Mind the Gap: Addressing Behavioural Inconsistencies with Formal Methods

J. K. F. Bowles and M. B. Caminati
School of Computer Science, University of St Andrews
Jack Cole Building, North Haugh, St Andrews KY16 9SX, UK
Email: {jkfb|mbc8}@st-andrews.ac.uk

Abstract—**In complex system design, it is important to construct several design models focusing on different aspects of a system to gain a better understanding of individual component structure and behaviour. Scenarios of execution are commonly used to specify partial behaviour and interactions between a group of system objects or components. However, partial specifications may hide inconsistencies or an otherwise unintentionally incomplete or underspecified behavioural model. This paper proposes a new powerful technique combining constraint solvers and theorem provers to complete partial specifications and determine overall model inconsistencies. We use a true-concurrent model, namely labelled event structures, which can be used as the underlying semantics of widely used workflow or scenario-based languages. We show how an interplay between the theorem prover Isabelle and constraint solver Z3 can be used for detecting and solving partial specifications and inconsistencies over event structures.**

I. Introduction

As modern systems become more complex, design approaches model different aspects of a system separately to gain a better understanding of individual component structure and behaviour. It is widely recognised that modelling the complete behaviour of a component is difficult [1], and instead we model several possible scenarios of execution separately. Scenarios give a partial behaviour of a component and include interactions with other system components. In industry, individual scenarios are often captured using UML's sequence diagrams [2]. 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 instead we are interested in modelling (partial) business processes within an organisation, commonly captured using BPMN [3]. In both cases, we need a means to compose models (scenarios or processes). In addition, there may also be a need to correct detected inconsistencies in the resulting model, or make minimal changes leading to an observationally equivalent but reversible model. These issues have been under explored in the literature.

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 [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. 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 [7]. In our recent work [13], [14], [15], 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 for the conjunction of all constraints and denoting the composed model. We used Alloy [16] in [13], [14] and Z3 [17] in [15]. We have conducted several experiments in [15] to show 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 [9], [12], but the use of Z3 is a novelty of [15]. In further recent work, we have also used Z3 for detecting and resolving conflicts between models in a healthcare setting [18].

Our approach in [15] uses event structures [19] as an underlying semantics for sequence diagrams in accordance to [20], [21]. Process languages like BPMN have been given a semantics based on Petri nets [22], where the unfoldings of Petri nets have a direct correspondence to event structures [23]. This paper uses event structures as the underlying semantic model for scenario-based languages or BPMN, and further explores how the theorem prover Isabelle [24] and constraint solver Z3 [17] can be used in a powerful combination for *detecting and solving partial specifications and inconsistencies over event structures*. More concretely, the contributions of this paper include an approach to: (1) fill gaps in a composed model for enabling reversibility of a valid event structure back to a model given in the original scenario-based language, and (2) detect and address label inconsistencies in composed models arising from incompatible constraints over shared variables. The role of Z3 as an SMT solver is essential here since it allows, among others, the manipulation of arithmetic constraints [25]. Even though we will stay mostly at the level of event structures throughout, we stress that our overall aim is to provide a mechanism, applicable to scenario and business process languages, which can be used for automatically generating a correct model composition (if existing), and its enhanced reversible counterpart required to obtain a solution in the original domain language.

This paper is structured as follows. We describe the contributions for the present paper in Section II, and our formal model in Section III. We show how the model is captured in Isabelle and linked to Z3 in Section IV. In Section V, we show the steps involved in generating a valid composition through initial parallel composition (A), extension of the model for reversibility (B) and inconsistent label detection and recovery

(C). We finish the paper with a description of related work in Section VI, and some concluding remarks in Section VII.

II. Context and Contribution

Isabelle [24] 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.

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 [17] provides users an additional powerful combination to be able to produce more proofs automatically.

In this paper, we exploit the interface between Isabelle and Z3 in a novel way 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 (such as required for reversing the model), while being able to prove at any point that the models are valid. If our model contains inconsistent behaviour we are able to locate the conflicting events. In addition, we address inconsistencies in models arising from incompatible (shared) variable constraints and have developed a solution that changes the composed result accordingly. This is best illustrated with a simple example shown as UML sequence diagrams [2].

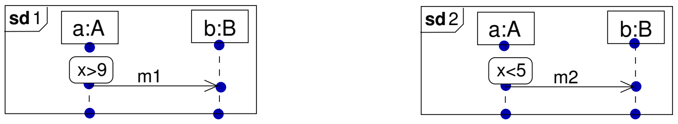


Fig. 1. Two scenarios involving the same object instances.

Fig. 1 shows two extracts of scenarios involving the same instances a and b[1]. In the first case, a sends a message m1 to b, and in the second a sends message m2 to b. The sending of a message is preceded by a so-called *state invariant*, i.e., a constraint over the value of the variable x. When doing

[1]Note that the locations marked along the lifelines are only shown for convenience, as it makes it easier to see the corresponding semantic model later on.

