_i to obtain a sorting of all the selected events of Ev_i. This can be done by introducing an injective function $s_i : Ev_i \rightarrow \mathbb{N}$, and then imposing that it 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:

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

4.2 Using ν for Trace Selection

As done in the example of Fig. 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:

$$\begin{aligned}&\forall j,\ k \in Ev_i.\\ &(\text{isSelected}\, j \wedge \text{isSelected}\, k \wedge s_i\,(j) \leq |Ev_i| \wedge s_i\,(k) \leq |Ev_i| \wedge s_i\,(k) - s_i\,(j) = 1)\\ &\rightarrow \text{clock}\,(k) = \text{clock}\,(j) + \nu_2\,(j)\,,\end{aligned}$$

where ν_2 is the second component of ν, yielding the duration.

The formula above leaves the clocks of the roots undetermined, hence we need to set them separately. This allows us to introduce dephasing between different models, by specifying different clocks for the roots of different models, which means starting each model at dephased times. 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 priority (which we also refer to as score) to each event and to each pair of events from distinct models, through the function priority and Score, respectively, both yielding integer values. 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 $m \neq n$, $m, n \in \{1, \ldots, n\}$:

$$\begin{aligned}\forall j \in Ev_m,\ k \in Ev_n.\, &\text{isSelected}\,(j) \wedge \text{isSelected}\,(k) \rightarrow \text{Score}\,(j,k) =\\ &f\,(\text{clock}\,(j)\,, \text{clock}\,(k)\,, \nu_2\,(j)\,, D\,(\mu\,(j)\,, \mu\,(k)))\,,\end{aligned}$$

where D calculates the absolute conflict (a negative number) between events based on their label, and is passed to f. 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 conflicts between events in distinct models, the other criterion when picking a trace is the absolute priority of each event. Therefore, we also require

$$\begin{aligned}\forall j.\, \text{isSelected}\,(j) &\rightarrow \text{priority}\,(j) = \nu_1\,(j)\\ \forall j. \neg\, \text{isSelected}\,(j) &\rightarrow \text{priority}\,(j) = 0,\end{aligned}$$

where ν_1 is the first component of ν, yielding the priority.

To obtain the final trace, we sum over all the Score (j, k) and over all the values priority (j), and pick the trace maximising such sum. To do so, we need to exploit the optimizing part of the SMT solver Z3, νZ [2].

4.3 Example

We test the output of our approach with respect to the simple example of Fig. 3. In the first case (*trace*1 of Fig. 3), all the models start executing together, and the SMT solver yields the optimal trace on the left of 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. The right-hand side of Table 1 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 Fig. 3):

Table 1. Outputs corresponding to *trace*1 (left) and *trace*3 (right)

clock	event	order	priority	duration
0	e0	1	1	1
0	f0	1	1	1
0	g0	1	1	1
1	e1	2	1	1
1	f1	2	1	1
1	g1	2	1	1
2	e2	3	5	3
2	f3	3	1	2
2	g2	3	1	2
4	g3	4	1	1
5	e4	4	5	2
5	g4	5	1	4

clock	event	order	priority	duration
0	e0	1	1	1
1	e1	2	1	1
1	g0	1	1	1
2	e2	3	5	3
2	g1	2	1	1
3	g3	3	1	1
4	f0	1	1	1
4	g2	4	1	2
5	e4	4	5	2
5	f1	2	1	1
6	f2	3	3	3
6	g4	5	1	4

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.

5 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 a lower-level representation of it. For example, by representing relations as boolean predicates taking two arguments; this, in turn, typically makes 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 (for example higher-order logic, HOL) which allows 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 assertion directly executable by SMT solvers. And, at the same time, we can prove, inside Isabelle, the equivalence between the standard definitions and those closer to the SMT language. Since the latter directly generate the SMT code used for our computations, the formal equivalence proof is also a proof of correctness for our generated SMT code.

Hence, we write in Isabelle a definition of event structure which is close to 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 `causality` and `conflict` constitute a valid event structure. In the definition above, causality and conflicts are sets of pairs, which permits 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)
```

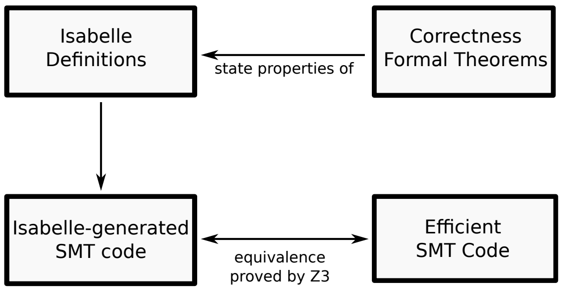

Fig. 4. Overview of the formal verification of the SMT code.

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))",
```

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 formula 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 4.1: there we had to resort to an alternative, less intelligible definition of maximality of configuration (1), because the original definition implied quantifying over all configurations.

