Automated Error-Detection and Repair for Compositional Software Specifications

Dalal Alrajeh and Robert Craven

Department of Computing, Imperial College London, UK
{dalal.alrajeh,robert.craven}@imperial.ac.uk

Abstract. The complexity of error diagnosis in requirements specifications, already high, is increased when requirements refer to various system components, on whose interaction the system's aims depend. Further, finding causes of error, and ways of overcoming them, cannot easily be achieved without a systematic methodology. This has led researchers to explore the combined use of verification and machine-learning to support automated software analysis and repair. However, existing approaches have been limited by using formalisms in which modularity and compositionality cannot be explicitly expressed. In this paper we overcome this limitation. We define a translation from a representative process algebra, Finite State Processes, into the action language $\mathcal{C}+$. This enables forms of verification not supported by previous methods. We then use a logic-programming equivalent of $\mathcal{C}+$, to which we apply inductive logic programming for learning repairs to system components while ensuring no new errors are introduced and interactions with other components are maintained. These two phases are iterated until a correct specification is reached, enabling rigorous and scalable support for automated analysis and repair of component-based specifications.

1 Introduction

Research into formal specification, verification and error diagnosis has played a significant role in improving software safety and reliability. Such methods rely on specifying the system in a formal language (e.g., temporal logic, process algebras) and using automated verification techniques such as model checking and theorem proving to check that the specified system satisfies some given property. Though such methods are useful for detecting errors in software specifications (e.g., [14]), identifying the exact causes of error and resolving them is a very difficult task that is mostly performed manually—defeating the aim of automation, and increasing the likelihood of error.

In recent years researchers in software engineering have responded to this by deploying a combination of verification and machine learning techniques to improve software specifications. For example, in [1] the authors describe a method for incrementally refining a consistent specification, expressed in first-order temporal logic, with respect to some given property using an integration of model checking and Inductive Logic Programming (ILP). In [2], the authors give a method for revising temporal specifications that may be incorrect or inconsistent using model checking and artificial neural networks. Such advances overcome some of the difficulties of generating alternative candidate repairs to detected errors, ensuring consistency of the computed solutions with the available specification and property. However, a very significant drawback of such approaches is that verification and specification improvement are at system level only: they do not relate the specification to the individual system components, nor do they support compositional analysis. With this drawback come the familiar problems of modularity, scalability and realizability. (Realizability means that suggested repairs can be assigned to and achieved by individual components.) Furthermore, these approaches either require the engineer to specify an action thought to have produced the error in the output violation run; or require the engineer to simplify the diagnosis procedure by assuming that the last action in the run is the cause.

In this paper we propose a new approach for incrementally detecting errors in and repairing compositional software specifications using verification and ILP. Our framework: (i) supports error-diagnosis and repair at component level rather than system level; (ii) diagnoses multiple errors in a single iteration; (iii) can hypothesize faults at any point in a violation run, and fix them wherever they are; (iv) finds all minimal repairs with respect to a given input language; (v) guarantees deadlock-free repairs to components consistent with the original specification; and (vi) is fully automated.

Our systems specifications are given in Finite State Processes (FSPs), a well-studied process algebra [20]. FSPs enable a user to represent the behaviour at the architectural level, specifying the system in a modular manner as a composition of processes executed concurrently and interacting with each other through shared actions. FSPs contain operations common to most algebraic languages and are supported by the model checker LTSA [20]. We show how FSP descriptions can be formulated in the action language $\mathcal{C}+$ [12] (from non-monotonic reasoning in A.I.) and its corresponding logic programming representation, $\mathcal{EC}+$ [6] which are the languages used by the verification and learning tasks respectively. $\mathcal{C}+$ is a natural choice: similarly to FSPs, it has a semantics of LTSs and allows concise representation of domains. It also supports many forms of reasoning, including the computation of all runs of a given length that satisfy a given description, and the construction complex queries over runs, states and transition of processes. We describe a systematic translation of $\mathcal{C}+$ into logic programs, where the resulting logic programs allows us to deploy ILP in the discovery of repairs.

The paper is structured at follows. §2 gives background, and §3 our running example. In §4 we describe verification using $\mathcal{C}+$. §5 presents the use of $\mathcal{EC}+$ and ILP to correct FSP descriptions. Related work and a conclusion follow in §6.

2 Background

Labelled Transition Systems (LTSs)

LTSs [15] are behaviour models representing the changing states of a system, in response to actions occurring within or outside the system. Both FSPs and $\mathcal{C}+$ use LTSs in their semantics. An LTS L is a structure (S, A, Δ, S_0), where S is a finite set of *states*, A is a finite set of *action labels* (also known as the *alphabet*), $\Delta \subseteq S \times A \times S$ is the *transition* relation, and S_0 is a set of *initial states*. An LTS is *deterministic* iff for each $s \in S$ and each action label $a \in A$, there is at most one state s' for which $(s, a, s') \in \Delta$. It is called *deadlock-free* if for each $s \in S$ reachable from an initial state in S_0, there is at least one state s' for which $(s, a, s') \in \Delta$. We use $s \xrightarrow{a} s'$ as a shorthand for $(s, a, s') \in \Delta$. A *run of length* n through an LTS (S, A, Δ, S_0) is a sequence $(s_0, a_0, s_1, \ldots, a_{n-1}, s_n)$ such that $s_0 \in S_0$ and for all i with $0 \leqslant i < n$, $(s_i, a_i, s_{i+1}) \in \Delta$. We often write such runs as expressions $(s_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{n-1}} s_n)$.

Where several LTSs represent the individual components of a larger system, the individual LTSs can be composed together by synchronising the actions common to their alphabets and interleaving the remaining actions. To denote the composed behaviour of two LTSs $P = (S_P, A_P, \Delta_P, S_{P,0})$ and $Q = (S_Q, A_Q, \Delta_Q, S_{Q,0})$ we use the commutative and associative binary parallel composition operator, $\|$. The LTS $(P \parallel Q)$ is the structure $(S_P \times S_Q, A_P \cup A_Q, \Delta, S_{P,0} \times S_{Q,0})$ such that $(s_p, s_q) \xrightarrow{a} (s'_p, s'_q)$ iff either: (i) $a \in A_P \setminus A_Q$ and $s_p \xrightarrow{a} s'_p$ and $s_q = s'_q$; (ii) $a \in A_Q \setminus A_P$ and $s_q \xrightarrow{a} s'_q$ and $s_p = s'_p$; or (iii) $a \in A_P \cap A_Q$ and $s_p \xrightarrow{a} s'_p$ and $s_q \xrightarrow{a} s'_q$. (i) and (ii) represent independent execution of P and Q; (iii) gives joint execution of a shared action.

Finite State Processes (FSPs)

FSPs [20] are process algebras, based on CSP [13] and CCS [22], for describing the behaviour of components of concurrent systems. Each component is represented as a *primitive process*, comprising a number of *local processes*, which can be thought of as phases of the component's operation. The scope of a local process is the primitive process in which it is defined. A *composite process* represents the composition of a set of primitive processes. The signature of a primitive process includes a process name, names of its local processes and a set of action labels A, the *alphabet*. Primitive process operators include '$\rightarrow$' for action prefixing (showing which actions can be performed to lead to another local process) and '$|$' for choice (more than one possible action).

The language also allows for definitions of constants, integer ranges, sets of action labels. In the current paper we focus on the basic syntax for FSPs. We refer to it here as *fundamental FSPs* and define it in §4. Each 'full' FSP has an equivalent formulation in terms of fundamental FSPs—so there is no loss of expressivity [20].