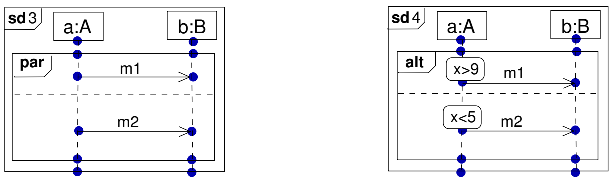


Fig. 2. Usual parallel composition (left) and solution that respects labels (right).

model composition, for scenarios as above, approaches in the literature (including our own recent work) do not treat shared variables, and composition is effectively parallel composition with synchronisation points identified by what is in common (in this case the instances). Such mechanisms construct a composition such as shown in Fig. 2 on the left (which ignores inconsistent labels omitted from the figure), when in fact it should take into account the state invariants and correspond to the model shown in Fig. 2 on the right. Note that **par** and **alt** are operators in sequence diagrams to capture parallel and alternative behaviour respectively. The model in Fig. 2 (left) implies that both messages m1 and m2 are sent, and the order in which they are sent is arbitrary. Alternative fragments may contain guards for the operands, shown here with state invariants. The model in Fig. 2 (right) implies that either m1 or m2 are sent but not both, and in either case only if the state invariant is satisfied. If $5 \leq x \leq 9$, then no message is sent. In this paper, we show formally how we produce a solution first as shown in Fig. 2 on the left, and in case of label inconsistencies resolve it through the use of alternatives as shown on the right.

III. The formal model

The model we use underlying common scenario-based or process languages are labelled (prime) event structures [19], or event structures for short. What appeals about this model is its simplicity and ability to naturally describe fundamental notions present in such high-level languages including sequential, parallel and iterative behaviour (or the unfoldings thereof) as well as nondeterminism (cf. [20], [21]). In an event structure, distributed computations are captured as 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 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. The formal definition as defined for instance in [19] 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., $e\#e^{'} \rightarrow^* e^{''} \Rightarrow e\#e^{''}$ for all $e, e^{'}, e^{''} \in Ev$. Two events $e, e^{'} \in Ev$ are *concurrent*, $e \; co \; e^{'}$ iff $\neg(e \rightarrow^* e^{'} \vee e^{'} \rightarrow^* e \vee e\#e^{'})$.

We assume *discrete* event structures. Discreteness imposes a finiteness constraint on the model, i.e., there is 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.

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. A labelled event structure is a pair $\mathcal{L} = (Ev, \mu)$. The set L can be an arbitrary set depending on the domain of use. Here, labels either denote formulas (constraints over integer variables, e.g., $x \leq 10$ or $y = 5$) or the send/receipt of a message (e.g., (m, s) or (m, r) where m is being sent or received, respectively).

Labelled event structures are often described visually showing only immediate causality and immediate conflict. Consider the following example of an event structure (omitting labels) in Fig. 3. Events e_2 and e_3 are concurrent, events e_6 and e_7 are in conflict (and by propagation also e_6 and e_{10}, and so on). A possible execution fragment in this model corresponds to the set of events $C = \{e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_8\}$.

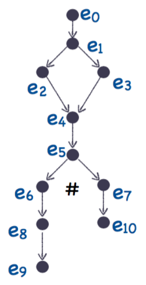


Fig. 3. A simple event structure

Formally, an execution fragment in $\mathcal{L}$ is a subset of events $\sigma \subseteq Ev$ which is (1) *downwards-closed*: if $e \in \sigma$ and $e^{'} \rightarrow^* e$ then $e^{'} \in \sigma$, and (2) *conflict-free*: for all $e, e^{'} \in \sigma$, $\neg(e\#e^{'})$. A trace over $\mathcal{L}$ is a maximal execution fragment. The set C given above is not maximal, but $C \cup \{e_9\}$ is. The event structure has two possible traces.

The parallel composition of two or more models is given by the union of the events keeping the relations as before (cf. citeKue-TCS-06). For the event structures associated to the diagrams from Fig. 1, parallel composition is illustrated in Fig. 4.

The idea here is that the event structure $\mathcal{L}_1 = (E_1, \mu_1)$ associated to **sd 1** from Fig. 1 shown in Fig. 4 (top left) has six events (three per instance). Events e_0, e_2, e_4 correspond to instance a starting the scenario, sending message m_1, and ending the scenario, respectively. Similarly for $\mathcal{L}_2$ associated

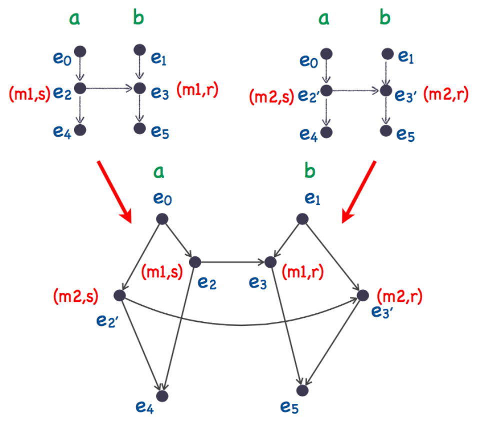


Fig. 4. Parallel composition of event structures

to **sd 2** shown in Fig. 4 (top right). Only immediate causality and partial labels are shown. The only two events with a more interesting label are $\mu_1(e_2) = \{x > 9, (m_1, s)\}$ and $\mu_2(e_2') = \{x < 5, (m_2, s)\}$. Since these events are concurrent they can coexist in a possible trace of execution. This means that we can have an inconsistent state for the system, since both $x > 9$ and $x < 5$ are required and cannot be guaranteed.