In Isabelle, we can easily render the pen-and-paper definitions of event structure (which we have seen earlier), of configuration and of trace. We start by writing the condition specifying that our candidate configuration `C` is conflict-free:

```
definition "isConflictFree Cf C = ((C × C) ∩ Cf = {})",
```

and the condition about `C` being downward closed:

```
definition "isDownwardClosed Ca C =
        (C ⊆ events Ca &
        (∀ e f. e ∈ C & (f, e) ∈ Ca → f ∈ C))".
```

This allows the immediate definition of configuration:

```
definition "isConfiguration Ca Cf C =
  isConflictFree Cf C & isDownwardClosed Ca C",
```

and that of being a trace:

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

where the last line expresses the maximality of the configuration `C`. We write the same line (i.e., maximality) in the way seen in Section 4.1:

```
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 4.1, the notion of configuration admits a straightforward formulation in SMT, while the problematic one is that of *maximality* for a 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 4.1 take as input the immediate causality relations $G_i, i = 1, \ldots, n$, and use them to calculate via SMT the causalities $\to_i^*$.

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 Diagram 4. In this case, however, the obtained SMT code is not as efficient as the one we manually wrote in Section 4.1: it is a general fact that the efficiency of SMT code can depend dramatically on formal details, such as eliminating quantifiers by explicit enumeration, rewriting the assertions in normal forms, etc... We want to keep both the efficiency of the manually-written 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 Diagram 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 trace maximality calculation is correct. Correctness, as usually, 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. Not only: we can rest assured 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, i.e., the trace selection (Section 4.1). 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 manually written SMT code are equivalent. Again, this would imply correctness as soon as we trust the solver and Isabelle; additionally, in this case we would also need to trust νZ (see end of Section 4.2), which is not used in trace computation but only in trace selection.

6 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 [10], 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 several model views or diagrams, Widl et al. [24] deal with composing concurrently evolved sequence diagrams in accordance to the overall behaviour given in state machines. They make direct use of SAT-solvers for the composition. Liang et al. [15] 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. Bowles and Bordbar [6] 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 [5] but it has not been automated. On the other hand, Zhang et al. [26] and Rubin et al. [21] 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 [3,4]. Our experience with Alloy has shown that it has limitations which have a direct impact on the scalability of the approach [7]. 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 [9]. Z3 [16] performs much better and we have used it in more recent work [7,13,8]. We do not know of other approaches using Z3 for model composition.

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 [13]. 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 inconsistencies. 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.

7 Conclusions

Inspired by a problem from the medical domain, we have explored a novel approach to compose scenarios and their underlying, possibly dephased, traces of execution. Our approach allows us to detect and avoid inconsistencies (if possible) to generate a valid set of traces of execution for a 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 other models or alternatives. Moreover, our approach is able to find the best trace of execution with respect to these constraints. Key to our approach is the use of SMT solvers to search for the best solution. Our approach uses a novel combination of the theorem prover Isabelle and the constraint solver Z3, where the theorem prover is fundamental to guarantee the correctness of the approach and to facilitate the interaction with Z3 through the provided SMT-LIB generator. This is important because, on one hand, writing SMT code directly is time-consuming and error-prone while, on the other hand, the existing interfaces of SMT solvers with higher-level languages (e.g., APIs) are not currently, to the best of our knowledge, formally verified.