A composite process represents the composition of a number of primitive processes. Similarly to LTSs, the operator $\|$ is used to denote the composed behaviour of two processes. The expression $P \parallel Q$ means that the P and Q may execute actions independently but must synchronise actions common to their alphabets. A composite process is identified by a process name preceded by the symbol "$\|$". In §4 we will describe the precise syntax and semantics of the restricted form of FSPs we work with.

$\mathcal{C}+$ and $\mathcal{EC}+$

Action languages [11] are logical formalisms for representing the way systems change as a consequence of actions and events occurring in them.

The language of $\mathcal{C}+$ is built over a *multi-valued propositional signature* σ, with *fluent constants* σ^{f}—describing states—and the *action constants* σ^{a}—describing actions and events. An *atom* $c{=}v$ has $c \in \sigma$ and $v \in dom(c)$—the non-empty *domain* of c's *values*. An *interpretation* I maps constants to values; we write $I \models c{=}v$ iff $I(c) = v$. $\mathrm{I}(\sigma)$ is the set of interpretations of the signature σ. A *fluent formula* is built from Boolean connectives using fluent atoms ($c{=}v$ where $c \in \sigma^{\mathrm{f}}$) and $\bot$ and $\top$ (for logical truth); an *action formula* is made from atoms containing only action constants, with $\top$, and must contain at least one action constant. The LTSs (S, A, Δ, S_0) used in the semantics of $\mathcal{C}+$ have $S \subseteq \mathrm{I}(\sigma^{\mathrm{f}})$ and $A = \mathrm{I}(\sigma^{\mathrm{a}})$; S_0 is S, and $\Delta \subseteq S \times A \times S$.

Causal laws determine the states S and transition relation Δ. A *static law* has the form F **if** G, where F and G are fluent formulas: if a state satisfies G, it must also satisfy F. A *fluent dynamic law* has the form F **if** G **after** ψ, where F and G are fluent formulas, and ψ is any formula with signature $\sigma^{\mathrm{f}} \cup \sigma^{\mathrm{a}}$. This means: for any transition (s, a, s'), if $s \cup a \models \psi$ and $s' \models G$, then $s' \models F$. (If G is $\top$, then we abbreviate the rule as F **after** ψ.) Finally, an *action dynamic law* is an expression α **if** ψ, where α is an action formula and ψ is as above. These mean: if $s \cup a \models \psi$, then $a \models \alpha$.

An *action description* is a set of causal laws; each defines an LTS (S, A, Δ, S_0). A law **inertial** c means the value of fluent constant f persists by default; **exogenous** a means (roughly) that a can be executed, or not executed, in every state. In later sections we will use the further abbreviation of **default** α **if** β for the action dynamic law α **if** $\alpha \wedge \beta$ and **nonexecutable** α **if** β for the fluent dynamic law $\bot$ **if** $\top$ **after** $\alpha \wedge \beta$. The **default** law represents that, where β is true, then α is true by default, whilst the **nonexecutable** is understood as false is derivable from a state where $\alpha \wedge \beta$ is true.

Current implementations of $\mathcal{C}+$, such as `iCCalc`, [1] are based on SAT solvers. Queries specify partial information about the values of fluent and action constants in the states and transitions of a run through the transition system, and answers take the form of the complete set of runs consistent with that specification.

In [6] it is shown that a subclass of action descriptions of $\mathcal{C}+$ (those without circular dependencies in causal laws) can be represented as normal logic programs (See Appendix) whose form is closely related to that of the Event Calculus [16]. We have adapted the form of those logic programs to suit our $\mathcal{C}+$ action descriptions for FSPs. The $\mathcal{EC}+$ formulation includes a translation of the specific causal theory, as well as core, domain-independent clauses to enable reasoning and ensure the semantics is respected. Domain specific facts include those of the predicate `domain/2` (giving the domain of a fluent constant) and `causes/5`—where a fact `causes(C2,V2,Act,C1,V1)` corresponds to the presence of a fluent dynamic law $C2{=}V2$ **if** $\top$ **after** $Act{=}\mathbf{t} \wedge C_1{=}V_1$.

In conjunction with information about what is initially true in a given run (facts of `init/2`) and what actions occur at different times (facts of `happens/2`), the sets of clauses of an $\mathcal{EC}+$ logic program have stable models that are in one-to-one correspondence with runs through the LTS defined by the action descriptions [6].

Inductive Logic Programming

ILP [23] is a symbolic machine-learning technique for computing a hypothesis H from a background theory B (a logic program) and examples ($E = E^- \cup E^+$) such that: (i) $B \cup H \models e^+$ for each $e^+ \in E^+$; (ii) $B \cup H \not\models e^-$ for each $e^- \in E^-$. For shorthand we write (i) and (ii) as $B \cup H \models E^+$ and $B \cup H \not\models E^-$. A hypothesis space is the set of all hypotheses $\{H_i\}$ that satisfies the conditions set above. To restrict the size of the hypothesis space, some ILP methods make use of *mode declarations* (*MD*), a form of language bias that specifies the syntactic form of the hypotheses to be learned.

[1] See `http://www.doc.ic.ac.uk/~rac101/iccalc/`

It contains both head and body declarations that describe predicates that may appear, the desired input and output behaviour and number of instantiations. We use $s(MD)$ to denote the set of all hypotheses satisfying *MD*.

Typically, B is assumed partial but correct. The ILP task is to generate a hypothesis H that extends B to explain the examples. ILP is applicable to problems in which B is partially incorrect and must be revised. Parts of the background suspected to be responsible are removed from B and put in the revisable theory T. The revision of a theory T involves applying a transformation to T to obtain a new theory H, denoted *r(T,H)*, by deleting rules, adding facts, adding conditions to rules or deleting conditions from rules. A repair is called *minimal* if the number of revision operations required to transform one theory into another (the sum of all deletions and additions) is minimal.

Definition 1. *An* inductive task *is a tuple* $\langle B, T, E, MD \rangle$*.* B *is the background theory,* T *is a revisable theory s.t.* $T \subseteq s(MD)$*,* $E = E^+ \cup E^-$ *is the examples and MD is the mode declaration. The logic program* H*, where* $H \subseteq s(MD)$ *and* $r(T,H)$*, is an* inductive solution *to the task* $\langle B, T, E, MD \rangle$ *iff* $B \cup H \models E^+$ *and* $B \cup H \not\models E^-$*.*

3 Running Example

Our running example is based on the production cell system [7]. It comprises two conveyor belts (feed belt and deposit belt), two products (a and b), a robot arm and two tools (drill and oven). The feed belt conveys raw products for the robot arm to pick up and process; the deposit belt conveys the processed products out of the cell. We define FSPs ARM, TOOL and RAW_PRODUCT (process names are in small capitals; actions are in italic), with the sets PRODUCTTYPES $= \{a, b\}$ and TOOLSET $= \{oven, drill\}$:

$$
\begin{aligned}
&\textsc{Arm} = \textsc{Idle},\\
&\quad \textsc{Idle} = ([p : \textsc{ProductTypes}].\textit{getFeedbelt} \rightarrow \textsc{Picked\_Up}[p]),\\
&\quad \textsc{Picked\_Up}[p : \textsc{ProductTypes}] = (\textit{put}[t : \textsc{ToolSet}][p] \rightarrow \textsc{Processing}[t][p]\\
&\qquad\quad \mid [p].\textit{putDepositbelt} \rightarrow \textsc{Idle} \mid [p].\textit{getFeedbelt} \rightarrow \textsc{Picked\_Up}[p]),\\
&\quad \textsc{Processing}[t : \textsc{ToolSet}][p : \textsc{ProductTypes}] = (\textit{get}[t][p] \rightarrow \textsc{Picked\_Up}[p]).\\
&\textsc{Tool}(T = \text{`}any) = (\textit{put}[T][p : \textsc{ProductTypes}] \rightarrow \textit{get}[T][p] \rightarrow \textsc{Tool}).\\
&\textsc{Raw\_Product}(P = \text{`}any) = ([P].\textit{available} \rightarrow [P].\textit{getFeedbelt} \rightarrow \textsc{Tool\_Available}\\
&\qquad\quad \mid [P].\textit{unavailable} \rightarrow \textsc{Raw\_Product}),\\
&\quad \textsc{Tool\_Available} = (\textit{put}[t : \textsc{ToolSet}][P] \rightarrow \textit{get}[t][P] \rightarrow \textsc{Tool\_Available}\\
&\qquad\quad \mid [P].\textit{putDepositbelt} \rightarrow \textsc{Raw\_Product}).
\end{aligned}
$$