When inconsistent event labels are present, we need to add a new pair of events to the conflict relation to prevent the events from ever being part of the same trace. However, this may have a ripple effect on other events and relations. Concretely, adding $e_2'\#e_2$ alone would lead to an invalid event structure since by propagation we obtain $e_4\#e_4$ which is impossible since $\#$ is irreflexive. A usual solution is to duplicate any shared event e' that causally succeeds the added conflicting event pair (in this case only e_4 and e_5) as shown in Fig. 5. It

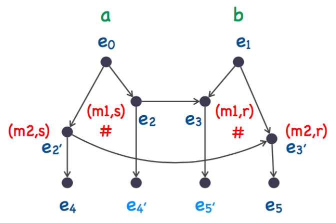


Fig. 5. Adding conflict to inconsistent labels

should be noted that the event structures in Fig. 4 and Fig. 5 do not correspond to the ones we obtain with the automated transformation defined in [15] when applied to the sequence diagrams of Fig. 2. We need to add events as shown in Fig. 6 to have a direct correspondence to Fig. 2 on the left (similarly for the other case).

In general, given two (or more) scenarios that have been transformed into event structures, we do the following steps automatically: construct the parallel composition as before, find conflicting labels if present and reconstruct a valid model in that case, extend the model (to denote fragment boundaries) to enable reversing the model to the original domain. Model

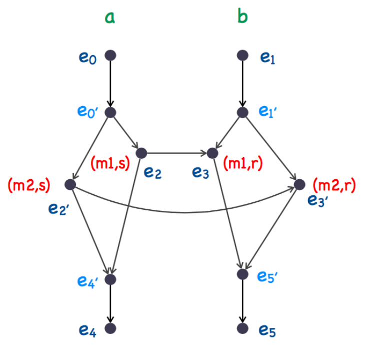


Fig. 6. Parallel composition of event structures with added events for reversibility

inconsistencies may be found during the first two steps. We show how these steps can be followed and how correctness can be ensured throughout with our tool combination in the next sections. Actually reversing the model to the original domain is not in the scope of the present paper.

IV. Using Isabelle and Z3

Specifying event structures in Isabelle is straightforward. We need to define relations for causality `Ca` and conflict `Co`, and their properties. For instance, `Ca` is a partial order (reflexive, antisymmetric and transitive) and `Co` is irreflexive, symmetric and propagates over causality. The transitivity over `Ca`, referred to by `Trans`, is given below as an example.

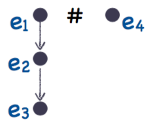
```
abbreviation "Trans Ca == ∀ x y z.
        (Ca x y & Ca y z → Ca x z)".
```

Further, the following allows us to talk about an event structure satisfying all required properties.

```
abbreviation "IsLes Ca Co == (Reflex Ca &
  Antisym Ca & Trans Ca &
  Irrefl Co & Sym Co & Propagation Co Ca)".
```

In other words, `IsLes` is a higher-order function returning whether `Ca` and `Co` form a valid event structure. Given this definition (which conforms to Definition 1), we can take advantage of Isabelle's built-in Z3 code generator to exploit Z3's powers for checking whether a given model is a valid event structure[2]. As mentioned earlier, establishing whether a given structure is indeed an event structure is important in the context of model composition.

Consider a particularly simple example: event e_1 causes event e_2, which in turn causes e_3, and e_1 is in conflict with a further event e_4 as shown below.



The idea is to challenge Z3 to find a proof that such a structure is *not* a valid event structure. The following theorem in Isabelle shows what we want to prove:

[2]Since Z3 adheres to the SMT-LIB standard, everything we will say about Z3 will also apply verbatim to other SMT solvers, like cvc4.

```
theorem assumes "IsLes Ca Co" and "Ca e1 e2"
  and "Ca e2 e3" and "Co e1 e4" shows False
sledgehammer run [provers=z3, minimize=false,
  overlord=true, timeout=1] (assms)
```

The theorem above makes some assumptions (hypotheses) written after the keyword `assumes`[3]. 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 `Ca` and `Co`, and assertions for each of the stated hypotheses. The underlying idea is that Isabelle is trying to prove the theorem by contradiction and passes to Z3 all the hypotheses and the negation of the thesis (which here would be `True`). When running Z3 on the input produced by Isabelle, Z3 returns `sat` showing that all the assumptions are satisfiable (it is a valid event structure). Whenever we get a `sat` answer from Z3, we can ask it to provide a model by appending the command `(get-model)` to the generated file.