This paper focused on the semantics of the underlying behavioural models. Separately we are developing mechanisms to visualise the solutions obtained back to the designer. We have used Graphviz in our earlier work in [3,4] 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. 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. A further direction for future work is to make the scheme presented here to deal with incompatibilities and priorities even more flexible by using soft constraints: currently, the trace selection is performed by expressing a maximisation problem with hard constraints only; however, soft constraints can be implemented, e.g., via the SMT-LIB command `check-sat-assuming`. Finally, future work will also tackle the issue of finding a way to accommodate indefinite loops and non-terminating behaviours, possibly present in given models, in our approach.

References

1. Araújo, J., Whittle, J., Kim, D.: Modeling and composing scenario-based requirements with aspects. In: RE 2004. pp. 58–67. IEEE Computer Society Press (2004)
2. Bjørner, N., Phan, A.D., Fleckenstein, L.: νz-an optimizing smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 194–199. Springer (2015)
3. Bowles, J., Alwanain, M., Bordbar, B., Chen, Y.: Matching and merging scenarios automatically with Alloy. In: et al., S.H. (ed.) Model-Driven Engineering and Software Development. pp. 100–116. CCIS 506, Springer (2015)
4. Bowles, J., Bordbar, B., Alwanain, M.: 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. pp. 252–269. LNCS 9407, Springer (2015)
5. Bowles, J.K.F.: Decomposing Interactions. In: AMAST 2006. pp. 189–203. LNCS 4019, Springer (2006)
6. Bowles, J., Bordbar, B.: A formal model for integrating multiple views. In: ACSD 2007. pp. 71–79. IEEE Computer Society Press (2007)
7. Bowles, J., Bordbar, B., Alwanain, M.: Weaving true-concurrent aspects using constraint solvers. In: Application of Concurrency to System Design (ACSD 2016). IEEE Computer Society Press (June 2016)

8. Bowles, J.K.F., Caminati, M.B.: Mind the gap: addressing behavioural inconsistencies with formal methods. In: 23rd Asia-Pacific Software Engineering Conference (APSEC). IEEE Computer Society (2016)
9. D'Ippolito, N.N., Frias, M., Galeotti, J., Lanzarotti, E., Mera, S.: Alloy+hotcore: A fast approximation to unsat core. In: Abstract State Machines, Alloy, B and Z. pp. 160–173. LNCS 5977, Springer (2010)
10. Harel, D., Marelly, R.: Come, Let's Play: Scenario-based Programming Using LSCs and the Play-Engine. Springer (2003)
11. Jackson, D.: Software Abstractions: logic, language and analysis. MIT Press (2006)
12. Klein, J., Hélouët, L., Jézéquel, J.: Semantic-based weaving of scenarios. In: AOSD'06. pp. 27–38. ACM (2006)
13. Kovalov, A., Bowles, J.: Avoiding medication conflicts for patients with multimorbidities. In: 12th International Conference on Integrated Formal Methods. pp. 376–392. LNCS 9681, Springer (2016)
14. Küster-Filipe, J.: Modelling concurrent interactions. Theoretical Computer Science 351, 203–220 (2006)
15. Liang, H., Diskin, Z., Dingel, J., Posse, E.: A general approach for scenario integration. In: MoDELS 2008. pp. 204–218. LNCS 5301, Springer (2008)
16. Moura, L.D., Bjørner, N.: Z3: An efficient smt solver. In: TACAS 2008. pp. 337–340. LNCS 4963, Springer (2008)
17. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS 2283, Springer (2002)
18. OMG: Business Process Model and Notation. Version 2.0. OMG, http://www.omg.org. (2011), document id: formal/2011-01-03.
19. OMG: UML: Superstructure. Version 2.4.1. OMG, http://www.omg.org. (2011), document id: formal/2011-08-06
20. R.Reddy, Solberg, A., R.France, Ghosh, S.: Composing sequence models using tags. In: Proc. of MoDELS Workshop on Aspect Oriented Modeling (2006)
21. Rubin, J., Chechik, M., Easterbrook, S.: Declarative approach for model composition. In: MiSE 2008. pp. 7–14. ACM (2008)
22. Uchitel, S., Brunet, G., Chechik, M.: Synthesis of partial behavior models from properties and scenarios. IEEE Transactions on Software Engineering 35(3), 384–406 (2009)
23. Whittle, J., Araújo, J., Moreira, A.: Composing aspect models with graph transformations. In: Proceedings of the 2006 international workshop on Early aspects at ICSE. pp. 59–65. ACM (2006)
24. Widl, M., Biere, A., Brosch, P., Egly, U., Heule, M., Kappel, G., Seidl, M., Tompits, H.: Guided merging of sequence diagrams. In: SLE 2012. pp. 164–183. LNCS 7745, Springer (2013)
25. Winskel, G., Nielsen, M.: Models for Concurrency. In: Abramsky, S., Gabbay, D., Maibaum, T. (eds.) Handbook of Logic in Computer Science, Vol. 4, Semantic Modelling, pp. 1–148. Oxford Science Publications (1995)
26. Zhang, D., Li, S., Liu, X.: An approach for model composition and verification. In: NCM 2009. pp. 1102–1107. IEEE Computer Society Press. (2009)