where *'any* means any constant value assigned in the composed system. ARM, TOOL and RAW_PRODUCT are primitive process names. ARM has three local processes: IDLE, PICKED_UP and PROCESSING. It is initially idle. When it is idle it can pick up a product from the feed belt $[p : \textsc{ProductTypes}].\textit{getFeedbelt}$ (in which case it progresses to the PICKED_UP process). Once it has picked up a product p, it can either put the product in a tool t ($\textit{put}[t][p]$) and move to the PROCESSING phase or, from the same state, place it in the deposit belt $[p].\textit{putDepositbelt}$ and return to the IDLE phase, or it can get another product from the feed belt and continue in the same phase. If it puts in the tool for processing then it can remove the product from the tool and return to the PICKED_UP local process and continue from there. Note that in FSP, indices may appear either before or after action labels. The composite system is defined as below.

$||\text{TOOLS} = (\text{TOOL}(\mathit{oven}) \parallel \text{TOOL}(\mathit{drill}))$.
$||\text{RAW\_PRODUCTS} = (\text{RAW\_PRODUCT}(a) \parallel \text{RAW\_PRODUCT}(b))$.
$||\text{PRODUCTIONCELL} = (\text{ARM} \parallel \text{TOOLS} \parallel \text{RAW\_PRODUCTS})$.

Consider the property "The robot arm should not process products a and b at the same time". We are interested in checking if this situation is permissible in our composite system PRODUCTIONCELL. In the following sections we show how to detect automatically violations to such properties and repair the specification if any violations exist.

4 Compositional Verification in $\mathcal{C}+$

This section presents a new approach to verification for component-based systems represented as FSPs. It considers a fundamental FSP description as input and automatically translates it into the $\mathcal{C}+$ language. `iCCalc` is then used to verify that a property specified in $\mathcal{C}+$ holds in every run of the system. Our focus is on a class of properties (called safety properties [18]) which express the notion that no 'bad' state will be reached and that are expressible in Linear Temporal Logic (LTL) [21]. In what follows we give details of the FSP translation and verification using `iCCalc`.

Specifying FSPs in $\mathcal{C}+$

Translation from FSPs into $\mathcal{C}+$ starts from *fundamental FSPs*.

Definition 2. *Let A be a finite set of action labels, and $\mathcal{Q}$ a finite set of state labels, called* Q-labels*, of the form Q_i. Then a* fundamental FSP definition *has the form:*

$$\begin{aligned}\text{PROC} = {} & Q_0, \\ & Q_0 = (a_{1,1} \rightarrow Q_{1,1} \mid \cdots \mid a_{1,l_1} \rightarrow Q_{1,m_1}), \\ & \ldots, Q_n = (a_{n,1} \rightarrow Q_{n,1} \mid \cdots \mid a_{n,l_n} \rightarrow Q_{n,m_n}).\end{aligned}$$

where the $a_{i,j}$ are in A and the $Q_i, Q_{i,j}$ are in $\mathcal{Q}$. (It is clear that a fundamental FSP definition is also a full FSP.) We also use a representation of a fundamental FSP PROC *as 4-tuple of the form $(\mathcal{Q}, A, \mathit{trans}, \mathcal{Q}^*)$, where $\mathcal{Q}$ and A are as above, $\mathcal{Q}^* \subseteq \mathcal{Q}$ is the set of* initial local processes*, and trans $\subseteq \mathcal{Q} \times A \times \mathcal{Q}$ (the* transition relation*) represents the effect of the actions on the FSP as above: $(Q_i, a_{j,k}, Q_{l,m}) \in$ trans iff $Q_i = (\cdots \mid a_{j,k} \rightarrow Q_{l,m} \mid \cdots)$ forms part of the fundamental FSP definition. We will refer to fundamental FSPs just as 'FSPs' where this causes no confusion. Note that Q_0 always represents the initial local state of a process. For a fundamental FSP* PROC*, we use $\mathcal{Q}_{\text{PROC}}$, A_{PROC}, $\mathit{trans}_{\text{PROC}}$ and $\mathcal{Q}^*_{\text{PROC}}$ to refer to elements of the tuple representation.*

Any full FSP can be translated into a fundamental FSP representation behaviourally equivalent (allowing the same sequences of actions to be performed) to the original. The main features of fundamental FSPs compared to the original FSPs are: (i) definitions of sequences of action prefixes are split by creating new local processes for each action prefix, (ii) each range-indexed local process is replaced with a set of local processes, one for each value in the specified range, and (iii) each range-indexed action prefix is replaced with a choice of action prefix for each value in the range.

We work with the tuple-based representation of fundamental FSPs. The semantics is an LTS; an FSP $(\mathcal{Q}, A, \mathit{trans}, \mathcal{Q}^*)$ defines the LTS $(\mathcal{Q}, A, \{(Q, a, Q') \mid \mathit{trans}(Q, a) = Q'\}, \mathcal{Q}^*)$. Composition of the LTSs defined by fundamental FSPs is then given using the definitions in §2. (The LTS defined by a 'full' FSP is equivalent to that defined by its fundamental equivalent.) To illustrate, consider the (full) FSP definition ARM in our running example. The equivalent fundamental FSP is shown below, where we have marked the identity of states according to their Q-values.[2]

$$
\begin{aligned}
\text{ARM} = \; & Q_0,\\
& Q_0 = (\mathit{b.getFeedbelt} \rightarrow Q_1 \mid \mathit{a.getFeedbelt} \rightarrow Q_4),\\
& Q_1 = (\mathit{b.putDepositbelt} \rightarrow Q_0 \mid \mathit{b.getFeedbelt} \rightarrow Q_1 \mid \mathit{put.drill.b} \rightarrow Q_2\\
& \qquad \mid \mathit{put.oven.b} \rightarrow Q_3 \mid \mathit{a.getFeedbelt} \rightarrow Q_4),\\
& Q_2 = (\mathit{get.drill.b} \rightarrow Q_1), Q_3 = (\mathit{get.oven.b} \rightarrow Q_1),\\
& Q_4 = (\mathit{a.putDepositbelt} \rightarrow Q_0 \mid \mathit{b.getFeedbelt} \rightarrow Q_1 \mid \mathit{a.getFeedbelt} \rightarrow Q_4\\
& \qquad \mid \mathit{put.drill.a} \rightarrow Q_5 \mid \mathit{put.oven.a} \rightarrow Q_6),\\
& Q_5 = (\mathit{get.drill.a} \rightarrow Q_4), Q_6 = (\mathit{get.oven.a} \rightarrow Q_4).
\end{aligned}
$$