It should be noted that, to check whether a given model is an event structure or whether it can be completed (by adding pairs to the causality and conflict relations) are essentially the same problem for Z3. Z3 returns (if existing) a model satisfying the given assertions, and any additional derived relation pairs are automatically inferred.

Conversely, we use the same mechanism for finding a problem in a model. Suppose that the model we are given is not a valid event structure, we would like Z3 to detect where the problem lies. To make it easier to locate the problem, it is better to use more fine-grained assumptions. Consider a similar example model and theorem, where we state individual assumptions of a valid event structure explicitly[4].

```
lemma assumes "Reflex Ca" "Antisym Ca"
  "Trans Ca" "Irrefl Co" "Sym Co"
  "Propagation Co Ca" "Ca e1 e2" "Ca e2 e3"
  "Co e1 e4" "Ca e1 e4" shows False
sledgehammer run [provers=z3, minimize=false,
  overlord=true, timeout=1] (assms)
```

Notice that the only difference between this model and the previous one, is an added causality relation between events e_1 and e_4 given by the last hypothesis `Ca e1 e4`. Since we broke the hypotheses further, we will get distinct labelled assertions in the Isabelle-generated Z3 file (see extract listing below). The labels, automatically added by Isabelle, help us to identify which events and/or properties are violated.

```
(assert (! (not false):named a0))
(assert (! (ca$ e1$ e4$):named a1))
(assert (! (co$ e1$ e4$):named a2))
```

[3]The `ands` are optional, added here for clarity, but omitted in subsequent snippets.

[4]The keywords `theorem`, `lemma`, and `corollary` are all the same for Isabelle.

```
(assert (! (ca$ e2$ e3$):named a3))
(assert (! (ca$ e1$ e2$):named a4))
(assert (! (forall ((?v0 A$)(?v1 A$)(?v2 A$))
     (=> (and (co$ ?v0 ?v1) (ca$ ?v1 ?v2))
         (co$ ?v0 ?v2))):named a5))
(assert (! (forall ((?v0 A$) (?v1 A$))
     (=> (co$ ?v0 ?v1) (co$ ?v1 ?v0)))
         :named a6))
(assert (! (forall ((?v0 A$))
           (not (co$ ?v0 ?v0))):named a7))
(assert (! (forall ((?v0 A$)(?v1 A$)(?v2 A$))
     (=> (and (ca$ ?v0 ?v1) (ca$ ?v1 ?v2))
         (ca$ ?v0 ?v2))):named a8))
(assert (! (forall ((?v0 A$) (?v1 A$))
     (=> (and (ca$ ?v0 ?v1) (ca$ ?v1 ?v0))
         (= ?v0 ?v1))):named a9))
(assert (! (forall ((?v0 A$))
     (=> (exists ((?v1 A$))
         (or (ca$ ?v0 ?v1) (ca$ ?v1 ?v0)))
             (ca$ ?v0 ?v0))):named a10))
(check-sat)
(get-unsat-core)
```

Running `z3 unsat_core=true` on the obtained file we get:

```
unsat (a1 a2 a5 a6 a7)
```

Besides the unsatisfiable outcome, we know that the problem is caused by simultaneously imposing causality between e_1 and e_4 (a1), conflict between e_1 and e_4 (a2), propagation condition (a5), conflict symmetry (a6), and conflict irreflexivity (a7). In other words, from $e_1 \to e_4$, $e_1 \# e_4$ and propagation of conflict over causality, we get $e_4 \# e_4$ which is impossible since conflict is irreflexive.

V. Enhanced composition mechanisms

A. Basic parallel composition

We show here how parallel composition can be obtained from Isabelle automatically as expected. We use our example from earlier (cf. Fig. 4), with individual event structures given by:

```
abbreviation "Ca1 == {(e0,e2), (e1,e3),
  (e2,e3), (e2,e4), (e3,e5)}"
abbreviation "Ca2 == {(e0,e2'), (e1,e3'),
  (e2',e3'), (e2',e4), (e3',e5)}"
```

We only need to specify immediate causality (the remaining causality pairs are generated automatically), and neither structure has events in conflict. We are looking for a composition `Ca` which includes both event structures and satisfies the property `isLes`. We formulate a lemma that attempts to prove that this is impossible:

```
lemma shows
"¬ (∃ Ca. (events Ca=events Ca1 ∪ events Ca2 &
Ca ⊇ Ca1 ∪ Ca2 & isLes Ca {}))"

nitpick [card eventType2=1-50, max_potential=0].
```

The command `nitpick` challenges Isabelle to find a counterexample to this lemma, which, if found, gives us the expected composition. In our case, the output is:

```
Nitpicking formula...
Nitpick found a counterexample:
Skolem constant:
Ca = {(e0,e0),(e0,e2),(e0,e3),(e0,e4),(e0,e5),
(e0,e2'),(e0,e3'),(e1,e1),(e1,e3),(e1,e5),
(e1,e3'),(e2,e2),(e2,e3),(e2,e4),(e2,e5),
(e3,e3),(e3,e5),(e4,e4),(e5,e5),(e2',e4),
(e2',e5),(e2',e2'),(e2',e3'),(e3',e5),(e3',e3')}
```

which corresponds exactly to our composed event structure shown in Fig. 4 (note that it generates the complete set of pairs of events related by causality).