When translating into $\mathcal{C}+$, we work with sets of fundamental FSPs. The fluent constants will be their names; we use the fact that signatures of $\mathcal{C}+$ are multi-valued by setting the domain of each such fluent constant to be the Q-values (the states of the local processes) for the corresponding FSP. The only action constant is ACT, with domain the union of the sets of all action labels for each fundamental FSP. The causal laws of the translation encode the particular behaviour of the FSP. Consider again the ARM process. The $\mathcal{C}+$ translation has causal laws including:

$$
\begin{array}{ll}
\text{ARM}{=}Q_1 \textbf{ after } \text{ARM}{=}Q_0 \wedge \text{ACT}{=}\mathit{b.getFeedbelt} & \textbf{inertial } \text{ARM}\\
\text{ARM}{=}Q_4 \textbf{ after } \text{ARM}{=}Q_0 \wedge \text{ACT}{=}\mathit{a.getFeedbelt} & \textbf{default } \text{ACT}{=}\mathit{a.getFeedbelt}\\
\textbf{nonexecutable } \text{ACT}{=}\mathit{a.getFeedbelt} \textbf{ if } \text{ARM}{=}Q_2 & \\
\textbf{nonexecutable } \text{ACT}{=}\mathit{a.getFeedbelt} \textbf{ if } \text{ARM}{=}Q_3 &
\end{array}
$$

The **caused** laws encode the response to actions; the **inertial** law ensures that the local state of ARM continues in its current state unless it is caused to be otherwise; the **default** laws ensure that actions can occur by default; and the **nonexecutable** laws specify the conditions under which actions cannot occur. Further: $\sigma^{\mathrm{f}} = \{\text{ARM}\}$, $\mathit{dom}(\text{ARM}) = \{Q_0, Q_1, Q_2, Q_3, Q_4, Q_5, Q_6\}$, $\sigma^{\mathrm{a}} = \{\text{ACT}\}$, $\mathit{dom}(\text{ACT}){=}\{\mathit{put.drill}.\mathrm{X}$, $\mathit{put.oven}.\mathrm{X}$, $\mathit{get.drill}.\mathrm{X}$, $\mathit{get.oven}.\mathrm{X}$, $\mathrm{X}.\mathit{getFeedbelt}$, $\mathrm{X}.\mathit{putDepositbelt}\}$, for $\mathrm{X} \in \{a, b\}$.

Definition 3. *Let F be a set of fundamental FSPs, named* $\{\text{PROC}_1, \ldots, \text{PROC}_n\}$. *The $\mathcal{C}+$ translation of F is $\mathsf{F}_{\mathcal{C}+}$, where $\sigma^f = \{\text{PROC}_1, \ldots, \text{PROC}_n\}$ and $\sigma^a = \{\text{ACT}\}$, with:*

$$\mathit{dom}(\text{PROC}_i) = \{Q \mid \text{PROC}_i \in \mathsf{F},\ Q \in \mathcal{Q}_{\text{PROC}_i}\} \qquad \mathit{dom}(\text{ACT}) = \bigcup\{A_{\text{PROC}} \mid \text{PROC} \in \mathsf{F}\}$$

and where the laws of $\mathsf{F}_{\mathcal{C}+}$ are:

[2] The complete FSP description is available at
`http://www.doc.ic.ac.uk/~da04/sefm14/production_cell.fsp`

$$\{\text{PROC}{=}Q' \textbf{ after } \text{PROC}{=}Q \wedge \text{ACT}{=}a \mid \exists\text{PROC} \in \mathcal{F},\ (Q,a,Q') \in \textit{trans}_{\text{PROC}}\}$$
$$\cup \{\textbf{inertial } \text{PROC} \mid \text{PROC} \in \mathcal{F}\} \cup \{\textbf{default } \text{ACT}{=}a \mid a \in \textit{dom}(\text{ACT})\}$$
$$\cup \{\textbf{nonexecutable } \text{ACT}{=}a \textbf{ if } \text{PROC}{=}Q \mid a \in A_{\text{PROC}}, \neg\exists Q'((Q,a,Q') \in \textit{trans}_{\text{PROC}}\}$$

As with our illustration using the ARM process, the first set of laws encodes the response to actions; and the second set ensures that local states of processes persist unless caused to change. The third set of laws ensure that synchronisation is correctly modelled: where a is in the alphabet of the processes $\text{PROC}_1, \ldots, \text{PROC}_n$, then a can occur only when each of those processes is in an appropriate local state. Our first theorem shows that any transition in an LTS defined by a fundamental FSP is matched by a transition in the LTS defined by the corresponding $\mathcal{C}+$ action description.

Theorem 1. *Let $\mathcal{F}$ be a set of fundamental FSPs, $\mathcal{F} = \{\text{PROC}_1, \ldots, \text{PROC}_n\}$, such that (S, A, Δ, S_0) is the LTS defined by the composition $||\mathcal{F} = (\text{PROC}_1 \parallel \cdots \parallel \text{PROC}_n)$ (where no PROC_i itself contains a composition). Let $\mathcal{F}_{\mathcal{C}+}$ be the corresponding $\mathcal{C}+$ action description, with LTS (S', A', Δ', S_0'). Then there is a mapping $\lambda : S \rightarrow S'$ such that for any $(s_1, a, s_2) \in \Delta$, $(\lambda(s_1), \{\text{ACT}{=}a\}, \lambda(s_2)) \in \Delta'$.*
Proof. (See the Appendix.) □

This result allows us to prove that runs through the LTS defined by the FSP starting at the initial state are in 1-1 correspondence with runs through the LTS defined by the $\mathcal{C}+$ encoding, starting at its initial state.

Theorem 2. *Let $\mathcal{F}$ be a set of fundamental FSPs, $\mathcal{F} = \{\text{PROC}_1, \ldots, \text{PROC}_n\}$, such that (S, A, Δ, S_0) is the LTS defined by the composition $||\mathcal{F} = (F_1 \parallel \cdots \parallel F_n)$. Let $\mathcal{F}_{\mathcal{C}+}$ be the corresponding $\mathcal{C}+$ action description and (S', A', Δ', S_0') its LTS, where s_0' is $\{\text{PROC}_i{=}Q_i \mid Q_i \in s_0\}$. Then there is a run $(s_0, a_0, s_1, a_1, \ldots, a_{n-1}, s_n)$ through (S, A, Δ, S_0) iff there is a run $(s_0', a_0', s_1', a_1', \ldots, a_{n-1}', s_n')$ through (S', A', Δ', S_0'), with (i) $a_i' = \{\text{ACT}{=}a_i\}$ and (ii) $s_i' = \{\text{PROC}_k{=}Q_k \mid Q_k \in s_i\}$.*

Proof. Left to right is a trivial inductive consequence of Theorem 1. Right to left is a simple inductive proof on the length of runs (details omitted owing to space limits). □

As a result of Theorem 2, fundamental FSPs—and therefore also *all* FSPs—have a $\mathcal{C}+$ equivalent, in the sense that runs through the transition systems defined encode the same information. Tools developed for $\mathcal{C}+$ which allow different kinds of analysis can therefore be brought to bear in reasoning about properties of FSPs, which is the aim for the rest of this section. As the translation preserves the compositionality of the original FSP specification, one may remove and add the parts of the $\mathcal{C}+$ action description corresponding to different processes without harm, to streamline verification. It may be possible to make use of theorems in [27], which show when causal laws in an action description are redundant, in order to reduce the size of the translation. This would make the representation of the action description smaller, without affecting the state space.

Detecting Errors in `iCCalc`

We use `iCCalc` for verification. It takes as input a $\mathcal{C}+$ action description and a query as a partial specification of a run. Part of that query involves a specification of the initial state, which is extracted from the FSP specification (for each primitive process).

Verification involves checking runs of the system satisfy desirable properties, expressed in `iCCalc` as propositional formulas of time-stamped fluent and action constants `t:C=V`, where constant C must have the value V at t (the tth state or transition in the run). In this paper, we suppose properties are given in the `iCCalc` language. In [6] it is shown how bounded model checking over LTL can be expressed and performed in `iCCalc` using these constraints. The length of runs may be adjusted and incrementally increased as needed. The minimum length of runs needed to ensure completeness of the verification process can be calculated using methods such as those in [4].

To prove a property for a system composed of several FSP processes, we check the LTS defined by the corresponding $\mathcal{C}+$ action description satisfies the negation of that property. If a run satisfying its negation is found then the original property is violated and any run produced represents a counterexample. The constraint in the third argument of the query is expressed with the constant C being either ACT, the name of a primitive process or a fluent constant, and V representing the name of some action, a local state of a process or a fluent value ($\bot$ or $\top$). For instance, consider the property mention in §3 "The robot arm should not process two different product types at the same time". This is violated if there is a run leading to a state where the robot is processing product a and b concurrently. To check if this is possible in our model, we extend our production cell description in $\mathcal{C}+$ to include the fluent constants: $\{$*processing.a*, *processing.b*$\}$ where *processing.a* becomes true once the action *a.getFeedbelt* happens, and becomes false once *a.putDepositbelt* occurs and defined as false in the initial state. A similar definition is given to *processing.b*. In $\mathcal{C}+$:

$$\textit{processing.a}\ \textbf{after}\ \text{ACT} = \textit{a.getFeedbelt} \qquad \neg\textit{processing.a}\ \textbf{after}\ \text{ACT} = \textit{a.putDepositbelt}$$

$$\textbf{default}\ \neg\textit{processing.a}$$

We include predefined queries to our input files for `iCCalc`, using a predicate *rinit* in which the third parameter captures each process's initial local state (see Appendix). To check whether there is a run from the initial state leading to a state where a the robot's arm is processing two products a and b at the same time we prompt `iCCalc` with the query **query** $\textit{rinit}(1..m, [\textbf{max} : \textit{processing.a}, \textbf{max} : \textit{processing.b}])$ where `m` is an upper-bound on the length of the runs we interested in (in this example m was set to 100). In our running example, `iCCalc` finds all six solutions in the composed system of length four showing a case where the robot's arm gets product b from the feed belt while its processing product a and vice versa—we show three of the six here (the rest in the Appendix):

$$\mathsf{r}^1 = (s_0 \xrightarrow{a.available} s_1 \xrightarrow{b.available} s_2 \xrightarrow{a.getFeedbelt} s_3 \xrightarrow{b.getFeedbelt} s_4)$$
$$\mathsf{r}^2 = (s_0 \xrightarrow{a.available} s_1 \xrightarrow{a.getFeedbelt} s_2 \xrightarrow{b.available} s_3 \xrightarrow{b.getFeedbelt} s_4)$$
$$\mathsf{r}^3 = (s_0 \xrightarrow{b.available} s_1 \xrightarrow{a.available} s_2 \xrightarrow{a.getFeedbelt} s_3 \xrightarrow{b.getFeedbelt} s_4)$$

`iCCalc` produces runs representing all shortest (distinct) counterexamples to the original property from the initial state of the composed system. This is an advantage over other approaches as it allows the learning procedure (§5) to diagnose problematic runs simultaneously and hence suggest minimal repairs for all of them in a single iteration.

Because the underlying technology for the verification is propositional SAT-solving, verification is in general NP-complete w.r.t. the clausal representation `iCCalc` uses. In practice we have found the time `iCCalc` takes appears to be sensitive to the *structure* of the action descriptions; we leave the further investigation of this for future work.

5 Repairing Compositional Specifications

The detection of violation runs in the verification phase shows the composition of the processes violates the original property. However, the location of the errors and their exact causes may be unclear. Errors may occur in the composition of all or some of the processes, or be caused by a single component within the composite system. They may be caused by an over-constrained, under-specified or incorrect specification. Hence any repair must take all these considerations into account and ensure that any fix would not introduce further errors. In this section we use ILP to address these problems.

Encoding Process Descriptions

To enable the use of ILP, we first translate the $\mathcal{C}+$ theory for a set of fundamental FSPs, from §4, into an $\mathcal{EC}+$ logic program $\mathsf{F}_{\mathcal{EC}+}$ using a variant of the $\mathcal{C}+$ to $\mathcal{EC}+$ translation detailed in [6]. The mapping for **caused** and **nonexecutable** clauses follows that described in [6]. For each fluent constant f and action constant a the $\mathcal{EC}+$ program contains a fact fc(f) and av(a); if a value $v \in dom(c)$, there is a fact domain(c,v). Further, for every member v of the domain of a fluent constant, there is a fact fv(v). Thus, the Q-values for a process are recorded. For instance, the program obtained from encoding the extract of causal laws in §4 is:[3]

```
causes(arm, q1, b_getFeedbelt, arm, q0).
causes(arm, q4, a_getFeedbelt, arm, q0).
inertial FC :- fc(FC).
nonexecutable(a_getFeedbelt, arm, q2).
nonexecutable(a_getFeedbelt, arm, q3).
fc(arm).  av(b_getFeedbelt). av(b_getFeedbelt). fv(q0). fv(q1).
fv(q2). fv(q3). fv(q4). domain(arm, q0). domain (arm, q1).
domain(arm, q3). domain(arm, q4).
domain(act, b_getFeedbelt). domain(act, a_getFeedbelt).
```

Interpretations of $\mathsf{F}_{\mathcal{EC}+}$ are given with respect to an initial state, encoded using the predicate init/2, e.g., init(arm, q0), and runs expressed as a conjunction of happens literals. To capture multiple runs in our $\mathcal{EC}+$ description, we enrich the signature of $\mathcal{EC}+$ programs to include run constants σ^r and extend $\mathcal{EC}+$ predicates happens, caused and broken with an additional argument for runs, e.g., happens(a,t,r) means action a happens at time t in run r. The domain-independent axioms in $\mathcal{EC}+$ programs, *Axioms*, are updated accordingly. They fall into four parts, so that $Axioms = Ax_1 \cup Ax_2 \cup Ax_3 \cup Ax_4$. The first component, Ax_1, are inspired by the event calculus, and were given in [6]; they are described in the Appendix.

We introduce the predicate alphabet(c, a), which says that action a is in the alphabet of process c. The $\mathcal{C}+$ to $\mathcal{EC}+$ translation is extended to generate these for each action in the alphabet of every process in the $\mathcal{C}+$ theory. To ensure the semantics of FSP descriptions are preserved when learning repairs, we further include a set of constraints in the *Axioms*. Thus, Ax_2 is:

[3] The full program is available at
http://www.doc.ic.ac.uk/~da04/sefm14/production_cell.lp

```
:-  causes(C, V1, A, C, V),       :-  causes(C, V1, A,  C, V),
    causes(C, V2, A, C, V),           nonexecutable(A, C, V).
    V1 != V2.                     :-  causes(C, V, A, C, V0),
:-  causes(C, V, AV, C, V0),          not domain(C,V0).
    not domain(C,V).              :-  nonexecutable(A, C, V),
:-  causes(C, V1, A,  C, V),          not alphabet(C, A).
    not alphabet(C, A).
```

The top-left constraint ensures determinism: a process may not be caused to be in two different local states. The middle-left states that a process cannot be caused to be in a Q state outside its domain. The bottom-left specifies that only actions in the alphabet of a process may cause it to transit to a new state. The constraints on the right say that (i) an action cannot cause a system to evolve to a new state by executing a nonexecutable action, (ii) a process cannot be caused to transit to or from a state not within its domain and (iii) a process can only restrict the occurrence of actions within its alphabet.

In addition, it is necessary to ensure any changes to the existing process description result in a component specification that is deadlock-free. To do this, we include the following in *Axioms* which state collectively that a process must at least be able to evolve to one other state from every state in its domain. Our Ax_3 contains:

```
  exists_nextQstate(Process, From):-
       causes(Process, To, A, Process, From).
  :-  not exists_nextQstate(Process, From).
```

Note that although the above ensures that in any model of the $\mathcal{EC}+$ program, each process is deadlock-free, it does not guarantee this for the composite system. For the latter, the program must also include definitions of composite states reachable from the initial composite state, and a constraint similar to the above but with respect to composite states. We do not include these for lack of space.