B. Extending a model for reversibility

In order to be able to reverse the obtained event structure, we need, however, to add a few events to the model which correspond to the beginning/ending of the parallel fragments in a sequence diagram. Note that this step may be carried out after parallel composition was obtained (done in 5.1), or after checking for inconsistent labels (done in 5.3). We introduce it at this point in the paper, because it is reasonably straightforward, and may in addition be useful to understand our approach for detecting and resolving inconsistent labels.

To extend a model for reversibility, we first need to extract from the causality relation the pairs of events in immediate causality. Mathematically, this means to go from a partial order to the corresponding covering relation, which is possible since our structures are discrete.

```
abbreviation "order2strictCover P ==
Union {{x}×(next1 P {x})|x.x∈(Domain P)}"
```

We also introduce explicitly the concurrency relation as follows.

```
abbreviation "concurrency Ca Co ==
    ((events Ca)×(events Ca))-Ca-(Ca⁻¹)-Co"
```

This states that two events are concurrent, if they are not related by causality (where $\texttt{Ca}^{-1}$ is used to denote the reverse of `Ca`) or in conflict. We need to add events only to the concurrent branches created by the composition, hence:

```
abbreviation "newConc ==
  (concurrency composedCa {} -
    concurrency Ca1 {} - concurrency Ca2 {})"
```

For each new pair of concurrent events, we now want to check whether they have a common immediate successor (respectively, predecessor).

```
abbreviation "divergingTriangles == ((λ (x,y).
({x,y}, predecessors
  (order2strictCover composedCa){x} ∩
 predecessors
  (order2strictCover composedCa){y}))`newConc)"

abbreviation "convergingTriangles == ((λ (x,y).
({x,y}, successors
  (order2strictCover composedCa) {x} ∩
 successors
  (order2strictCover composedCa) {y}))`newConc)"
```

Above, `successors R X` gives us the set of successors of any of the elements of `X` according to the order relation `R` (similarly for `predecessors`). The definitions above select all the triples consisting of two concurrent events with a common immediate successor or predecessor, but we still need to remove empty sets. We omit the rules for that here since they are not essential for what follows.

Finally, we insert a new event between each new pair of concurrent events in the composed model and their common immediate successor or predecessor, obtaining a new causality relation:

```
definition "extendedComposedCa = composedCa ∪
Union ((λ (X, Z). Z × freshEventGenerator ‘‘Z ∪
(freshEventGenerator ‘‘Z) × X)‘
  divergingTrianglesNonDegenerate) ∪
(Union ((λ (X, Z). X × freshEventGenerator ‘‘Z ∪
(freshEventGenerator ‘‘Z) × Z)‘
  convergingTrianglesNonDegenerate))"
```

Here, `freshEventGenerator` is a function to associate to events in the original `composedCa` new and distinct events.

Let `composedCa` be the result obtained by Nitpick at the end of last section for our running example and corresponding to Fig. 4. The extended model is ready for being reversed corresponding to Fig. 6. The following

```
value "order2strictCover extendedComposedCa"
```

produces the intended solution below.

```
"{(e4',e4),(e5',e5),(e3',e5'),(e2',e4'),
  (e2',e3'),(e3,e5'),(e2,e4'),(e2,e3),
  (e1',e3'),(e1',e3),(e1,e1'),(e0',e2'),
  (e0',e2),(e0,e0')}"
```

Above we used `order2strictCover` defined earlier to prune the output and make it more readable.

C. Dealing with conflicting labels using Z3

We now consider event labels and check whether events are consistent with respect to their labels. For example, we might have one event e_1 with a label stating a condition over a boolean variable that must be `true`, and another event e_2 for which the same variable must be `false`. More generally, e_1 could have a label corresponding to `x<10`, and e_2 a label corresponding to `x>15`, where `x` is the same numerical variable. In both cases events e_1 and e_2 are inconsistent. For simplicity, we will look at the consistency of pairs of events at a time, but the approach is extensible for any finite number of events.

The problem we want to address is formally expressed as follows. We are given two labelled event structures ($\mathcal{L}_1$ and $\mathcal{L}_2$), where event labels may include arithmetic formulas over a shared set of variables $Var = \{x_0, \ldots, x_{n-1}\}$ for $n \in \mathbb{N}$ over the domain of integers. A formula may contain any arithmetic operation, comparisons, boolean operators, and, for each possible variable evaluation, returns a boolean value. We want to check for all pairs of events (e_1, e_2) with $e_1 \in Ev_1$ and $e_2 \in Ev_2$, whether $\mu_1(e_1)$ and $\mu_2(e_2)$ are simultaneously satisfiable for some evaluation of the variables occurring in the labels. Since this is essentially a satisfiability problem, we exploit Isabelle's built-in Z3 code generator to obtain an answer.