To represent runs, we augment the language with the predicate `attempt(a, t, r)`, meaning there is an attempt to execute the action a at time t in run r. Consequently, a run is encoded in $\mathcal{EC}+$ as two sets: (i) a set of `attempt` facts, and (ii) a rule with `happens` literals in the body as defined below.

Definition 4. *Let* $\mathsf{r} = (s_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{n-1}} s_n)$ *be a run. Its* $\mathcal{EC}+$ *translation is* $\mathsf{r}_{\mathcal{EC}+} = \mathsf{r}_{Ext} \cup \mathsf{r}_{Hap}$ *where* $\mathsf{r}_{Ext} = \{\mathtt{attempt}(a_0,0,r) \ldots \mathtt{attempt}(a_{t-1},t-1,r)\}$*, and* r_{Hap} *is the clause* `run:-`*happens(*$a_0,0,r$*), . . ., happens(*$a_{t-1},t-1,r$*) with* `run` *is a predicate uniquely to run* r*. For simplicity, we use* $\alpha_{\mathsf{r}_{Hap}}$ *to denote the head of clause* r_{Hap}*.*

Finally, we further extend the set *Axioms* with Ax_4, below. The predicate `nonexecuta-ble` expresses that an action is cannot be performed at a time point within a run.

```
happens(A, T, R):-                :- attempt(A1, T, R),
    attempt(A, T, R),                 attempt(A2, T, R),
    not nonexcutable(A,T, R).         A1 != A2.
nonexcutable(A,T, R):-
    caused(C, V, T, R),
    nonexecutable(A, C, V).
```

The first rule means that an action happens if it has been attempted at a time in which it may occur. The second rule says that an action is not executable at a time t in a run r if it the system has evolved to a state from which it cannot occur. The constraint ensures that actions may not occur concurrently. This completes our $\mathcal{EC}+$ encoding.

Learning Repairs

Our proposed repair method locates the cause of the violation run detected during verification and revises the FSP descriptions to prevent these from occurring, whilst guaranteeing the modifications are consistent with the composite specification and do not introduce deadlock. To achieve this using ILP, the revision task $\langle B, T, E, MD\rangle$ is set by assigning specific elements of the $\mathsf{F}_{\mathcal{EC}}+$ program to B, T and E and defining *MD*.

When learning repairs for process descriptions, the revision task may be explicitly guided to explore the repair of specific components or all components within a given description. The ability to specify this is particularly useful if the specification contains process descriptions that are known to be correct or cannot be modified (as is the case in legacy systems). Hence when applying the revision task to a set of FSP descriptions, their $\mathcal{EC}+$ encoding is split into two sets: those for which revisions may be explored are added to T and those which are unmodifiable are included in B. Recall that a component specification in $\mathcal{EC}+$ is represented as a collection of $\mathbf{caused}$ and $\mathbf{nonexecutable}$ clauses in which its process label appears. The background B also includes *Axioms* and the $\mathcal{EC}+$ encoding of the runs obtained from Def. 4.

As mentioned in §4, `iCCalc` generates the shortest runs the composed system may execute to reach an undesirable state. The purpose of the repair is to identify necessary changes to the FSP descriptions so that these sequences are no longer permissible. Therefore for each violation run r, the constant appearing in the head of its r_{Hap} rule is included in the negative examples E^-.

As we are only interested in hypotheses that influence the set of runs permissible in the LTS of a composite system, we define the mode declaration to include rules that contain a *causes* and *nonexceutable* atom in the head. (Modification to the domain and alphabet of processes is discussed in the §6.) The repair task is defined as follows.

Definition 5. *Let $\mathsf{F} = \mathsf{F}^1, ..., \mathsf{F}^{m-1}, \mathsf{F}^m, ..., \mathsf{F}^n$ be a set of fundamental FSPs and R a set of violation runs such that each run in R exists in the LTS defined by $\mathsf{F}^1||...||\mathsf{F}^n$. F^* is said to be a repaired specification of F with respect to R if H is an inductive solution to the inductive task $\langle B, T, E, MD\rangle$ such that:*

$$B = \mathsf{F}^1_{\mathcal{EC}+} \cup ... \cup \mathsf{F}^{m-1}_{\mathcal{EC}+} \cup Axioms \cup \bigcup_{1<i} \{\mathsf{r}^i_{\mathcal{EC}+} | r^i \in \mathsf{R}\}; \quad T = \mathsf{F}^m_{\mathcal{EC}+} \cup ... \cup \mathsf{F}^n_{\mathcal{EC}+};$$

$$E^- = \bigcup_{1<i} \alpha_{r^i_{Hap}} \textit{ for all } r^i \in \mathsf{R}; \quad \textit{and } \mathsf{F}^*_{\mathcal{EC}+} = \mathsf{F}^1_{\mathcal{EC}+}, ..., \mathsf{F}^{m-1}_{\mathcal{EC}+} \cup H.$$

Thus far we have only discussed the use of violation runs within the proposed approach. Although not required, it is possible to integrate information about runs that satisfy the property being verified and should be preserved by the repair task. This is done by applying the translation in Def. 4 to each desirable run r_j and including the constant appearing in the head of its $\mathsf{r}^j_{\text{Hap}}$ in the set of positive examples E^+. It is important to note that F^* is not unique. The approach will produce all possible sets of minimal repairs from which the engineer may select which one to use.

To compute the necessary repairs, we use the non-monotonic ILP tool ASPAL [5]. The ASPAL learning algorithm maps an ILP task into Answer Set Programming (ASP) [10] and uses an ASP solver to abduce ground literals from which a hypothesis H is constructed. For our running example, we include $\text{TOOLS}_{\mathcal{EC}+}$, $\text{RAW\_PRODUCTS}_{\mathcal{EC}+}$

and $(\mathsf{r}^1_{\mathcal{EC}+} \cup ... \cup \mathsf{r}^6_{\mathcal{EC}+})$ in the background theory, $\textsc{Arm}_{\mathcal{EC}+}$ as the revisable theory and $(\alpha_{\mathsf{r}^1_{\text{Hap}}} \cup \ldots \cup \alpha_{\mathsf{r}^6_{\text{Hap}}})$ as the negative examples. ASPAL returns a revised theory where

```
causes(arm, q4, a_getFeedbelt, arm, q4),
causes(arm, q1, b_getFeedbelt, arm, q4)
```

are deleted, and the facts

```
nonexecutable(b_getFeedbelt, arm, q4),
nonexecutable(b_getFeedbelt, arm, q4)
```

are added. Although ASPAL is based on ASP which is NP-complete, we have noted that the repair computation time increases with the number of rules that need grounding by the ASP solver, and the number of revisions required. Heuristics for optimising the repair procedure require further study. As a result of the above, the LTS model of the $\mathcal{C}+$ equivalent of $\mathsf{F}^\star_{\mathcal{EC}+}$ is no longer consistent with the violations runs $\mathsf{r}^1, ..., \mathsf{r}^6$ [6].

Theorem 3. *Let F be a set of fundamental FSPs and R a set of violation runs such that each run in R exists in the LTS defined by F. Let $\mathsf{F}^\star$ be the repaired specification of F with respect to R. Then each run $r \in \mathsf{R}$ no longer exists in the LTS defined by $\mathsf{F}^\star$.*

Proof. Induction on run length, and using the fact that there is a unique stable model.

Although the theorem above proves the repair procedure eliminates the violation runs detected, the repair process does not guarantee longer violation runs are prohibited. Therefore, the verification and learning processes are iterated to detect any additional violation runs and repair the description accordingly. Violation runs from previous iterations are accumulated in E^- to ensure they are not made permissible by later revisions. The convergence of this process is guaranteed once no further violation runs up to the completeness bound discussed in §4 are detected. Since the repair is with respect to sets of violation runs, the approach takes fewer iterations than other approaches that integrate verification and learning. Once the cycle terminates, the final description is translated back into fundamental FSPs. In our running example, checking the $\mathcal{C}+$ equivalent of $(\mathsf{F}_{\mathcal{EC}+} \cup H)$ against the same property in `iCCalc` shows that no further violations exist and thus the approach converges in a single iteration. Consequently, the revised theory is mapped back into FSP through an inverse application of the translation in Def. 3 (space limitations prevent our providing the full translation). The final outcome is a repaired specification in which the only process modified is *Arm*:[4]

$$\begin{aligned}
\textsc{Arm} = \; & Q_0, \\
& Q_0 = (\textit{b.getFeedbelt} \rightarrow Q_1 \;\mid\; \textit{a.getFeedbelt} \rightarrow Q_4), \\
& Q_1 = (\textit{b.putDepositbelt} \rightarrow Q_0 \mid \textit{b.getFeedbelt} \rightarrow Q_1 \mid \textit{put.drill.b} \rightarrow Q_2 \\
& \qquad \mid \textit{put.oven.b} \rightarrow Q_3), \\
& Q_2 = (\textit{get.drill.b} \rightarrow Q_1), Q_3 = (\textit{get.oven.b} \rightarrow Q_1), \\
& Q_4 = (\textit{a.putDepositbelt} \rightarrow Q_0 \;\mid\; \textit{a.getFeedbelt} \rightarrow Q_4 \;\mid\; \textit{put.drill.a} \rightarrow Q_5 \\
& \qquad \mid\; \textit{put.oven.a} \rightarrow Q_6), \\
& Q_5 = (\textit{get.drill.a} \rightarrow Q_4), Q_6 = (\textit{get.oven.a} \rightarrow Q_4).
\end{aligned}$$

[4] Full FSP available at: `http://www.doc.ic.ac.uk/ da04/ sefm14/production_cell_revised.lts`

6 Conclusion and Related and Future Work

In this paper we have shown how to repair compositional specifications described in FSP, following a phase of automatic verification. We showed how an action language widely studied in A.I. ($\mathcal{C}+$) and ILP may be used to detect violations in properties expressible in LTL, and compute minimal repairs to individual components while consistency with the rest of the specification is maintained. This also involved defining a translation from FSPs into $\mathcal{C}+$, and thence into its logic-programming equivalent, $\mathcal{EC}+$; the correctness of our translations was proved. Although the paper focuses on revising FSP descriptions, we see the work presented here as holding exciting potential for solving a wide range of problems in component-based software engineering.

To the best of our knowledge, the translation from process algebras into logic programs has not been explored before. Several authors have proposed using logic programming to reason about software behaviour described in other formalisms [1,2,25]. [1] provides a translation for specifications expressed in LTL to event calculus logic programs. However, both that work and [2] only generate a single violation run at a time and hence require users to provide additional positive and negative example runs to ensure computed solutions are not over-generalised and to speed up the convergence of the approach. This limitation is overcome in our work by the generation of multiple violation runs in a single verification step. Further, the formalism and semantics used here allow the modelling of concurrency without the need to introduce special actions explicitly in the language (e.g., ‘tick’ actions in [19]), removing one threat to scalability. The work in [25] for generating Event Calculus logic programs from descriptions expressed in a tabular specification language and applying abductive logic programming to discover violations to a restricted class of invariants, namely ‘single-state’ invariants. That work finds a restricted class of violations, and cannot repair specifications.

[24] also use learning, to compute assumptions representing LTSs which, when composed with given components, guarantee a property’s satisfaction. The learning method is L*, which finds a regular language over a given alphabet and produces a deterministic finite-state machine that accepts the language. L* requires access to an oracle that iteratively accepts and rejects a generated string, and updates a table containing state information accordingly. ILP, by contrast, uses an expressive logic-based formalism capable of capturing state information among many other constructs such as constraints over the types of computable changes; this is not possible in L*.

Our approach is somewhat related to work on controller synthesis, e.g., [7]. For instance, techniques such as [7] automatically generate controllers that, together with a given model of the environment, satisfy a given property. Although these have shown good results,such techniques find at most one solution, even if many exist. Which controller is produced is chosen at random. Although our approach has only been demonstrated to learn revisions for existing process descriptions, we believe it may be adapted to compute all minimal process descriptions for a given alphabet.

In future work, we will investigate the use of our method to check liveness properties. We will modify the approach to handle revisions to the alphabet and extend the translation to embrace non-determinism and complex features such as abstractions and priorities. We will apply the work to model distribution problems [26], compositional specification synthesis from scenarios [17] and self-adaptive software [8].

References

1. Alrajeh, D., et al.: Elaborating requirements using model checking and inductive learning. IEEE Trans. Software Eng. 39(3), 361–383 (2013)
2. Borges, R., et al.: Learning and representing temporal knowledge in recurrent networks. IEEE TNN 22(12) (2011)
3. Clark, K.: Negation as failure. In: Readings in Nonmonotonic Reasoning, pp. 311–325 (1978)
4. Clarke, E., Kroning, D., Ouaknine, J., Strichman, O.: Completeness and complexity of bounded model checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 85–96. Springer, Heidelberg (2004)
5. Corapi, D., et al.: Inductive logic programming as abductive search. In: Proc. ICLP 2010, pp. 54–63 (2010)
6. Craven, R.: Execution mechanisms for the action language $\mathcal{C}+$. PhD thesis. Imperial College London (2007)
7. D'Ippolito, N., et al.: Synthesis of live behaviour models for fallible domains. In: Proc. ICSE 2011, pp. 211–220 (2011)
8. Filieri, A., et al.: A formal approach to adaptive software: continuous assurance of non-functional requirements. Formal Aspects of Computing 24, 163–186 (2012)
9. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. ICLP 1988, pp. 1070–1080 (1988)
10. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
11. Gelfond, M., Lifschitz, V.: Action languages. Electron. Trans. Artif. Intell. 2, 193–210 (1998)
12. Giunchiglia, E., et al.: Nonmonotonic causal theories. Artif. Intell. 153(1-2), 49–104 (2004)
13. Hoare, C.: Communicating Sequential Processes. Commun. ACM 21(8), 666–677 (1978)
14. Johnson, K., et al.: An incremental verification framework for component-based software systems. In: Proc. CBSE 2013, pp. 33–42 (2013)
15. Keller, R.: Formal verification of parallel programs. CACM 19(7), 371–384 (1976)
16. Kowalski, R., Sergot, M.: A logic-based calculus of events. New Generation Computing 4, 67–95 (1986)
17. Krka, I., et al.: Synthesizing partial component-level behavior models from system specifications. In: Proc. ESEC/FSE, pp. 305–314 (2009)
18. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. 16(3), 872–923 (1994)
19. Letier, E., et al.: Deriving event-based transition systems from goal-oriented requirements models. Autom. Softw. Eng. 15(2), 175–206 (2008)
20. Magee, J., Kramer, J.: Concurrency: state models and java programs. John Wiley and Sons (1999)
21. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Springer-Verlag New York, Inc. (1992)
22. Milner, R.: A Calculus of Communicating Systems. Springer, New York (1982)
23. Muggleton, S., Raedt, L.D.: Inductive logic programming: theory and methods. Journal of Log. Program. 19(20), 629–679 (1994)
24. Pasareanu, C., et al.: Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning. Formal Methods in System Design 32, 175–205 (2008)
25. Russo, A., Miller, R., Nuseibeh, B., Kramer, J.: An abductive approach for analysing event-based requirements specifications. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 22–37. Springer, Heidelberg (2002)
26. Sibay, G.E., Uchitel, S., Braberman, V., Kramer, J.: Distribution of modal transition systems. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 403–417. Springer, Heidelberg (2012)