In HOL, we can represent this problem by introducing functions f_1 and f_2 associating to each event in Ev_1 and Ev_2, respectively, a λ-abstracted function over Var. To illustrate the idea, consider the following sets of events $Ev_1 = \{e_0, e_1, e_2\}$ and $Ev_2 = \{e'_0, e'_1, e'_2\}$, and corresponding labels given by:

$$\begin{array}{lrlr} e_0: & x_0 < 5 \; \& \; x_0 < x_1 & e'_0: & x_0 < 3 \; \& \; x_0 < x_1 \\ e_1: & x_0 > x_1 & e'_1: & x_0 + 1 > x_1 \\ e_2: & x_0 + x_1 > 5 & e'_2: & x_0 + x_1 < 11 \end{array}$$

The following statements encode λ-abstractions to reflect all possible evaluations of the variables that satisfy the labels given.

Listing 1. labels expressed in HOL

```
"f1 (map1 e0) = (λ x0 x1. x0 < 5 & (x0 < x1))"
"f1 (map1 e1) = (λ x0 x1. x0 > x1)"
"f1 (map1 e2) = (λ x0 x1. x0 + x1 > 5)"
"f2 (map2 e0') = (λ x0 x1. x0 < 3 & (x0 < x1))"
"f2 (map2 e1') = (λ x0 x1. x0 + 1 > x1)"
"f2 (map2 e2') = (λ x0 x1. x0 + x1 < 11)"
```

Above, `map1` and `map2` are bijections mapping events to integers, which is needed in order to interface our events of arbitrary type with Z3. This is because Z3 performs best when dealing with integers, a native type. We can, nonetheless, ignore this internal mapping done for convenience, and keep our models with symbolic event types.

We now want Z3 to identify which pairs of events are satisfiable and which are not. To this end, we introduce a function g taking two events and yielding the answer. Then, we need to phrase our query in the form of a satisfiability problem: the idea is to introduce a formula which is always satisfiable, and in which $g\ e\ e'$ is true iff e and e' are simultaneously satisfiable. We can obtain this behaviour by the following disjunction:

```
((∃ x0 x1. ((f1 e1) x0 x1 & (f2 e2) x0 x1)) &
  (g e1 e2 = True)) ∨
((¬(∃ x0 x1.((f1 e1) x0 x1 & (f2 e2) x0 x1)))&
  (g e1 e2 = False))
```

This means that, if both the formulas associated to a pair of events are satisfiable for some variable assignment, g is true for those events, otherwise g is false. Such a g always exists, and can be found by Z3 giving us exactly what we want to find out. Since we want to see Z3's output in term of symbolic event types, rather than integers, we finally introduce a further map h to translate from integers to the symbolic event identifiers (e_0, e'_0, etc...). This corresponds to a further requirement between g and h:

```
"∀ e e'. h e e' = g (map1 e) (map2 e')"
```

All the statements above constitute the assumptions for a new lemma on which we can invoke `sledgehammer`. In this lemma, we also need the trivial assumptions to make `map1` and `map2` bijections, which we omit here.

The automatically generated Z3 file presents one labelled `assert` statement for each of the lemma's hypotheses. We do not include an extract of it below as it is not essential to understand what is happening, and the output is similar to examples shown earlier. Running Z3 on the result with an explicit request for `get-model`, we get the answer, of which we give below the relevant extract.

```
(define-fun h$ ((x!1 EventType3$)
                (x!2 EventType4$)) Bool
  (ite (and (= x!1 e1$a) (= x!2 e1$)) true
  (ite (and (= x!1 e2$a) (= x!2 e0$)) true
  (ite (and (= x!1 e0$a) (= x!2 e2$)) true
  (ite (and (= x!1 e1$a) (= x!2 e0$)) false
  (ite (and (= x!1 e0$a) (= x!2 e1$)) false
  (ite (and (= x!1 e2$a) (= x!2 e2$)) true
  (ite (and (= x!1 e1$a) (= x!2 e2$)) true
  (ite (and (= x!1 e2$a) (= x!2 e1$)) true
    (g$ (map1$ x!1) (map2$ x!2))))))))))))
```

Essentially, we obtain that h is false over events e_1 and e'_0, and events e_0 and e'_1, as expected.

This general mechanism can be applied to our example of Figure 4, where two events have labels consisting of sets of incompatible inequalities: $\mu_1(e_2) = \{x > 9, (m_1, s)\}$ and $\mu_2(e'_2) = \{x < 5, (m_2, s)\}$. We start by translating them as done in listing 1, that is, the corresponding statement for the event e_2 will be

```
"f1 (map1 e2) = (λ x. x > 9)"
```

and similarly for the others.