27. Sergot, M., Craven, R.: Some Logical Properties of Nonmonotonic Causal Theories. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 198–210. Springer, Heidelberg (2005)

Appendix

Theorem 1. Let F be a set of fundamental FSPs, $\mathsf{F} = \{\textsc{Proc}_1, \ldots, \textsc{Proc}_n\}$, such that (S, A, Δ, S_0) is the LTS defined by the composition $||\mathsf{F} = (\textsc{Proc}_1 \parallel \cdots \parallel \textsc{Proc}_n)$ (where no $\textsc{Proc}_i$ itself contains a composition). Let $\mathsf{F}_{\mathcal{C}+}$ be the corresponding $\mathcal{C}+$ action description, with LTS (S', A', Δ', S'_0). Then there is a mapping $\lambda : S \to S'$ such that for any $(s_1, a, s_2) \in \Delta$, $(\lambda(s_1), \{\textsc{act}{=}a\}, \lambda(s_2)) \in \Delta'$.

Proof. Members $s \in S$ have the form $(Q_1, \ldots, Q_n)$ (§2). Let λ be s.t. $\lambda(Q_1, \ldots, Q_n)$ is $\{\textsc{Proc}_1{=}Q_1, \ldots, \textsc{Proc}_n{=}Q_n\}$. $\{\textsc{Proc}_1{=}Q_1, \ldots, \textsc{Proc}_n{=}Q_n\}$ is clearly a state of the LTS defined by $\mathsf{F}_{\mathcal{C}+}$. Assume $(s_1, a, s_2) \in \Delta$, and let $s_1 = (Q_1, \ldots, Q_n)$ and $s_2 = (Q'_1, \ldots, Q'_n)$. We must show $(\lambda(s_1), \{\textsc{act}{=}a\}, \lambda(s_2))$ is in Δ'. Let F_a be those members of F whose alphabets include a, i.e., $\mathsf{F}_a = \{\textsc{Proc} \in \mathsf{F} \mid a \in A_{\textsc{Proc}}\}$. Then for all i such that $1 \leqslant i \leqslant n$, if $\textsc{Proc}_i \in \mathsf{F}_a$ we have that $\mathit{trans}_{\textsc{Proc}}(Q_i, a) = Q'_i$, by definition of the transition systems defined by FSPs, and thus a law

$$\textsc{Proc}{=}Q'_i \textbf{ after } \textsc{Proc}_i{=}Q_i \wedge \textsc{act}{=}a$$

in $\mathsf{F}_{\mathcal{C}+}$. (Note also that if $\textsc{Proc}_i \notin \mathsf{F}_a$ then $\mathit{trans}_i(Q_i, a)$ is undefined.) Also using Definition 3 there must be a law

$$\textbf{default } \textsc{act}{=}a \textbf{ if } \textsc{Proc}^*_1{=}Q^*_1 \wedge \cdots \wedge \textsc{Proc}^*_m{=}Q^*_m$$

in $\mathsf{F}_{\mathcal{C}+}$, where $\textsc{Proc}^*_1, \ldots, \textsc{Proc}^*_m$ are the members of F_a and the $Q^*_1, \ldots, Q^*_m$ the corresponding values in s_1. The presence (and absence) of these causal laws, together with the $\mathbf{inertial}$ laws, means that there is a transition

$$(\{\textsc{Proc}_1{=}Q_1, \ldots, \textsc{Proc}_n{=}Q_n\}, \{\textsc{act}{=}a\}, \{\textsc{Proc}_1{=}Q'_1, \ldots, \textsc{Proc}_n{=}Q'_n\})$$

in Δ', which is precisely $(\lambda(s_1), \{\textsc{act}{=}a\}, \lambda(s_2))$ given our definition of λ. □

Logic Programs

A *normal logic program* is a set of rules of the form

$$L :- L_1, \ldots, L_m, \mathit{not}\ N_1, \ldots, \mathit{not}\ N_n \tag{1}$$

where the L, L_i and N_i are *atoms* and $0 \leqslant m$, $0 \leqslant n$. L is called the head of the rule whilst $L_1, \ldots, L_m, \mathit{not}\ N_1, \ldots, \mathit{not}\ N_n$ is referred to as the rule's body. A *literal* is an atom possibly preceded by *not*, where *not* is the negation as failure operator [3]. When a program does not contain *not* in its rules, it is called a *definite logic program.*

A (Herbrand) *model* M of a logic program P, is a set of ground atoms such that, for each ground instance C of a rule in P, M satisfies the head of C whenever it satisfies the body of C. A program is consistent if it has at least one model. A model M is *minimal* if it does not strictly include any other model. Definite programs always have a unique minimal model. Normal programs may have one, none, or several minimal models. When there is no unique minimal model, alternative semantics are often provided to single out specific models as the intended model.

Let P be a normal logic program and M be a set of atoms; then the *reduct* [9] of P with respect to M, written P^M, is

$$\{L : -L_1, \ldots, L_m \mid (1) \in P \wedge \forall i \leqslant n\ (N_i \in M)\}$$

The reduct of any normal logic program is a definite logic program, and therefore has a unique minimal model. If I is the minimal Herbrand model of P^M then M is said to be a stable model of P.

Core Axioms

We present the domain-independent axioms for our translation to $\mathcal{EC}+$ in §5. The first, event-calculus inspired component, Ax_1, are:

```
caused(C, V, 0, R) :-              caused(C, V, T1, R) :-
  domain(C, V),                      domain(C, V),
  init(C, V).                        0 < T1,  T is T1 - 1,
                                     caused(C, V, T, R),
caused(C, V, T1, R) :-               not broken(C, V, T, T1, R).
  domain(C, V),
  0 < T1,                          broken(C, V, T1, T2, R) :-
  T is T1 - 1,                       domain(C, V),
  happens(A, T, R),                  0 =< T1,  T1 < T2,
  domain(C, V0),                     domain(C, V1),  V1 != V,
  caused(C, V0, T, R),               happens(A, T1, R),
  causes(C, V, A, C, V0).            causes(C, V1, A, C, V),
                                     caused(C, V, T1, R).
```

The top-left axiom states that anything known true initially is caused to be true. The bottom-left axiom states that if there is a fluent dynamic law (causes/5) of the right form, and its conditions on the previous state and action performed hold, then the relevant fluent atom holds. The top-right axiom states that the values of fluent constants persist inertially by default; and the bottom-right, broken/5 axiom gives the circumstances overriding that default.

`iCCalc` Query

```
query(rinit(N), N,
 [0:arm=q0,   0:raw_product^[a]=q0,   0:raw_product^[b]=q0,
  0:tool^[drill]=q0,  0:tool^[oven]=q0,
  -(0:being_processed^[a]),
  -(0:being_processed^[b])]).
```

Remaining Violation Runs from §4

$$\mathsf{r}^4 = (s_0 \xrightarrow{b.available} s_1 \xrightarrow{b.getFeedbelt} s_2 \xrightarrow{a.available} s_3 \xrightarrow{a.getFeedbelt} s_4)$$
$$\mathsf{r}^5 = (s_0 \xrightarrow{a.available} s_1 \xrightarrow{b.available} s_2 \xrightarrow{b.getFeedbelt} s_3 \xrightarrow{a.getFeedbelt} s_4)$$
$$\mathsf{r}^6 = (s_0 \xrightarrow{b.available} s_1 \xrightarrow{a.available} s_2 \xrightarrow{b.getFeedbelt} s_3 \xrightarrow{a.getFeedbelt} s_4)$$