By generating the Z3 input file and running Z3 on it, we are given the answer that e_2 and e'_2 are not compatible. To resolve this, we want to add a new conflict relation between these events. Further, we want to obtain any other required changes in the relations automatically. We use Nitpick, similarly to what we have done before.

```
lemma assumes "Ca1 ∪ Ca2 − C ⊆ Ca3" "card C<=2"
"(e2, e2') ∈ Co3" "(e2, e3) ∈ Ca3"
"(e2', e3') ∈ Ca3" shows "¬ (isLes Ca3 Co3)"
 nitpick [card eventType2=8−14,
   timeout=40, max_potential=0]
```

The lemma above challenges Nitpick to find a counterexample event structure $\mathcal{L}_3$ (with relations $Ca3$ and $Co3$) made by removing at most two pairs from $Ca1 \cup Ca2$, in such a way that e_2 and e_3 are in conflict in the new event structure. Nitpick answers by finding a counterexample, (i.e., $\mathcal{L}_3$ exists), building it, and formally proving it is a valid event structure.

```
Nitpicking formula...
Nitpick found a counterexample:

  Free variables:
    C = {(e2, e4), (e3, e5)}
    Ca3 = {(e0, e0), (e0, e2), (e0, e3),
 (e0, e4), (e0, e5), (e0, e2'), (e0, e3'),
 (e1, e1), (e1, e3), (e1, e5), (e1, e3'),
 (e2, e2), (e2, e3), (e3, e3), (e4, e4),
 (e5, e5), (e2', e4), (e2', e5), (e2', e2'),
 (e2', e3'), (e3', e5), (e3', e3')}
    Co3 = {(e2, e4), (e2, e5), (e2, e2'),
 (e2, e3'), (e3, e4), (e3, e5), (e3, e2'),
 (e3, e3'), (e4, e2), (e4, e3), (e4, e5),
 (e4, e3'), (e5, e2), (e5, e3), (e5, e4),
 (e2', e2), (e2', e3), (e3', e2), (e3', e3),
 (e3', e4)}
```

Since Nitpick removed for us the causality pairs from Figure 4 relating $e_2 \rightarrow e_4$ and $e_3 \rightarrow e_5$, we just need to re-introduce causality pairs from e_2 and e_3 to the new events e'_4 and e'_5 respectively:

```
abbreviation "FinalCa == addEventToCausality
  (addEventToCausality Ca3 e2 e4') e3 e5'"
abbreviation "FinalCo == addEventToConflict
  (addEventToConflict Co3 e2 e4') e3 e5'"
```

Above, `addEventToCausality` and `addEventToConflict` are generic functions defined in the way to preserve the property of being an event structure, and this is formally granted by an Isabelle theorem that we proved:

```
theorem lm30: assumes "e' ∈ events Ca"
"isLes Ca Co" "e ∉ events Ca" "e ∉ events Co"
shows "isLes (addEventToCausality Ca e' e)
             (addEventToConflict Co e' e)".
```

By applying this theorem twice, we obtain that $(FinalCa, FinalCo)$ is an event structure. At the same time, all the definitions used up to now are formulated in a way to preserve the computability so that, for example, we can query Isabelle as follows:

```
value "order2strictCover FinalCa",
```

obtaining the answer

```
"{(e3', e5), (e2', e3'), (e2', e4), (e0, e2'),
  (e0, e2), (e1, e3'), (e1, e3), (e2, e3),
  (e2, e4'), (e3, e5')}"
  :: "(eventType2 × eventType2) set",
```

which corresponds to Figure 5, as intended.

VI. Related work

The use of SAT solvers such as Alloy, as an aid to generating composition of models is not new. Zhang et al. [12] and Rubin et al. [9] have used Alloy for the composition of UML class diagrams, but their approaches are not fully automated and they only address composition of static models. Widl et al. [11] use of SAT-solvers to compose concurrently evolved sequence diagrams with respect to an overall behavioural specification given as a state machine. Liang et al. [8] present a method of integrating sequence diagrams based on the formalisation of sequence diagrams as typed graphs. Both [11], [8] focus on less complex structures. For example, they do not deal with structured interactions which can introduce considerable complexity. In recent work, we have looked at modelling and composing more complex scenarios in a fully automated approach (e.g. [14]). Later, we replaced Alloy by the SMT solver Z3 in [15] for performance and scalability reasons only. However, the fact that SMT solvers extend SAT solvers and are able to deal with more complex problems including arithmetic constraints, means that we can express and deal with much more complex models and constraints too. A good example of where an SMT solver has proven to be particularly useful due to the integer arithmetics capabilities is given in [18], where Z3 is used to detect and resolve (medication) conflicts between (pharmaceutical) graphs in a medical context. Nonetheless, for modelling and building composed models automatically, no known approach has attempted to address and explore inconsistencies derived from variable conditions as we do here. We also have a more general framework based on the theorem prover Isabelle, which gives us further assurance as to the correctness of all our steps. To the best of our knowledge, this is both novel and powerful. 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 specifying and reasoning about complex behaviour.

VII. Conclusion

The integrated approach proposed in this paper allows us to gather tasks, which are typically either done manually or through several steps involving distinct tools, into a fully automated approach involving one formal tool, namely Isabelle. Isabelle is a theorem prover that can be used toensure rigorous definitions and theorems throughout, and can be used to compute event structures (in our case), either directly or by calling internally additional tools such as the SMT solver Z3 and nitpick. Through the use of Z3 and its ability to handle arithmetic constraints, we were able to extend earlier work to address the detection of conflicts between models involving shared variables and inconsistent states. Our work in [15] is the first approach known to us that uses Z3 for the composition of behavioural models. This paper is the first to address arithmetic conflicts in models through a powerful combination of the theorem prover Isabelle and Z3, and the first to exploit Isabelle's infrastructure to interface the final user with SMT solvers: indeed, Isabelle's smtlib2 translator is used internally to use SMT solvers as theorem provers, and not conceived to be exposed to the user. We show how this Isabelle feature can be more useful than that, by allowing the user to avoid error-prone first-order logic syntax, and to use the highly expressive HOL to both ease the definitions and prove their correctness. Furthermore, this allows combining SMT techniques with other means for computations, like HOL functional definitions and Nitpick, thus addressing distinct parts of the problem at hand through the most suitable approach.

Finally, we are working on further improvements to our automated transformations that map scenario or process-based language models to our formal model, and further enables the full reversibility of the transformation back to the original domain model. Only fully automated reversible approaches can make a more definite contribution as they have the potential of being accepted in industry.

Acknowledgment

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

References

[1] S. Uchitel, G. Brunet, and M. Chechik, "Synthesis of partial behavior models from properties and scenarios," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 384–406, 2009.

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

[3] ——, *Business Process Model and Notation. Version 2.0*, OMG, http://www.omg.org., 2011, document id: formal/2011-01-03.

[4] J. Araújo, J. Whittle, and D. Kim, "Modeling and composing scenario-based requirements with aspects," in *RE 2004*. IEEE Computer Society Press, 2004, pp. 58–67.

[5] J. Bowles and B. Bordbar, "A formal model for integrating multiple views," in *ACSD 2007*. IEEE Computer Society Press, 2007, pp. 71–79.

[6] R.Reddy, A. Solberg, R.France, and S. Ghosh, "Composing sequence models using tags," in *Proc. of MoDELS Workshop on Aspect Oriented Modeling*, 2006.

[7] J. Klein, L. Hélouët, and J. Jézéquel, "Semantic-based weaving of scenarios," in *AOSD'06*. ACM, 2006, pp. 27–38.

[8] H. Liang, Z. Diskin, J. Dingel, and E. Posse, "A general approach for scenario integration," in *MoDELS 2008*, ser. LNCS 5301. Springer, 2008, pp. 204–218.

[9] J. Rubin, M. Chechik, and S. Easterbrook, "Declarative approach for model composition," in *MiSE 2008*. ACM, 2008, pp. 7–14.

[10] J. Whittle, J. Araújo, and A. Moreira, "Composing aspect models with graph transformations," in *Proceedings of the 2006 international workshop on Early aspects at ICSE*. ACM, 2006, pp. 59–65.

[11] M. Widl, A. Biere, P. Brosch, U. Egly, M. Heule, G. Kappel, M. Seidl, and H. Tompits, "Guided merging of sequence diagrams," in *SLE 2012*, ser. LNCS 7745. Springer, 2013, pp. 164–183.

[12] D. Zhang, S. Li, and X. Liu, "An approach for model composition and verification," in *NCM 2009*. IEEE Computer Society Press., 2009, pp. 1102–1107.

[13] J. Bowles, M. Alwanain, B. Bordbar, and Y. Chen, "Matching and merging scenarios automatically with Alloy," in *Model-Driven Engineering and Software Development*, ser. CCIS 506, S. H. et al., Ed. Springer, 2015, pp. 100–116.

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

[15] ——, "Weaving true-concurrent aspects using constraint solvers," in *Application of Concurrency to System Design (ACSD 2016)*. IEEE Computer Society Press, June 2016.

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

[17] L. D. Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS 2008*, ser. LNCS 4963. Springer, 2008, pp. 337–340.

[18] A. Kovalov and J. K. F. Bowles, "Avoiding medication conflicts for patients with multimorbidities," in *iFM 2016*, ser. LNCS 9681. Springer, 2016, pp. 376–392.

[19] G. Winskel and M. Nielsen, "Models for Concurrency," in *Handbook of Logic in Computer Science, Vol. 4, Semantic Modelling*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Oxford Science Publications, 1995, pp. 1–148.

[20] J. Küster-Filipe, "Modelling concurrent interactions," *Theoretical Computer Science*, vol. 351, pp. 203–220, 2006.

[21] J. K. F. Bowles, "Decomposing Interactions," in *AMAST 2006*, ser. LNCS 4019. Springer, 2006, pp. 189–203.

[22] R. Dijkman, M. Dumas, and C. Ouyang, "Semantics and analysis of business process models in bpmn," *Information and Software Technology*, vol. 50, pp. 1281–1294, 2008.

[23] M. Nielsen, G. Plotkin, and G. Winskel, "Petri nets, event structures and domains, part i," *TCS*, vol. 13, pp. 85–108, 1981.

[24] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, ser. LNCS 2283. Springer, 2002.

[25] L. D. Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, 2011.